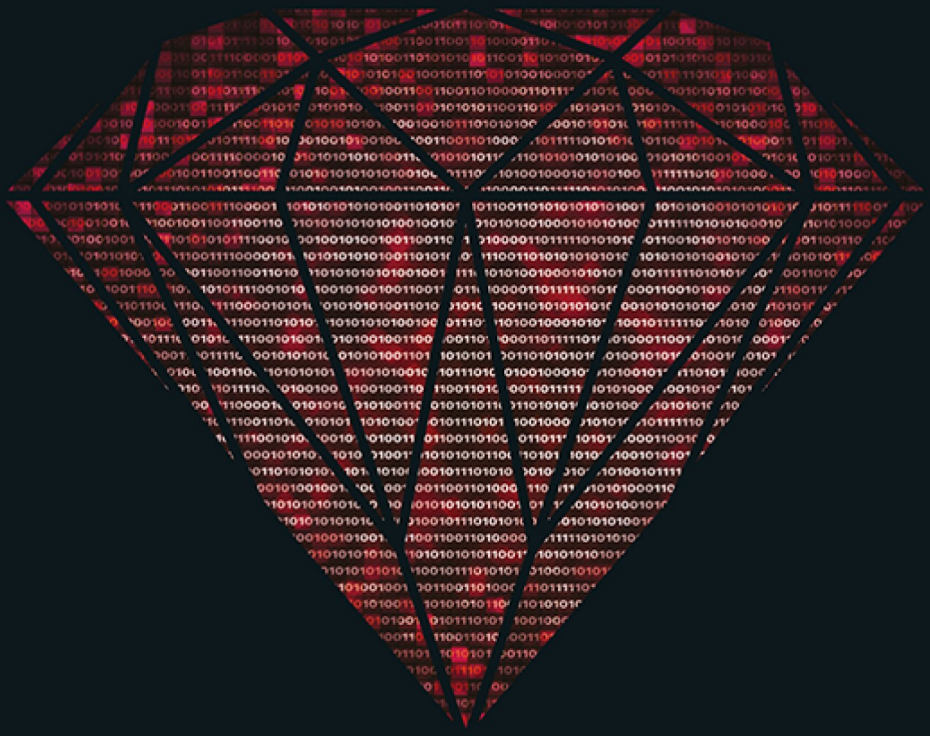
David A. Black

# RUBY

## Przewodnik programisty



Poznaj kolejny
język programowania!

Helion

Tytuł oryginału: The Well-Grounded Rubyist

Tłumaczenie: Piotr Pilch
Projekt okładki: Studio Gravite / Olsztyn;
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-1106-0

Original edition copyright © 2014 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2015 by HELION SA.
All rights reserved.

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 jakiejkolwiek 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 właścicieli.

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. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
*http://helion.pl/user/opinie/rubprp_ebook*
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- Poleć książkę na Facebook.com
- Kup w wersji papierowej
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

## Pochwały dotyczące pierwszego wydania

„Jeśli musisz się nauczyć języka Ruby, sięgnij właśnie po tę książkę! Więcej nie będziesz potrzebować".

**— John Griffin, Overstock.com**

„Znajomość języka Ruby przez Davida Blacka i jego umiejętność przekazania tej wiedzy są fenomenalne".

**— Philip Hallstrom, PJKH LLC**

„Długo oczekiwana dalsza część książki Davida zatytułowanej *Ruby for Rails*, tym razem z naciskiem na język Ruby".

**— Rick DeNatale, DenHaven Consulting**

„Pozycja obowiązkowa dla wszystkich, którzy chcą używać języka Ruby do nauki programowania obiektowego lub programowania w ogólności".

**— Robert Klemme, Orga Systems**

„David Black ponownie stworzył publikację, która powinna znaleźć się na biurku każdego projektanta aplikacji Ruby".

**— Robert Dempsey, Atlantic Dominion Solutions**

„Jak dotąd najlepsza książka dla chcących poznać język Ruby".

**— M. Edward (Ed) Borasky, ADP**

„Wewnętrzne mechanizmy działania języka Ruby? To tajemnica ujawniona przez Davida Blacka w tej książce".

**— Patrick Steger, Zühlke Engineering AG**

„Solidny, bazujący na znakomitych przykładach kodów przewodnik po języku Ruby, który jest idealny do nauki języka".

**— Doug Warren, Java Web Services**

„Chciałbym, aby ta książka była dostępna, gdy poznawałem środowisko Rails".

**— Patrick Dennis, Management Dynamics Inc.**

„Wspaniała dalsza część książki *Ruby for Rails*".

**— Mark Eagle, Intercontinental Exchange Inc.**

„Szczególnie polecane dla każdego, kto uczy się języka Ruby".

**— Greg Donald, VUMC**

„Wyśmienite i gruntowne wprowadzenie do języka Ruby".

**— Curtis Miller, Freebootr.com**

*Barbarze Aronstein Black*
*i dla upamiętnienia Charlesa L. Blacka Jr.*
*(1915 – 2001)*
*z wyrazami miłości.*
*Dziękuję za geny, dzięki którym*
*zostałem pisarzem.*

# Spis treści

*Przedmowa 17*
*Przedmowa do pierwszego wydania 19*
*Podziękowania 21*
*O książce 23*

**CZĘŚĆ I FUNDAMENTY JĘZYKA RUBY ..... 29**

*Rozdział 1. Wprowadzenie do nauki języka Ruby 31*

1.1. Ogólne wprowadzenie do języka Ruby 32
*1.1.1. Niezbędna składnia języka Ruby 33*
*1.1.2. Różnorodność identyfikatorów języka Ruby 35*
*1.1.3 Wywołania metod, komunikaty i obiekty języka Ruby 37*
*1.1.4. Tworzenie i zapisywanie prostego programu 39*
*1.1.5. Kierowanie programu do interpretera języka Ruby 40*
*1.1.6. Operacje wejścia-wyjścia związane z plikami i danymi wprowadzanymi przy użyciu klawiatury 42*
1.2. Anatomia instalacji języka Ruby 45
*1.2.1. Podkatalog standardowych bibliotek języka Ruby 46*
*1.2.2. Katalog rozszerzeń języka C (RbConfig::CONFIG[archdir]) 46*
*1.2.3. Katalogi site_ruby (RbConfig::CONFIG[sitedir]) i vendor_ruby (RbConfig::CONFIG[vendordir]) 47*
*1.2.4. Katalog gems 47*
1.3. Rozszerzenia i biblioteki programistyczne języka Ruby 48
*1.3.1. Ładowanie plików i rozszerzeń zewnętrznych 48*
*1.3.2. Ładowanie pliku określonego w domyślnej ścieżce ładowania 49*
*1.3.3. Żądanie składnika 50*
*1.3.4. Polecenie require_relative 52*
1.4. Standardowe narzędzia i aplikacje języka Ruby 52
*1.4.1. Opcje wiersza poleceń interpretera 53*
*1.4.2. Omówienie interaktywnego interpretera języka Ruby irb 57*
*1.4.3. Narzędzia ri i Rdoc 59*
*1.4.4. Narzędzie do zarządzania zadaniami rake 60*
*1.4.5. Instalowanie pakietów za pomocą polecenia gem 62*
1.5. Podsumowanie 64

*Rozdział 2. Obiekty, metody i zmienne lokalne 65*

2.1. Komunikowanie się z obiektami 66
*2.1.1. Język Ruby i obiektowość 66*
*2.1.2. Tworzenie obiektu ogólnego 67*

*2.1.3. Metody pobierające argumenty 69*
*2.1.4. Wartość zwracana metody 70*
2.2. Tworzenie obiektu: działanie biletu 71
*2.2.1. Obiekt biletu — przede wszystkim działanie 71*
*2.2.2. Odpytywanie obiektu biletu 72*
*2.2.3. Skracanie kodu obiektu biletu za pomocą interpolacji łańcuchów 73*
*2.2.4. Dostępność biletu: wyrażanie stanu boolowskiego w metodzie 74*
2.3. Wbudowane zachowania obiektu 76
*2.3.1. Unikatowe identyfikowanie obiektów za pomocą metody object_id 77*
*2.3.2. Uzyskiwanie możliwości obiektu za pomocą metody respond_to? 78*
*2.3.3. Wysyłanie komunikatów do obiektów za pomocą metody send 79*
2.4. Dokładna analiza argumentów metody 80
*2.4.1. Argumenty wymagane i opcjonalne 80*
*2.4.2. Wartości domyślne argumentów 81*
*2.4.3. Kolejność parametrów i argumentów 82*
*2.4.4. Działania niedozwolone w przypadku list argumentów 85*
2.5. Zmienne lokalne i przypisywanie do zmiennych 85
*2.5.1. Zmienne, obiekty i odwołania 87*
*2.5.2. Odwołania w przypisaniu do zmiennej i ponowne przypisanie 89*
*2.5.3. Odwołania i argumenty metody 91*
*2.5.4. Zmienne lokalne i przypominające je elementy 92*
2.6. Podsumowanie 93

*Rozdział 3. Organizowanie obiektów za pomocą klas 95*

3.1. Klasy i instancje 96
*3.1.1. Metody instancji 97*
*3.1.2. Przesłanianie metod 97*
*3.1.3. Ponowne otwieranie klas 98*
3.2. Zmienne instancji i stan obiektu 100
*3.2.1. Inicjowanie obiektu ze stanem 102*
3.3. Metody ustawiające 103
*3.3.1. Znak równości (=) w nazwach metod 104*
*3.3.2. „Lukier" składniowy dla metod przypominających przypisania 105*
*3.3.3. Pełnia możliwości metod ustawiających 106*
3.4. Atrybuty i rodzina metod attr_* 108
*3.4.1. Automatyzowanie tworzenia atrybutów 108*
*3.4.2. Podsumowanie metod attr_* 111*
3.5. Dziedziczenie i hierarchia klas języka Ruby 111
*3.5.1. Pojedyncze dziedziczenie: po jednym dla klienta 113*
*3.5.2. Przodkowie obiektów i nie do końca brakujące łącze: klasa Object 113*
*3.5.3. Starszy brat El Viejo: BasicObject 114*
3.6. Klasy jako obiekty i odbiorcy komunikatów 115
*3.6.1. Tworzenie obiektów klasy 115*
*3.6.2. Wywoływanie metod przez obiekty klasy 116*
*3.6.3. Metoda pojedynczego obiektu 117*
*3.6.4. Kiedy i dlaczego należy tworzyć metodę klasy? 119*
*3.6.5. Porównanie metod klasy z metodami instancji 120*

3.7. Szczegóły dotyczące stałych 120
*3.7.1. Podstawowe zastosowanie stałych 121*
*3.7.2. Porównanie ponownego przypisania z modyfikowaniem stałych 123*
3.8. „Natura" i „wychowanie" w przypadku obiektów języka Ruby 124
3.9. Podsumowanie 126

*Rozdział 4. Moduły i organizacja programu 127*

4.1. Podstawowe informacje dotyczące tworzenia i używania modułów 128
*4.1.1. Moduł hermetyzujący „podobieństwo do stosu" 129*
*4.1.2. Dodawanie modułu do klasy 131*
*4.1.3. Bardziej zaawansowane użycie modułów 133*
4.2. Moduły, klasy i wyszukiwanie metody 135
*4.2.1. Demonstracja podstaw wyszukiwania metod 135*
*4.2.2. Definiowanie tej samej metody więcej niż raz 138*
*4.2.3. Sposób działania instrukcji prepend 141*
*4.2.4. Podsumowanie reguł wyszukiwania metod 142*
*4.2.5. Nawigacja w obrębie ścieżki wyszukiwania metod za pomocą słowa kluczowego super 143*
4.3. Metoda method_missing 145
*4.3.1. Łączenie metody method_missing ze słowem kluczowym super 146*
4.4. Projekt i nadawanie nazw w przypadku klas i modułów 150
*4.4.1. Dodawanie modułów do klas i/lub dziedziczenie 151*
*4.4.2. Zagnieżdżanie modułów i klas 153*
4.5. Podsumowanie 154

*Rozdział 5. Obiekt domyślny (self), zasięg i widoczność 155*

5.1. Obiekt self, czyli obiekt bieżący/domyślny 156
*5.1.1. Co zostaje obiektem self i w jakim miejscu? 157*
*5.1.2. Obiekt self najwyższego poziomu 158*
*5.1.3. Obiekt self w definicjach klas, modułów i metod 159*
*5.1.4. Obiekt self jako domyślny odbiorca komunikatów 162*
*5.1.5. Określanie zmiennych instancji za pośrednictwem obiektu self 164*
5.2. Określanie zasięgu 166
*5.2.1. Zasięg globalny i zmienne globalne 166*
*5.2.2. Zasięg lokalny 169*
*5.2.3. Interakcja między zasięgiem lokalnym i obiektem self 171*
*5.2.4. Zasięg i określanie stałych 173*
*5.2.5. Składnia zmiennych klasy, zasięg i widoczność 175*
5.3. Wdrażanie reguł uzyskiwania dostępu do metod 182
*5.3.1. Metody prywatne 182*
*5.3.2. Metody chronione 186*
5.4. Tworzenie i używanie metod najwyższego poziomu 187
*5.4.1. Definiowanie metody najwyższego poziomu 187*
*5.4.2. Predefiniowane (wbudowane) metody najwyższego poziomu 188*
5.5. Podsumowanie 189

*Rozdział 6. Techniki przepływu sterowania 191*

6.1. Warunkowe wykonywanie kodu 192
*6.1.1. Instrukcja if i powiązane z nią instrukcje 192*
*6.1.2. Składnia przypisania w treści instrukcji warunkowych i testach 197*
*6.1.3. Instrukcje case 200*
6.2. Powtarzanie działań za pomocą pętli 205
*6.2.1. Bezwarunkowe wykonywanie pętli za pomocą metody loop 205*
*6.2.2. Warunkowe wykonywanie pętli za pomocą słów kluczowych while i until 206*
*6.2.3. Wykonywanie pętli na bazie listy wartości 209*
6.3. Iteratory i bloki kodu 209
*6.3.1. Elementy iteracji 209*
*6.3.2. Iteracja w zwykłym stylu 210*
*6.3.3. Anatomia wywołania metody 210*
*6.3.4. Porównanie nawiasów klamrowych oraz pary słów kluczowych do i end w składni bloku kodu 211*
*6.3.5. Implementowanie metody times 213*
*6.3.6. Ważność metody each 214*
*6.3.7. Od metody each do metody map 216*
*6.3.8. Parametry bloku i zasięg zmiennych 218*
6.4. Obsługa błędów i wyjątki 221
*6.4.1. Zgłaszanie wyjątków i stosowanie dla nich klauzuli rescue 221*
*6.4.2. Słowo kluczowe rescue na ratunek! 222*
*6.4.3. Jawne zgłaszanie wyjątków 224*
*6.4.4. Przechwytywanie wyjątku w klauzuli rescue 225*
*6.4.5. Klauzula ensure 227*
*6.4.6. Tworzenie własnych klas wyjątków 228*
6.5. Podsumowanie 229

CZĘŚĆ II WBUDOWANE KLASY I MODUŁY .....231

*Rozdział 7. Wbudowane elementy podstawowe 233*

7.1. Konstruktory literałów języka Ruby 234
7.2. Powracający „lukier” składniowy 235
*7.2.1. Definiowanie operatorów przez określanie metod 236*
*7.2.2. Dostosowywanie operatorów jednoargumentowych 238*
7.3. Metody z nazwą zakończoną wykrzyknikiem i metody „niebezpieczne” 239
*7.3.1. Destrukcyjne efekty (zmieniające odbiorcę) uznawane za „niebezpieczne” 240*
*7.3.2. Destrukcyjność i „niebezpieczeństwo” zmieniają się niezależnie 241*
7.4. Metody wbudowane i niestandardowe to_* służące do konwersji 242
*7.4.1. Konwersja łańcucha: metoda to_s 243*
*7.4.2. Konwersja tablic za pomocą metody to_a i operatora * 246*
*7.4.3. Konwersja liczb za pomocą metod to_i i to_f 247*
*7.4.4. Metody z rodziny to_* przyjmujące role 248*
7.5. Stany i obiekty boolowskie oraz obiekt nil 250
*7.5.1. Obiekty true i false jako stany 251*
*7.5.2. Obiekty true i false 252*
*7.5.3. Obiekt specjalny nil 254*

7.6. Porównanie dwóch obiektów 255
*7.6.1. Testy równości 255*
*7.6.2. Porównania i moduł Comparable 256*
7.7. Inspekcja możliwości obiektów 258
*7.7.1. Wyświetlanie listy metod obiektu 259*
*7.7.2. Odpytywanie obiektów klas i modułów 260*
*7.7.3. Listy filtrowanych i wybranych metod 261*
7.8. Podsumowanie 261

## *Rozdział 8. Łańcuchy, symbole i inne obiekty skalarne 263*

8.1. Zastosowanie łańcuchów 264
*8.1.1. Notacja łańcuchów 264*
*8.1.2. Podstawowe modyfikacje łańcuchów 268*
*8.1.3. Odpytywanie łańcuchów 272*
*8.1.4. Porównywanie łańcuchów i określanie ich kolejności 275*
*8.1.5. Transformacja łańcuchów 276*
*8.1.6. Konwersje łańcuchów 279*
*8.1.7. Kodowanie łańcuchów — krótkie wprowadzenie 280*
8.2. Symbole i ich zastosowanie 282
*8.2.1. Główne cechy charakterystyczne symboli 283*
*8.2.2. Symbole i identyfikatory 284*
*8.2.3. Symbole w praktyce 286*
*8.2.4. Porównanie łańcuchów i symboli 288*
8.3. Obiekty liczbowe 289
*8.3.1. Klasy liczbowe 290*
*8.3.2. Wykonywanie operacji arytmetycznych 290*
8.4. Czas i daty 292
*8.4.1. Tworzenie instancji obiektów daty/czasu 293*
*8.4.2. Metody odpytywania obiektów daty i czasu 295*
*8.4.3. Metody formatujące datę i czas 296*
*8.4.4. Metody konwersji daty i czasu 297*
8.5. Podsumowanie 299

## *Rozdział 9. Obiekty kolekcji i kontenerów 301*

9.1. Porównanie tablic i tablic asocjacyjnych 302
9.2. Przetwarzanie kolekcji za pomocą tablic 304
*9.2.1. Tworzenie nowej tablicy 304*
*9.2.2. Wstawianie, pobieranie i usuwanie elementów tablicy 308*
*9.2.3. Łączenie tablic z innymi tablicami 311*
*9.2.4. Transformacje tablic 312*
*9.2.5. Odpytywanie tablic 314*
9.3. Tablice asocjacyjne 315
*9.3.1. Tworzenie nowej tablicy asocjacyjnej 315*
*9.3.2. Wstawianie, pobieranie i usuwanie par tablic asocjacyjnych 317*
*9.3.3. Określanie domyślnych wartości i zachowania tablic asocjacyjnych 319*
*9.3.4. Łączenie tablic asocjacyjnych z innymi tablicami asocjacyjnymi 320*
*9.3.5. Transformacje tablic asocjacyjnych 321*

*9.3.6. Odpytywanie tablic asocjacyjnych 322*
*9.3.7. Tablice asocjacyjne jako ostatnie argumenty metody 323*
*9.3.8. Powrót do składni argumentów: argumenty nazwane (słów kluczowych) 324*
9.4. Zakresy 326
*9.4.1. Tworzenie zakresu 327*
*9.4.2. Logika włączenia do zakresów 328*
9.5. Zbiory 330
*9.5.1. Tworzenie zbiorów 331*
*9.5.2. Modyfikowanie elementów zbioru 331*
*9.5.3. Podzbiory i nadzbiory 334*
9.6. Podsumowanie 335

*Rozdział 10. Kolekcje: moduł Enumerable i klasa Enumerator 337*

10.1. Zapewnianie możliwości wyliczania za pośrednictwem metody each 338
10.2. Zapytania boolowskie dotyczące modułu Enumerable 340
10.3. Wyszukiwanie i wybieranie obiektów wyliczeniowych 343
*10.3.1. Uzyskiwanie pierwszego dopasowania za pomocą metody find 343*
*10.3.2. Uzyskiwanie wszystkich dopasowań za pomocą metod find_all (inaczej select) i reject 345*
*10.3.3. Wybieranie dopasowań operatora równości === za pomocą metody grep 345*
*10.3.4. Organizowanie wyników wybierania za pomocą metod group_by i #partition 347*
10.4. Operacje wyliczeniowe dotyczące elementów 348
*10.4.1. Metoda first 348*
*10.4.2. Metody take i drop 350*
*10.4.3. Metody min i max 350*
10.5. Metody powiązane z metodą each 352
*10.5.1. Metoda reverse_each 352*
*10.5.2. Metoda each_with_index (oraz metoda each.with_index) 352*
*10.5.3. Metody each_slice i each_cons 353*
*10.5.4. Metoda cycle 354*
*10.5.5. Zmniejszanie obiektu wyliczeniowego za pomocą metody inject 355*
10.6. Metoda map 356
*10.6.1. Wartość zwracana metody map 357*
*10.6.2. Odwzorowywanie wewnętrzne za pomocą metody map! 358*
10.7. Łańcuchy jako quazi-obiekty wyliczeniowe 359
10.8. Sortowanie obiektów wyliczeniowych 360
*10.8.1. W jaki sposób moduł Comparable przydaje się (albo nie) podczas sortowania obiektów wyliczeniowych? 362*
*10.8.2. Definiowanie logiki kolejności sortowania przy użyciu bloku 363*
*10.8.3. Zwięzłe sortowanie za pomocą metody sort_by 364*
10.9. Enumeratory i następny wymiar możliwości wyliczania 364
*10.9.1. Tworzenie enumeratorów z blokiem kodu 365*
*10.9.2. Powiązanie enumeratorów z innymi obiektami 367*
*10.9.3. Niejawne tworzenie enumeratorów za pomocą wywołań iteratora bez użycia bloku kodu 369*

10.10. Semantyka enumeratorów i ich zastosowanie 369
*10.10.1. Sposób użycia metody each enumeratora 370*
*10.10.2. Ochrona obiektów za pomocą enumeratorów 372*
*10.10.3. Rozbudowana iteracja z wykorzystaniem enumeratorów 373*
*10.10.4. Zapewnianie możliwości wyliczania za pomocą enumeratora 374*
10.11. Tworzenie łańcucha metod enumeratora 375
*10.11.1. Zmniejszanie liczby obiektów pośrednich 376*
*10.11.2. Indeksowanie obiektów wyliczeniowych za pomocą metody with_index 377*
*10.11.3. Operacje alternatywy wykluczającej na łańcuchach z wykorzystaniem enumeratorów 378*
10.12. „Leniwe" enumeratory 380
*10.12.1. Problem FizzBuzz w przypadku „leniwego" enumeratora 381*
10.13. Podsumowanie 382

*Rozdział 11. Wyrażenie regularne i operacje na łańcuchach oparte na wyrażeniach regularnych 385*

11.1. Czym są wyrażenia regularne? 386
11.2. Tworzenie wyrażeń regularnych 387
*11.2.1. Wyświetlanie wzorców 387*
*11.2.2. Proste dopasowywanie za pomocą wyrażeń regularnych literału 388*
11.3. Tworzenie wzorca w wyrażeniu regularnym 389
*11.3.1. Znaki literału we wzorcach 389*
*11.3.2. Znak wieloznaczny kropki (.) 390*
*11.3.3. Klasy znaków 390*
11.4. Dopasowywanie i przechwytywanie podłańcuchów oraz obiekt MatchData 392
*11.4.1. Przechwytywanie dopasowań podrzędnych za pomocą nawiasów okrągłych 392*
*11.4.2. Dopasowanie pomyślne i zakończone niepowodzeniem 394*
*11.4.3. Dwa sposoby uzyskiwania przechwyceń 395*
*11.4.4. Inne informacje zawarte w obiekcie MatchData 397*
11.5. Dostrajanie wyrażeń regularnych za pomocą kwantyfikatorów, zakotwiczeń i modyfikatorów 398
*11.5.1. Ograniczanie dopasowań za pomocą kwantyfikatorów 398*
*11.5.2. „Zachłanne" (i inne) kwantyfikatory 400*
*11.5.3. Zakotwiczenia i asercje wyrażeń regularnych 403*
*11.5.4. Modyfikatory 406*
11.6. Wzajemna konwersja łańcuchów i wyrażeń regularnych 408
*11.6.1. Idiomy związane z przepływem od łańcucha do wyrażenia regularnego 408*
*11.6.2. Przejście od wyrażenia regularnego do łańcucha 410*
11.7. Typowe metody używające wyrażeń regularnych 411
*11.7.1. Metoda String#scan 411*
*11.7.2. Metoda String#split 412*
*11.7.3. sub/sub! i gsub/gsub! 414*
*11.7.4. Równość przypadków i metoda grep 415*
11.8. Podsumowanie 417

*Rozdział 12. Operacje wejścia-wyjścia i operacje na plikach 419*

12.1. Struktura systemu operacji wejścia-wyjścia w języku Ruby 420
*12.1.1. Klasa IO 420*
*12.1.2. Obiekty IO jako obiekty wyliczeniowe 421*
*12.1.3. Obiekty STDIN, STDOUT i STDERR 422*
*12.1.4. Trochę więcej o danych wprowadzanych za pomocą klawiatury 423*
12.2. Podstawowe operacje na plikach 424
*12.2.1. Podstawy operacji odczytywania z plików 424*
*12.2.2. Odczytywanie plików oparte na wierszach 425*
*12.2.3. Odczytywanie plików w oparciu o bajty i znaki 426*
*12.2.4. Szukanie pozycji w pliku oraz odpytywanie o nią 427*
*12.2.5. Odczytywanie plików za pomocą metod klasy File 428*
*12.2.6. Zapisywanie w plikach 429*
*12.2.7. Użycie bloków do określania zasięgu operacji na plikach 430*
*12.2.8. Możliwości wyliczeniowe plików 431*
*12.2.9. Wyjątki i błędy dotyczące plikowych operacji wejścia-wyjścia 432*
12.3. Odpytywanie obiektów IO i File 433
*12.3.1. Uzyskiwanie informacji z klasy File i modułu FileTest 434*
*12.3.2. Uzyskiwanie informacji o plikach za pomocą klasy File::Stat 435*
12.4. Modyfikowanie katalogów za pomocą klasy Dir 436
*12.4.1. Wczytywanie pozycji katalogu 436*
*12.4.2. Modyfikowanie i odpytywanie katalogów 439*
12.5. Narzędzia plikowe biblioteki standardowej 440
*12.5.1. Moduł FileUtils 440*
*12.5.2. Klasa Pathname 442*
*12.5.3. Klasa StringIO 444*
*12.5.4. Moduł open-uri 446*
12.6. Podsumowanie 446

CZĘŚĆ III DYNAMIKA JĘZYKA RUBY .....449

*Rozdział 13. Indywidualizacja obiektów 451*

13.1. Tam, gdzie znajdują się metody pojedynczego obiektu: klasa pojedynczych obiektów 452
*13.1.1. Podwójne określanie za pomocą klas pojedynczych obiektów 453*
*13.1.2. Bezpośrednie sprawdzanie i modyfikowanie klasy pojedynczych obiektów 454*
*13.1.3. Klasy pojedynczych obiektów w ścieżce wyszukiwania metod 456*
*13.1.4. Metoda singleton_class 461*
*13.1.5. Obszerne omówienie metod klasy 461*
13.2. Modyfikowanie podstawowych klas i modułów języka Ruby 463
*13.2.1. Zagrożenia związane z modyfikowaniem podstawowych funkcjonalności 463*
*13.2.2. Zmiany addytywne 469*
*13.2.3. Zmiany z przekazaniem 469*
*13.2.4. Zmiany poszczególnych obiektów za pomocą metody extend 472*
*13.2.5. Użycie doprecyzowań do zmiany zachowania podstawowych obiektów 475*

13.3. BasicObject jako przodek i klasa 476
*13.3.1. Użycie klasy BasicObject 476*
*13.3.2. Implementowanie podklasy klasy BasicObject 478*
13.4. Podsumowanie 480

*Rozdział 14. Obiekty umożliwiające wywoływanie i uruchamianie 483*

14.1. Podstawowe funkcje anonimowe: klasa Proc 484
*14.1.1. Obiekty Proc 484*
*14.1.2. Obiekty Proc i bloki oraz różnice między nimi 485*
*14.1.3. Konwersje dotyczące bloków i obiektów Proc 487*
*14.1.4. Użycie metody Symbol#to_proc do zapewnienia zwięzłości 490*
*14.1.5. Obiekty Proc w roli domknięć 491*
*14.1.6. Argumenty i parametry obiektu Proc 494*
14.2. Tworzenie funkcji za pomocą metody lambda i konstruktora -> 494
14.3. Metody jako obiekty 496
*14.3.1. Przechwytywanie obiektów Method 496*
*14.3.2. Powody używania metod jako obiektów 497*
14.4. Rodzina metod eval 499
*14.4.1. Przetwarzanie dowolnych łańcuchów jako kodu za pomocą metody eval 499*
*14.4.2. Zagrożenia stwarzane przez metodę eval 501*
*14.4.3. Metoda instance_eval 501*
*14.4.4. Użycie metody class_eval (inaczej module_eval) 503*
14.5. Równoległe wykonywanie za pomocą wątków 505
*14.5.1. Kończenie działania, zatrzymywanie i uruchamianie wątków 506*
*14.5.2. Serwer dat z wątkami 508*
*14.5.3. Tworzenie serwera rozmów sieciowych używającego gniazd i wątków 509*
*14.5.4. Wątki i zmienne 511*
*14.5.5. Modyfikowanie kluczy wątków 512*
14.6. Wykonywanie poleceń systemowych w obrębie programów Ruby 515
*14.6.1. Metoda system i odwrócone apostrofy 515*
*14.6.2. Komunikacja z programami za pośrednictwem metod open i Open3.popen3 518*
14.7. Podsumowanie 521

*Rozdział 15. Wywołania zwrotne, „haki" i introspekcja w czasie wykonywania kodu 523*

15.1. Wywołania zwrotne i „haki" 524
*15.1.1. Przechwytywanie nierozpoznanych komunikatów za pomocą metody method_missing 525*
*15.1.2. Wychwytywanie operacji dołączania i wstawiania na początku 528*
*15.1.3. Przechwytywanie operacji rozszerzania 529*
*15.1.4. Przechwytywanie dziedziczenia za pomocą metody Class#inherited 531*
*15.1.5. Metoda Module#const_missing 532*
*15.1.6. Metody method_added i singleton_method_added 533*
15.2. Interpretowanie zapytań dotyczących możliwości obiektów 535
*15.2.1. Wyświetlanie metod nieprywatnych obiektu 535*
*15.2.2. Wyświetlanie listy metod prywatnych i chronionych 537*

*15.2.3. Uzyskiwanie metod instancji klas i modułów 539*
*15.2.4. Wyświetlanie listy metod pojedynczego obiektu danego obiektu 541*
**15.3. Introspekcja zmiennych i stałych 543**
*15.3.1. Wyświetlanie listy zmiennych lokalnych lub globalnych 543*
*15.3.2. Wyświetlanie listy zmiennych instancji 543*
**15.4. Śledzenie wykonywania kodu 544**
*15.4.1. Sprawdzanie danych śledzenia stosu za pomocą metody caller 544*
*15.4.2. Tworzenie narzędzia do analizowania danych śledzenia stosu 546*
**15.5. Wywołania zwrotne i inspekcja metod w praktyce 549**
*15.5.1. Fundament środowiska MicroTest: MiniTest 549*
*15.5.2. Określanie i implementowanie narzędzia MicroTest 551*
**15.6. Podsumowanie 554**

***Skorowidz 557***

# *Przedmowa*

Witamy w drugim wydaniu książki. W ciągu pięciu lat, jakie upłynęły od wprowadzenia do sprzedaży pierwszego wydania, język Ruby zmienił się i rozwinął, co najbardziej widoczne jest w przypadku wersji 2.0, która pojawiła się na dwudzieste „urodziny" języka Ruby (24 lutego 2013 r.). Pod koniec roku 2013, a dokładniej rzecz biorąc, w okresie świąt bożonarodzeniowych, udostępniono wersję 2.1 języka Ruby. Tę wersję omówiono w tym wydaniu książki.

Pierwsze wydanie książki spotkało się z bardzo dobrym przyjęciem. Najwyraźniej książka idealnie przypadła do gustu wielu czytelnikom, zarówno tym, którzy dopiero zaczynali przygodę z językiem Ruby, jak i wielu osobom, które już opanowały go w pewnym stopniu, ale zamierzały dokładnie poznać język, aby mieć pewność, że faktycznie jest dla nich zrozumiały. Szczególną przyjemność sprawiły mi te komentarze, w których czytelnicy opisali swoje wrażenia związane z lekturą książki porównywalne do bezpośredniej pracy z nauczycielem obsługi języka Ruby. W końcu też *jestem* takim nauczycielem. Choć uczenie jest czymś odmiennym od pisania książki, okazuje się, że te dwie czynności nie są aż tak różne.

Sporo rzeczy nie uległo zmianie w języku Ruby, ale dość wiele też się zmieniło. W drugim wydaniu książki zawarto mnóstwo nowego materiału, a ponadto w znacznym stopniu przeanalizowano i przebudowano treść oryginalnego wydania. To wydanie książki, podobnie jak pierwsze, nie ma być wyczerpującym omówieniem języka Ruby i standardowej biblioteki (można je znaleźć w innych publikacjach), lecz dogłębnym objaśnieniem kluczowych aspektów modelu obiektowego, klas wbudowanych i ważnych narzędzi programistycznych języka Ruby.

Pragnę gorąco powitać wszystkich czytelników książki, zarówno tych, którzy przeczytali pierwsze wydanie, jak i osoby, które po raz pierwszy ją nabyły. Mam nadzieję i ufam w to, że znajdziesz w książce wiele interesujących i ekscytujących informacji. Miłej lektury!

# *Przedmowa do pierwszego wydania*

W 2006 r. wydawnictwo Manning wprowadziło do sprzedaży moją książkę zatytułowaną *Ruby for Rails: Ruby Techniques for Rails Developers*. Książkę poświęconą środowisku *Ruby for Rails* (skrót *R4R*, pod jakim stało się ono znane) napisałem, aby umożliwić projektantom aplikacji używającym tego środowiska zdanie sobie sprawy z tego, że bycie takim projektantem oznacza bycie projektantem korzystającym z języka Ruby, a ponadto chciałem przekazać solidną wiedzę dotyczącą tego języka. Wybierając do omówienia zagadnienia związane z językiem Ruby, bazowałem na własnej opinii co do ich relatywnego znaczenia dla osób, które chciały nauczyć się języka Ruby głównie w celu poprawnego i efektywnego korzystania ze środowiska Rails.

Reakcja krytyków na moją książkę okazała się bardzo dobra. Publikacja wypełniła próżnię: nie była ani książką poświęconą tylko językowi Ruby, ani książką dotyczącą wyłącznie środowiska Rails, lecz pozycją traktującą o języku Ruby „zoptymalizowaną", jak można to ująć, pod kątem projektanta aplikacji używającego środowiska Rails. Przyjemnością było dla mnie to, jak książka została odebrana, a zwłaszcza opinie wielu osób, które po przeczytaniu jej pytały mnie, czy kiedykolwiek planowałem napisać książkę w całości poświęconą językowi Ruby. Osoby te zachęcały mnie do tego.

I właśnie tak, mówiąc w skrócie, postąpiłem.

Książka *Ruby. Przewodnik programisty* dotyczy wyłącznie języka Ruby. Została napisana z myślą o każdym, kto jest zainteresowany tym językiem. Wywodzi się ona z książki o środowisku *R4R*, ale nie do końca jest jej uaktualnieniem. Bardziej służy innym celom. Choć w pewnym stopniu pokrywa się z tą książką, zawiera też wiele nowego materiału (tak naprawdę, więcej niż początkowo przewidywałem). Całość treści książki, niezależnie od tego, czy powtórzona, czy nie, została dopracowana i zaktualizowana pod kątem użycia wersji 1.9.1 języka Ruby, czyli najnowszej wersji w chwili, gdy książka trafiła do druku.

Podkreślam, że nie będę mieć za złe, jeśli projektanci korzystający ze środowiska Rails *nie* przeczytają tej książki. Jestem natomiast pełen optymizmu z tego powodu, że w ciągu trzech lat od momentu wydania książki o środowisku R4R powszechna stała się idea, że projektanci używający tego środowiska powinni poznawać język Ruby. Ponadto wiele osób, które po raz pierwszy dowiedziały się o języku Ruby, korzystając ze środowiska Rails, niezależnie zainteresowało się tym językiem. Pragnę, aby ta książka czekała na te osoby, a także na wiele osób, które odkrywają język Ruby za pośrednictwem zupełnie innych kanałów.

A zatem niezależnie od tego, co sprawiło, że sięgnąłeś po tę książkę, mam nadzieję, że jej lektura okaże się przyjemnością.

# *Podziękowania*

Przede wszystkim dziękuję wszystkim, którym dziękowałem przy okazji pierwszego wydania. Niniejsze wydanie nie powstałoby, gdyby nie powstał zespół złożony z redaktorów, personelu odpowiedzialnego za produkcję, recenzentów, wydawców i przyjaciół.

W przypadku drugiego wydania podziękowania kieruję przede wszystkim do Dana Maharry, redaktora projektu, którego cierpliwość, wsparcie i wskazówki pozwalały zapewnić kontynuowanie prac nad książką, co w przeciwnym razie mogło nie mieć miejsca. Andrew Gibson, redaktor techniczny projektu, czuwał nad całością prac związanych z książką, oferując swoją fachowość, dzięki czemu książka wiele zyskała. Jodie Allen, korektorka, w dużym stopniu przyczyniła się do nadania tekstowi ostatecznej formy. Deepak Vohra, korektor techniczny, skrupulatnie sprawdził przykładowy kod i w odpowiednim czasie wychwycił kilka problemów. Melody Dolab, korektorka, zajęła się końcowym czytaniem całego tekstu i korektą.

Na etapie poprzedzającym produkcję Kevin Sullivan dbał o to, aby wszystko przebiegało zgodnie z planem. Mary Piergies, menedżer ds. produkcji z ogromnym doświadczeniem, ponownie pokierowała pracami finalizującymi książkę. Dennis Dalinnik, odpowiedzialny za skład, świetnie poradził sobie z zapewnieniem książce odpowiedniego wyglądu. Na etapie kontroli jakości Katie Tennant przekazała wiele cennych uwag i sugestii.

W trakcie prac nad książką swoimi komentarzami i krytyką podzieliło się kilku zewnętrznych recenzentów, którzy w znacznym stopniu pomogli mi w sprawieniu, że w porównaniu z pierwszym wydaniem drugie jest jeszcze bardziej dopracowane i przystępne. Dziękuję następującym osobom: Jasonowi Brown, Mariusowi Butuc, Hectorowi Sansores, Jeffreyowi Schwab, Dougowi Sparling, Andrei Tarocchi, Tedowi Roche i Williamowi E. Wheeler.

Podczas tworzenia drugiego wydania David Williams zapewniał mi wsparcie i motywację. Jestem mu bardzo wdzięczny za dbanie o to, abym kontynuował pracę, a także za utrzymywanie mojego morale na wysokim poziomie.

Jak zawsze podziękowania dla Yukihiro „Matz" Matsumoto za stworzenie wspaniałego języka Ruby. Nawiasem mówiąc, niedawno język ten obchodził swoje dwudzieste pierwsze urodziny (24 lutego 2014 r.), licząc od chwili pierwszego publicznego poinformowania o projekcie przez Matza. Jest mi szczególnie miło, że pojawienie się drugiego wydania zbiega się w czasie, przynajmniej w przybliżeniu, z tym wspaniałym kamieniem milowym.

# O książce

## Witamy

... w drugim wydaniu książki.

Ruby to uniwersalny, interpretowany język programowania zorientowany obiektowo, który został stworzony przez Yukihiro „Matz" Matsumoto. Po raz pierwszy informacje o języku ogłoszono w 1993 r. Pierwsza publiczna wersja pojawiła się w 1995 r. Język stał się bardzo popularny w Japonii w latach 90. Język znany jest i ceniony za swoją wyrazistość (zdolność do realizowania mnóstwa rzeczy przy użyciu stosunkowo krótkiego kodu) oraz elegancję i „wizualną płynność" jego składni i stylu. Język Ruby okazał się przydatny i efektywny w wielu różnych zastosowaniach programistycznych, począwszy od tworzenia skryptów administracyjnych i urządzeń wbudowanych, a skończywszy na projektowaniu aplikacji internetowych i przetwarzaniu dokumentów PDF. Co więcej, można zaryzykować zabrzmienie w sposób nietechniczny, stwierdzając, że programowanie w języku Ruby jest zabawne. W taki właśnie sposób został zaprojektowany. Jak powiedział Matz, język Ruby zoptymalizowano pod kątem komfortu pracy programisty. Tak naprawdę język Ruby początkowo był ulubionym projektem Matza, który zdobył popularność, ponieważ bardzo wielu innych programistów z przyjemnością używało projektu tego samego rodzaju języka, jaki stworzył Matz.

Pierwsza anglojęzyczna książka o języku Ruby (*Programming Ruby* autorstwa Dave'a Thomasa i Andy'ego Hunta; wydawnictwo Addison-Wesley) pojawiła się pod koniec roku 2000 i zapoczątkowała poza Japonią falę entuzjazmu związanego z językiem Ruby. Popularność języka na Zachodzie ciągle się zwiększała od momentu pojawienia się „książki z kilofem" (potoczne określenie publikacji autorstwa Thomasa i Hunta, które wywodzi się od ilustracji książki). Zaprezentowanie przez Davida Heinemeiera Hanssona cztery lata po pojawieniu się pierwszego wydania tej książki środowiska projektowania aplikacji internetowych Ruby on Rails spowodowało ogromny wzrost zainteresowania językiem Ruby na całym świecie. Począwszy od 2004 r., w kolejnych latach miał miejsce wzrost w stopniu wykładniczym wykorzystania języka Ruby, a także liczby wydawanych książek o nim, grup użytkowników języka Ruby oraz związanych z nim konferencji i innych wydarzeń.

Jestem projektantem używającym środowiska Rails i jego miłośnikiem. Jednocześnie jestem mocno przekonany, że jeśli nawet środowisko to nigdy by się nie pojawiło, ostatecznie świat „odkryłby" język Ruby w skali, jaką obserwujemy w „erze" środowiska Rails. Ruby to zbyt ujmujący i wszechstronny język, aby na zawsze pozostał nie do końca odkrytym klejnotem. Języka Ruby używam od prawie 14 lat. Z przyjemnością

prezentowałem go dużej liczbie osób za pośrednictwem moich książek i prowadzonych zajęć. Satysfakcję sprawia mi obserwowanie, jak zdecydowana większość tych osób zaakceptowała język Ruby z przyjemnością i zadowoleniem.

Właśnie z tego powodu napisałem tę książkę. Jej celem jest umożliwienie obszernego i dogłębnego zrozumienia zasad funkcjonowania języka Ruby, a także znacznego zestawu narzędziowego złożonego z technik i idiomów języka, których możesz użyć do prawdziwego programowania.

## Struktura książki

Drugie wydanie książki składa się z 15 rozdziałów podzielonych na następujące trzy części:

- Część I „Fundamenty języka Ruby".
- Część II „Wbudowane klasy i moduły".
- Część III „Dynamika języka Ruby".

Część I (rozdziały 1. – 6.) stanowi wprowadzenie do składni języka Ruby oraz kilku kluczowych zagadnień i semantyki, na których bazuje programowanie z wykorzystaniem tego języka. Są to: obiekty, metody, klasy i moduły, identyfikatory oraz inne elementy. Omówiono również cykl życia związany z programowaniem z użyciem języka Ruby (sposób przygotowywania i wykonywania plików kodu, tworzenie programów obejmujących więcej niż jeden plik), a także wiele narzędzi wiersza poleceń dołączonych do języka Ruby, które często są stosowane przez programistów, w tym interaktywny interpreter języka Ruby (irb), menedżer pakietów RubyGems (gem) i interpreter języka Ruby (ruby).

W części II (rozdziały 7. – 12.) dokonano przeglądu głównych klas wbudowanych, w tym łańcuchów, tablic, tablic asocjacyjnych, liczb, zakresów, dat i czasu, a także wyrażeń regularnych. Ponadto wyjaśniono przeznaczenie różnych klas wbudowanych i zaprezentowano podstawy korzystania z nich. W tej części poszerzana jest też ogólna znajomość języka Ruby przez omawianie takich zagadnień, jak logika boolowska języka Ruby, metody wbudowane służące do przekształcania klas obiektów (na przykład łańcucha w liczbę całkowitą), znaczące ułatwienia przeznaczone do tworzenia kolekcji i ich wyliczenia oraz techniki porównywania obiektów pod kątem tożsamości i równości. Dowiesz się też o operacjach wejścia-wyjścia związanych z plikami i konsolą, a także o wykonywaniu poleceń systemowych w obrębie programów Ruby.

Część III (rozdziały 13. – 15.) dotyczy obszaru związanego z dynamiką języka Ruby. W ramach tego zagadnienia będzie mowa o kilku tematach podrzędnych. Wśród nich znajdzie się kilka technik metaprogramowania, w tym narzędzia języka Ruby oferujące mechanizm refleksji w czasie wykonywania kodu i introspekcję obiektów, sposoby zapewniania obiektom zindywidualizowanych zachowań oraz obsługę funkcji, wątków i innych obiektów umożliwiających wywoływanie i uruchamianie. W tej części książki zaprezentowano też techniki umożliwiające wywoływanie poleceń systemowych w obrębie programu Ruby, a ponadto uwzględniono kilka oferowanych przez język Ruby procedur i wywołań zwrotnych środowiska wykonawczego wyzwalanych przez zdarzenia.

Są to na przykład procedury obsługi wywołań nieistniejących metod oraz przechwytywanie zdarzeń, takich jak dziedziczenie klas i definiowanie metody.

Język Ruby to system, a prezentowanie dowolnego systemu w ściśle „liniowy" sposób stanowi wyzwanie. Stawiam mu czoła, traktując proces uczenia jak coś w rodzaju rozszerzającej się spirali. Bazuję na tym, co znajome, lecz zawsze otwieram się na nieznane. Będzie się zdarzać, że tymczasowo zostanie przybliżony temat omawiany w dalszej części książki w stopniu wystarczającym do tego, aby można było dogłębnie zaznajomić się z bieżącym tematem. Gdy już się go opanuje w wymaganym stopniu, można powrócić do tematu tymczasowego i przestudiować go niezależnie. Drugie wydanie książki tak skonstruowano, aby prezentowało jak największą możliwą ilość materiału w jak najbardziej efektywny sposób, który jest spójny z misją książki, czyli zapewnieniem solidnych fundamentów z zakresu języka Ruby w postaci rzeczywistego i trwałego zrozumienia sposobu funkcjonowania języka.

## *Dla kogo przeznaczona jest książka?*

Treść drugiego wydania książki zoptymalizowano pod kątem czytelnika, który ma już pewne doświadczenie programistyczne, a być może nawet związane z językiem Ruby. Czytelnik ten oczekuje poszerzenia wiedzy o języku Ruby, nie tylko konkretnych technik (choć w książce omówiono ich mnóstwo), ale też zasad projektowych, które sprawiają, że język ten jest taki, a nie inny. Szczerze wierzę w świadome działania czytelnika. Jestem również przekonany, że nie oznacza to konieczności intensywnego myślenia każdorazowo przy okazji wpisywania wiersza kodu. Rozumiem przez to, że wiesz, jak najlepiej wykorzystać możliwości języka, a ponadto jak analizować problemy po ich wystąpieniu.

W odniesieniu do docelowej grupy czytelników asekurowałem się trochę w tym względzie, że dołączyłem kilka uwag wprowadzających w przypadku kilku tematów i technik, z którymi prawdopodobnie zaznajomieni są doświadczeni programiści. Proszę tych czytelników o pobłażliwość. Wspomnianych uwag nie będzie dużo. Wierzę w to, że nawet kilka słów wyjaśniających pojęcia, które pojawią się sporadycznie, może w przypadku wielu osób w zaskakująco dużym stopniu wpłynąć na ich pozytywny odbiór książki. Jeśli jesteś bardziej doświadczonym programistą i zauważasz fragmenty w książce, w których najwyraźniej staram się objaśniać podstawowe zagadnienia, proszę o wyrozumiałość. Jest to robione w dobrym celu.

Z tego samego powodu, jeśli po raz pierwszy masz styczność z programowaniem, bądź przygotowany na odrobienie z własnej inicjatywy dodatkowych „zadań domowych", które umożliwią rozpoczęcie realizowania działań w ramach procesu programowania. Jak najbardziej daj szansę drugiemu wydaniu książki. Choć nie jest ona wyłącznie wprowadzeniem do programowania, „przeprowadza" przez wszystkie aspekty praktyczne, w tym tworzenie i uruchamianie plików programu. Ponadto w książce język Ruby jest objaśniany od podstaw.

## Czego książka nie zawiera?

W drugim wydaniu książki dokonano poważnego i rozbudowanego omówienia języka Ruby. Nie pełni ona jednak roli materiału referencyjnego z zakresu tego języka. Istnieją klasy podstawowe, o których piszę niewiele albo nie piszę wcale. Omawiam tylko niewielką liczbę pakietów biblioteki standardowej. Takie było założenie. Nie jest konieczne, abym objaśniał sposób użycia każdego interfejsu API biblioteki standardowej. Tego też nie robię. Z dużym prawdopodobieństwem wymagasz, aby ktoś wyjaśnił dokładnie, co oznacza konstrukcja `class << self`, dlaczego dwie zmienne instancji rozdzielone dwoma wierszami nie są tą samą zmienną, czym się różnią metody pojedynczego obiektu i metody prywatne albo czym jest enumerator i pod jakim względem różni się od iteratora. Niezbędne jest uzyskanie związanych z tymi rzeczami informacji, a także poznanie ich od strony praktycznej w celu rozpoczęcia korzystania z nich. Oczywiście podczas pracy z językiem Ruby musisz bardzo dokładnie poznać bibliotekę standardową. Zachęcam do tego. Moim celem w tej książce jest przekazanie określonego rodzaju i stopnia znajomości zagadnień.

## Informacje o wersjach języka Ruby

Drugie wydanie książki dotyczy wersji 2.1 języka Ruby, czyli w momencie jej pisania najnowszej dostępnej wersji. Wersja 1.9 nadal jest powszechnie używana, choć przewiduję, że nie potrwa to zbyt długo. Jeśli nadal korzystasz z wersji 1.9, książka okaże się bardzo przydatna, a zwłaszcza wtedy, gdy są szanse na to, że w niezbyt odległej przyszłości zaczniesz używać jakiejś wersji języka Ruby 2.

## Konwencje dotyczące kodu, przykłady i dane do pobrania

W treści akapitów nazwy zmiennych i stałych języka Ruby są formatowane za pomocą stylu `czcionka`. Dla nazw klas i modułów używany jest ten styl w sytuacji, gdy są bezpośrednimi odwołaniami do istniejących obiektów klasy lub modułu. Przykładem jest zdanie: „Następnie zostanie ponownie otwarty blok definicji klasy `Person`". W każdej sytuacji będziesz w stanie na podstawie kontekstu stwierdzić fakt omawiania klasy, modułu lub innego elementu języka Ruby.

Kod źródłowy wszystkich przykładów zamieszczonych w książce jest dostępny do pobrania pod adresem *ftp://ftp.helion.pl/przyklady/rubprp.zip*.

Dla nazw programów, takich jak `ruby` i `rails`, stosowany jest styl `czcionka`. W tym przypadku nazwy odwołują się bezpośrednio do pliku wykonywalnego programu lub użycia wiersza poleceń. W przeciwnym razie nazwy te będą przedstawione przy użyciu zwykłej czcionki bez formatowania.

Kursywa lub gwiazdka są stosowane w przypadku wyrażeń z symbolami wieloznacznymi. Na przykład łańcuch `to_*` może wskazywać ogólną kategorią metod języka Ruby, która obejmuje metody `to_i` i `to_s`, natomiast łańcuch *`pozycja`*`_match` może odpowiadać nazwie metody `post_match` lub `pre_match`.

W celu uruchomienia odrębnych przykładowych kodów zawartych w książce możesz umieścić je w pliku tekstowym, a następnie wykonać dla niego polecenie `ruby`, albo

wpisać kod w narzędziu irb interaktywnego interpretera języka Ruby. W rozdziale 1. poznasz te techniki. W kolejnych rozdziałach książki będzie zakładane, że będziesz w stanie sobie poradzić z tymi działaniami, a ponadto podasz własne nazwy plików przykładów, jeśli takich nazw nie zasugerowano (lub jeśli preferujesz użycie innych nazw).

Znaczną liczbę przykładów znajdujących się w książce zaprezentowano w postaci sesji narzędzia irb. Na stronach książki zobaczysz wiersze, które za pomocą operacji wycinania i wklejania przeniesiono z otwartej sesji interaktywnej, w której kod był wprowadzany do narzędzia irb, a ono w odpowiedzi wykonywało wiersze kodu. Z łatwością rozpoznasz ten format (zwłaszcza gdy sam zaczniesz używać tego narzędzia).

Taki tryb prezentacji jest szczególnie odpowiedni w przypadku krótkich fragmentów kodu i wyrażeń. Ponieważ narzędzie irb zawsze wyświetla wyniki operacji wykonywania wszystkiego, co zostało wprowadzone (co bardziej przypomina kalkulator), umożliwia ono ujrzenie wyników przy jednoczesnym ograniczeniu liczby jawnie używanych poleceń `print`.

W innych sytuacjach dane wyjściowe przykładowych kodów są wyświetlane osobno po zaprezentowaniu przykładów, pokazywane razem z kodem (i wyraźnie oznaczane jako dane wyjściowe) lub wstawiane w treści omówienia następującego po zamieszczonym kodzie.

W niektórych przykładach z boku wierszy kodu pojawiają się dodatkowo liczby. Są one powiązane z konkretnymi punktami w następującym dalej omówieniu, a ponadto umożliwiają szybkie odnalezienie analizowanego wiersza kodu.

Wywołania programów z poziomu wiersza poleceń są identyfikowane przez znak `$`, który reprezentuje ogólny styl wiersza zachęty powłoki środowisk uniksowych. Większość tych poleceń zadziała w systemie Windows, nawet pomimo tego, że symbol zachęty może być inny (we wszystkich środowiskach, jak zawsze, dostępność poleceń jest zależna od ustawienia odpowiedniej zmiennej środowiskowej ścieżki).

# Część I

# Fundamenty języka Ruby

Celem tej części książki jest zaprezentowanie ogólnych, lecz praktycznych informacji podstawowych, na bazie których będziesz w stanie kontynuować dalsze eksploracje języka Ruby w części drugiej i trzeciej książki. Zaczniemy od rozdziału, który stanowi wprowadzenie do nauki języka Ruby. Po jego przeczytaniu będziesz miał możliwość wygodnego uruchamiania programów Ruby, a ponadto dobrą orientację w strukturze typowej instalacji języka Ruby. Jest to język obiektowy, dlatego im wcześniej zagłębisz się w to, jak są w nim obsługiwane obiekty, tym lepiej. W związku z tym obiekty będą zarówno sposobem wprowadzenia w omówienie języka (i jego znajomości), jak i „złotą nitką" prowadzącą nas do dalszych zagadnień i technik.

Obiekty są tworzone przez klasy. W rozdziale 3. dowiesz się, jak one działają. Po zaprezentowaniu klas w rozdziale 4. przyjrzymy się modułom. Moduły umożliwiają dostrajanie klas i obiektów przez wydzielanie części projektu obiektów do osobnych porcji kodu, które mogą być wielokrotnie wykorzystywane. Aby zrozumieć programy Ruby (zarówno własne, jak i innych osób), niezbędne jest zaznajomienie się z obecnym w języku Ruby pojęciem bieżącego *obiektu domyślnego* identyfikowanego przez słowo kluczowe `self`. W rozdziale 5. zostanie przybliżone pojęcie obiektu domyślnego, a ponadto omówione zostaną: obsługa w języku Ruby widoczności zmiennych oraz zasięgi.

W rozdziale 6., który jest ostatnim w tej części książki, dowiesz się o przepływie sterowania w programach Ruby, czyli o tym, jak „prowadzić" interpreter języka Ruby przez logikę instrukcji warunkowych (`if`), jak wielokrotnie wykonywać pętlę w kodzie, a nawet jak przerwać normalne wykonywanie programu w momencie wystąpienia błędu. Na końcu tego rozdziału będziesz myśleć razem z interpreterem języka Ruby podczas pisania i rozwijania kodu.

Tytuł części I to „Fundamenty języka Ruby". Oczywiście sugeruje on, że na treści tej części bazują kolejne części książki. Jest to prawda. Nie oznacza to jednak, że materiał zawarty w części pierwszej nie jest ważny sam w sobie. Jak stwierdzisz po przeczytaniu tych sześciu rozdziałów, ich treść prezentuje rzeczywiste techniki stosowane w języku Ruby, prawdziwy kod oraz informacje, których będziesz używać każdorazowo w trakcie tworzenia lub wykonywania programu Ruby. Nie są to „fundamenty" dlatego, że po zaznajomieniu się z nimi po raz pierwszy później o nich zapomnisz, ale dlatego, że z językiem Ruby związanych jest znacznie *więcej* rzeczy, które należy jeszcze opanować!

# 1 Wprowadzenie do nauki języka Ruby

**W tym rozdziale:**

- Niezbędna składnia języka Ruby.
- Objaśnienie podstawowych operacji programistycznych języka Ruby: pisanie kodu, zapisywanie, uruchamianie i sprawdzanie programów pod kątem błędów.
- Omówienie instalacji języka Ruby.
- Mechanizmy rozszerzeń języka Ruby.
- Standardowe narzędzie wiersza poleceń języka Ruby, w tym jego interaktywny interpreter (irb).

Książka zapewni fundamenty języka Ruby, a rozdział ten stanowi przygotowanie do tego. Celem rozdziału jest wprowadzenie do nauki języka Ruby przez przekazanie wiedzy i umiejętności wystarczających do dalszego poznawania języka w łatwy sposób.

Przyjrzymy się podstawowej składni języka Ruby i związanym z nim technikom, a także sposobowi jego działania. Będzie mowa o działaniach wykonywanych podczas pisania kodu programu, metodach uruchamiania programu przez interpreter języka Ruby oraz sposobie dzielenia programu na więcej niż jeden plik. Poznasz kilka przełączników zmieniających sposób działania interpretera języka Ruby (program o nazwie `ruby`, do którego kierowane są pliki programu w celu ich wykonania), a także dowiesz się, jak używać ważnych narzędzi pomocniczych mających za zadanie ułatwienie pracy programiście korzystającemu z języka Ruby i zwiększenie jej efektywności.

Treść rozdziału bazuje na ogólnym obrazie całości języka Ruby podzielonym na następujące trzy fundamentalne poziomy:

- Podstawowy język: reguły projektowe, składnia i semantyka.
- Rozszerzenia i biblioteki dołączone do języka Ruby oraz ułatwienia służące do dodawania rozszerzeń we własnym zakresie.
- Narzędzia wiersza poleceń oferowane przez język Ruby, w przypadku których uruchamiany jest interpreter oraz kilka innych ważnych narzędzi.

Nie zawsze możliwe jest osobne omawianie tych trzech poziomów, ponieważ jednak stanowią one część jednego systemu, w niniejszym rozdziale postąpimy w ten sposób w takim stopniu, w jakim jest to możliwe. W każdej sytuacji możesz jednak posłużyć się opisami tych trzech poziomów jak „wieszakami", na których będą zawieszane tematy podrzędne po ich wprowadzeniu.

**Ruby, ruby i... RUBY?!**

Ruby to język programowania. Mowa jest o takich pojęciach jak „nauka języka Ruby". Zadajemy następujące pytania: „Czy znasz język Ruby?". Nazwa `ruby` zapisana małymi literami dotyczy programu komputerowego, a dokładniej rzecz biorąc, interpretera języka Ruby, który wczytuje programy i uruchamia je. Nazwa ta będzie używana w następujących zdaniach: „Uruchomiłem program `ruby` dla mojego pliku, ale nic się nie stało" lub „Jaka jest pełna ścieżka do pliku wykonywalnego `ruby`?". I wreszcie: występuje nazwa RUBY, ale właściwie nie powinna ona istnieć. Nazwa Ruby nie jest akronimem, dlatego nigdy nie będzie poprawne zapisywanie jej za pomocą wyłącznie dużych liter. Wiele osób tak postępuje, podobnie jak (również niewłaściwie) w przypadku nazwy Perl, być może z tego powodu, że miały do czynienia z nazwami języków takimi jak BASIC i COBOL. Ruby nie jest tego rodzaju językiem. Nazwa Ruby dotyczy języka, a nazwa `ruby` odnosi się do jego interpretera.

Czy pierwszy rozdział pełni wyłącznie rolę przygotowania do dalszych rozdziałów? Treść tego rozdziału rządzi się swoim własnymi prawami: poznasz w nim rzeczywiste techniki związane z językiem Ruby oraz ważne kwestie dotyczące struktury języka. Celem jest przygotowanie Czytelnika. Jednak nawet ten proces będzie obejmować bliższe omówienie niektórych kluczowych aspektów języka Ruby.

## 1.1. Ogólne wprowadzenie do języka Ruby

Celem niniejszego podrozdziału jest zaznajomienie z językiem Ruby. Opiera się to na metodzie przeglądowej obejmującej omówienie całego cyklu poznawania wybranej składni, pisania kodu i uruchamiania programów.

Na tym etapie wymagane jest zainstalowanie języka Ruby na komputerze.[1] Przykłady zamieszczone w książce bazują na języku Ruby w wersji 2.1.0. Niezbędny jest również edytor tekstu (możesz skorzystać z dowolnego preferowanego edytora, jeśli jest on zwykłym edytorem tekstu, a nie procesorem tekstu) oraz katalog (inaczej folder), w którym będą zapisywane pliki programów Ruby. Katalog ten może mieć nazwę

[1] Kompletne i aktualne instrukcje instalacji języka Ruby dostępne są pod adresem *http://ruby-lang.org*.

*kod_ruby* lub *przykłady_ruby*. Dowolna nazwa jest poprawna, pod warunkiem że odróżnia ona katalog od innych katalogów roboczych, aby umożliwić śledzenie plików programów służących do ćwiczeń.

**Interaktywny program konsoli języka Ruby (irb) — Twój nowy, najlepszy przyjaciel**

Narzędzie irb wchodzi w skład języka Ruby. Jest to najpowszechniej używane narzędzie wiersza poleceń języka inne niż sam interpreter. Po uruchomieniu tego narzędzia wprowadzasz w nim kod Ruby, który następnie jest wykonywany. Na końcu wyświetlana jest wartość wynikowa.

W wierszu poleceń wpisz polecenie `irb` i wprowadź przykładowy kod znaleziony w treści książki. Oto przykład:

```
>> 100 + 32
=> 132
```

Otwarcie sesji narzędzia irb oznacza, że możesz testować fragment kodu Ruby w dowolnym momencie i ilości. Większość projektantów programów Ruby uważa to narzędzie za niezastąpione. W dalszej części rozdziału zamieszczono kilka przykładów jego użycia.

W przykładach zastosowania narzędzia irb, które zawarto w książce, używana będzie opcja `--simple-prompt` ułatwiająca odczytanie danych wyjściowych narzędzia.

```
irb --simple-prompt
```

Aby sprawdzić efekt działania tej opcji, spróbuj uruchomić narzędzie irb z tą opcją i bez niej. Jak się okaże, opcja powoduje, że zawartość ekranu jest znacznie bardziej przejrzysta. W domyślnym wariancie (bez tej opcji) narzędzie irb wyświetla więcej informacji, takich jak licznik wierszy sesji interaktywnej. Jednak w przypadku analizowanych przykładów opcja `--simple-prompt` jest wystarczająca.

Ponieważ irb to jedno z narzędzi wiersza poleceń dołączonych do języka Ruby, szczegółowo zostanie omówione dopiero w punkcie 1.4.2. Możesz już teraz przejść do niego i zaznajomić się z jego treścią. Jest to naprawdę proste zadanie.

Po zainstalowaniu języka Ruby i przygotowaniu środowiska roboczego możesz kontynuować wprowadzenie do języka Ruby, aby zapewnić sobie podstawy przed dalszym tworzeniem programów i eksplorowaniem języka. Niezbędne będzie wcześniejsze poznanie w wystarczającym stopniu podstawowej składni języka Ruby.

### 1.1.1. Niezbędna składnia języka Ruby

W zamieszczonych dalej trzech tabelach podsumowano wybrane funkcje języka Ruby, które pomogą zrozumieć przykłady zawarte w rozdziale, a także rozpocząć eksperymentowanie za pomocą języka Ruby. Nie musisz zapamiętywać funkcji, ale przejrzyj je i w razie potrzeby wracaj do nich później.

W tabeli 1.1 wyszczególniono niektóre podstawowe operacje języka Ruby. W tabeli 1.2 omówiono operacje pobierania podstawowych danych wprowadzonych za pomocą klawiatury, wysyłanie danych wyjściowych na ekran oraz najprostsze instrukcje warunkowe. W tabeli 1.3 w skrócie opisano obiekty specjalne języka Ruby i składnię powiązaną z komentarzami.

**Tabela 1.1.** Podstawowe operacje języka Ruby

| Operacja | Przykłady | Komentarze |
|---|---|---|
| Arytmetyczna | `2 + 3` (dodawanie)<br>`2 - 3` (odejmowanie)<br>`2 * 3` (mnożenie)<br>`2 / 3` (dzielenie)<br>`10.3 + 20.25`<br>`103 - 202.5`<br>`32.9 * 10`<br>`100.0 / 0.23` | Wszystkie te operacje mogą być wykonywane na liczbach całkowitych i zmiennopozycyjnych. Łączenie ze sobą liczb całkowitych i zmiennopozycyjnych, tak jak ma to miejsce w niektórych przykładach, powoduje uzyskanie wyniku w postaci liczby zmiennopozycyjnej.<br>Zauważ, że zamiast `.23` musisz wpisać `0.23`. |
| Przypisanie | `x = 1`<br>`string = "Witaj"` | Operacja wiąże zmienną lokalną (po lewej stronie) z obiektem (po prawej stronie). Na razie obiekt możesz traktować jako wartość reprezentowaną przez zmienną. |
| Porównywanie dwóch wartości | `x == y` | Zauważ, że występują dwa znaki równości, a nie jeden jak w przypisaniu. |
| Przekształcanie łańcucha liczbowego w liczbę | `x = "100".to_i`<br>`s = "100"`<br>`x = s.to_i` | Aby wykonać operację arytmetyczną, musisz upewnić się, że dostępne są liczby, a nie łańcuchy znakowe. Funkcja `to_i` dokonuje przekształcenia łańcucha w liczbę całkowitą. |

**Tabela 1.2.** Podstawowe metody danych wejściowych i wyjściowych oraz kontrola przepływu w języku Ruby

| Operacja | Przykłady | Komentarze |
|---|---|---|
| Wyświetlanie danych na ekranie | `print "Witaj"`<br>`puts "Witaj"`<br>`x = "Witaj"`<br>`puts x`<br>`x = "Witaj"`<br>`print x`<br>`x = "Witaj"`<br>`p x` | Metoda `puts` dodaje znak nowego wiersza do zwracanego łańcucha, jeśli na jego końcu nie ma jeszcze tego znaku. Metoda `print` nie dodaje takiego znaku. Metoda ta wyświetla dokładnie takie dane, jakie jej kazano, a następnie umieszcza kursor na ich końcu (uwaga: w przypadku niektórych platform dodatkowy wiersz jest automatycznie generowany na końcu działania programu). Metoda `p` zwraca łańcuch inspekcji, który może zawierać dodatkowe informacje o wyświetlanych danych. |
| Pobieranie wiersza danych wprowadzonych za pomocą klawiatury | `gets`<br>`string = gets` | Wiersz z wprowadzonymi danymi możesz przypisać bezpośrednio do zmiennej (zmienna `string` w drugim przykładzie). |
| Wykonywanie warunkowe | `if x == y`<br>`  puts "Tak!"`<br>`else`<br>`  puts "Nie!"`<br>`end` | Instrukcje warunkowe zawsze są zakończone słowem `end`. Więcej informacji na ich temat zamieszczono w rozdziale 6. |

Kilka zasadniczych aspektów języka Ruby i jego składni jest zbyt złożonych, aby zestawić je w tabeli. Konieczne będzie rozpoznawanie grupy różnych identyfikatorów języka Ruby, a przede wszystkim musisz zrozumieć, czym jest obiekt w tym języku, a także jak wygląda wywołanie metody. W dalszej części rozdziału przyjrzymy się obu aspektom języka.

**Tabela 1.3.** Obiekty specjalne i komentarze w języku Ruby

| Operacja | Przykłady | Komentarze |
|---|---|---|
| Obiekty specjalne jako wartości | `true`<br>`false`<br>`nil` | Obiekty `true` i `false` często pełnią rolę wartości zwracanych w przypadku wyrażeń warunkowych. Obiekt `nil` to swego rodzaju „nieobiekt" wskazujący na brak wartości lub wyniku. Obiekty `false` i `nil` powodują niepowodzenie wyrażenia warunkowego. Wszystkie inne obiekty (w tym oczywiście obiekt `true`, ale też `0` i łańcuchy puste) zapewniają pomyślne zakończenie takich wyrażeń. Więcej informacji na ten temat zamieszczono w rozdziale 7. |
| Obiekt domyślny | `self` | Słowo kluczowe `self` odnosi się do obiektu domyślnego. Słowo to określa rolę, jaką pełnią różne obiekty, zależnie od kontekstu wykonywania. Wywołania metod, które nie wyszczególniają obiektu wywołującego, są stosowane dla obiektu `self`. Więcej informacji na ten temat zamieszczono w rozdziale 5. |
| Wstawianie komentarzy w plikach kodu | `# Komentarz A`<br>`x = 1 # Komentarz A` | Komentarze są ignorowane przez interpreter. |

### 1.1.2. Różnorodność identyfikatorów języka Ruby

Język Ruby zawiera niewielką liczbę typów identyfikatorów, jakie trzeba będzie od razu rozpoznać i odróżniać od siebie. Oto struktura drzewa rodziny identyfikatorów:

- Zmienne:
  - lokalne,
  - instancji,
  - klas,
  - globalne.
- Stałe.
- Słowa kluczowe.
- Nazwy metod.

Jest to niewielka grupa, którą z łatwością można opanować. W dalszej części rozdziału dokonamy ich przeglądu. Pamiętaj o tym, że celem lektury tego punktu jest nabycie umiejętności rozpoznawania różnych identyfikatorów. W różnych miejscach książki dowiesz się również znacznie więcej o tym, jak i kiedy z nich korzystać. Na razie jest to tylko pierwsza lekcja z zakresu identyfikatorów.

#### ZMIENNE

Nazwy *zmiennych lokalnych* rozpoczynają się od małej litery lub znaku podkreślenia, a ponadto składają się z liter, znaków podkreślenia i/lub cyfr. `x`, `string`, `abc`, `start_value` i `firstName` to poprawne nazwy zmiennych lokalnych. Zauważ jednak, że w przypadku tworzenia nazw zmiennych lokalnych przy użyciu wielu słów konwencja w języku Ruby określa stosowanie znaków podkreślenia zamiast liter o różnej wielkości (na przykład użycie nazwy `first_name` zamiast nazwy `firstName`).

Nazwy *zmiennych instancji*, które służą do przechowywania informacji na potrzeby poszczególnych obiektów, zawsze rozpoczynają się od pojedynczego znaku `@`, po którym występuje taki sam zestaw znaków co w przypadku zmiennych lokalnych (na przykład `@age` i `@last_name`). Choć nazwa zmiennej lokalnej nie może zaczynać się od dużej litery, w nazwie zmiennej instancji może ona występować na pierwszej pozycji po znaku `@` (jednak na tej pozycji nie można użyć cyfry). Znakiem stosowanym po znaku `@` jest zwykle mała litera.

Nazw *zmiennych klas* przechowujących informacje dla poszczególnych hierarchii klas (i w tym przypadku nie przejmuj się na razie semantyką) dotyczą te same reguły co nazw zmiennych instancji, z tą różnicą, że nazwy rozpoczynają się od dwóch znaków `@` (na przykład `@@running_total`).

Nazwy *zmiennych globalnych* są rozpoznawane za pomocą umieszczonego na ich początku znaku `$` (na przykład `$population`). Segmentu następującego po tym znaku nie dotyczą konwencje obowiązujące przy określaniu nazw zmiennych lokalnych. Zmienne globalne mogą mieć nazwy `$:`, `$1` i `$/`, a także `$stdin` i `$LOAD_PATH`. Dopóki na początku nazwy jest znak `$`, jest to nazwa zmiennej globalnej. Z identyfikatorami zawierającymi w nazwie wyłącznie inne znaki niż alfanumeryczne spotkasz się prawdopodobnie wyłącznie w postaci predefiniowanych nazw, dlatego nie ma potrzeby martwienia się tym, jakie znaki interpunkcji są poprawne, a jakie nie.

W tabeli 1.4 podsumowano reguły nadawania nazw zmiennym w języku Ruby.

**Tabela 1.4.** Poprawne nazwy zmiennych w języku Ruby według ich typu

| Typ | Konwencja nazewnicza języka Ruby | Nazwy niezgodne z konwencją |
|---|---|---|
| Lokalne | `first_name` | `firstName`, `_firstName`, `__firstName`, `name1` |
| Instancji | `@first_name` | `@First_name`, `@firstName`, `@name1` |
| Klas | `@@first_name` | `@@First_name`, `@@firstName`, `@@name1` |
| Globalne | `$FIRST_NAME` | `$first_name`, `$firstName`, `$name1` |

### STAŁE

Nazwy stałych rozpoczynają się od dużej litery. `A`, `String`, `FirstName` i `STDIN` to poprawne nazwy stałych. W przypadku tworzenia nazw stałych za pomocą wielu słów konwencja nazewnicza języka Ruby określa użycie różnej wielkości liter (np. `FirstName`) lub znaku podkreślenia, który rozdziela słowa złożone wyłącznie z dużych liter (np. `FIRST_NAME`).

### SŁOWA KLUCZOWE

W języku Ruby występuje wiele słów kluczowych: predefiniowane oraz zastrzeżone terminy powiązane ze specyficznymi zadaniami i kontekstami programistycznymi. Słowa kluczowe obejmują słowa `def` (na potrzeby definicji metod), `class` (do definiowania klas), `if` (wykonywanie warunkowe) i `__FILE__` (nazwa aktualnie wykonywanego pliku). Istnieje około 40 słów kluczowych. Zwykle są one krótkimi identyfikatorami złożonymi z jednego wyrazu (w przeciwieństwie do tworzonych z wykorzystaniem znaku podkreślenia).

**NAZWY METOD**

Nazw metod w języku Ruby dotyczą te same reguły i konwencje co zmiennych lokalnych (z tym wyjątkiem, że nazwy metod mogą być zakończone znakami ?, ! lub =, których znaczenie zostanie później przedstawione). Z definicji metody nie zwracają na siebie uwagi jako takie, lecz po prostu łączą się ze strukturą programu w postaci wyrażeń, które zapewniają wartość. W przypadku niektórych kontekstów samo przyjrzenie się wyrażeniu nie pozwala stwierdzić, czy masz do czynienia z nazwą zmiennej lokalnej, czy metody. Jest to zamierzone.

Skoro mowa o metodach, to po ogólnym zaznajomieniu się z identyfikatorami języka Ruby powróćmy do jego semantyki, a w szczególności do bardzo ważnej roli obiektu i jego metod.

### 1.1.3. Wywołania metod, komunikaty i obiekty języka Ruby

W języku Ruby wszystkie struktury danych i wartości, począwszy od prostych wartości skalarnych (niepodzielnych), takich jak liczby całkowite i łańcuchy, a skończywszy na złożonych strukturach danych (np. tablice), są traktowane jako *obiekty*. Każdy obiekt ma możliwość rozpoznania określonego zbioru *komunikatów*. Każdy komunikat zrozumiały dla obiektu odpowiada bezpośrednio *metodzie*, czyli nazwanej procedurze wykonywalnej, której wykonanie może być wyzwolone przez obiekt.

Obiekty są reprezentowane przez konstruktory literału (np. znaki cudzysłowu w przypadku łańcuchów) lub zmienne, z którymi zostały powiązane. Wysyłanie komunikatu jest realizowane za pośrednictwem specjalnego operatora kropki: komunikat znajdujący się po jej prawej stronie jest wysyłany do obiektu po lewej stronie kropki (dostępne są inne, bardziej specjalistyczne sposoby wysyłania komunikatów do obiektów, ale znak kropki to najczęstszy i najbardziej podstawowy z nich). Przeanalizuj następujący przykład z tabeli 1.1:

```
x = "100".to_i
```

Kropka oznacza, że komunikat `to_i` jest wysyłany do łańcucha `"100"`, który jest wywoływany przez *odbiorcę* komunikatu. Możliwe jest również stwierdzenie, że metoda `to_i` jest *wywoływana* w łańcuchu `"100"`. Wynik wywołania metody, czyli liczba całkowita 100, pełni rolę prawej strony przypisania do zmiennej `x`.

**Skąd się wzięła podwójna terminologia?**

Dlaczego komplikuje się wszystko, używając zarówno określenia „wysyłanie komunikatu `to_i`", jak i „wywoływanie metody `to_i`"? Z jakiego powodu na dwa sposoby opisywana jest ta sama operacja? Wynika to stąd, że nie do końca operacje są identyczne. Przeważnie komunikat jest wysyłany do obiektu odbierającego, który wykonuje odpowiednią metodę. Jednak czasami nie istnieje odpowiednia metoda. Po prawej stronie kropki możesz umieścić cokolwiek — i nie ma gwarancji, że odbiorca będzie zawierać metodę pasującą do wysyłanego komunikatu.

Jeśli wygląda to na chaos, tak nie jest, ponieważ obiekty mogą przechwytywać nieznane komunikaty i podejmować próby nadania im znaczenia. Na przykład środowisko do projektowania aplikacji internetowych Ruby on Rails intensywnie korzysta z techniki polegającej na wysyłaniu do obiektów nieznanych komunikatów, przechwytywaniu ich i dynamicznemu nadawaniu im znaczenia na podstawie takich warunków dynamicznych jak nazwy kolumn tabel bieżącej bazy danych.

Metody mogą pobierać *argumenty*, które także są obiektami (prawie wszystko w języku Ruby ma postać obiektu, choć niektóre struktury syntaktyczne, które ułatwiają tworzenie i modyfikowanie obiektów, same nie są nimi). Oto wywołanie metody z argumentem:

```
x = "100".to_i(9)
```

Wywołanie metody `to_i` w obiekcie `100` z argumentem `9` generuje dziesiętną liczbę całkowitą równoznaczną liczbie 100 o podstawie 9: zmienna `x` jest równa wartości dziesiętnej 81.

Powyższy przykład prezentuje również użycie nawiasów okrągłych dla argumentów metody. Nawiasy te są zwykle opcjonalne, ale w bardziej złożonych przypadkach mogą być niezbędne do zapewnienia przejrzystości tego, co w przeciwnym razie mogłoby być niejednoznaczne w składni. Tak po prostu dla pewności wielu programistów korzysta z nawiasów okrągłych w większości lub we wszystkich wywołaniach metod.

Cała zawartość programu Ruby to obiekty i wysyłane do nich komunikaty. Jako programista używający języka Ruby większość czasu spędzisz na określaniu działań, jakie mogą zostać zrealizowane przez obiekty (przez definiowanie metod), lub żądaniu od nich wykonania tych działań (przez wysyłanie im komunikatów).

Wszystko to zostanie znacznie obszerniej omówione w dalszej części książki. I tym razem ten krótki przegląd stanowi jedynie część procesu wprowadzania do nauki języka Ruby. Gdy ujrzysz kropkę w miejscu, które w innym razie byłoby czymś niewytłumaczalnym, należy interpretować to jako komunikat (po prawej stronie kropki) wysyłany do obiektu (po lewej stronie kropki). Pamiętaj też o tym, że niektóre wywołania metody przyjmują formę *uproszczonych* wywołań, takich jak wywołanie komunikatu `puts` w następującym przykładzie:

```
puts "Witaj."
```

Pomimo braku w tym przypadku kropki oznaczającej wysyłanie komunikatu oraz jego jawnego odbiorcy ma miejsce wysyłanie do obiektu komunikatu `puts` z argumentem `"Witaj."`. Obiektem tym jest obiekt domyślny `self`. W czasie działania programu zawsze zdefiniowany jest obiekt `self`, choć to, jaki obiekt jest tym obiektem, zmienia się zgodnie z określonymi regułami. Znacznie więcej informacji o obiekcie `self` zamieszczono w rozdziale 5. Na razie miej świadomość tego, że uproszczone zapisy, takie jak `puts`, mogą oznaczać wywołanie metody.

W języku Ruby najważniejszym pojęciem jest obiekt. Blisko z nim powiązane i odgrywające istotną dodatkową rolę jest pojęcie *klasy*.

### POCHODZENIE OBIEKTÓW W KLASACH

*Klasy* definiują klastry zachowania lub funkcjonalności, a każdy obiekt jest instancją dokładnie jednej klasy. Język Ruby zapewnia dużą liczbę klas wbudowanych, które reprezentują ważne podstawowe typy danych (są to na przykład klasy `String`, `Array` i `Fixnum`). Każdorazowo podczas tworzenia obiektu łańcuchowego tworzona jest instancja klasy `String`.

Możliwe jest też tworzenie własnych klas. Możesz nawet modyfikować istniejące klasy języka Ruby. Jeśli nie lubisz sposobu działania łańcuchów lub tablic, możesz to zmienić. Choć prawie zawsze decydowanie się na coś takiego nie jest dobrym pomysłem, język Ruby umożliwia to (w rozdziale 13. przyjrzymy się zaletom i wadom wprowadzania zmian w klasach wbudowanych).

Wprawdzie każdy obiekt języka Ruby jest instancją klasy, ale pojęcie klasy ma mniejsze znaczenie niż obiektu. Wynika to z tego, że obiekty mogą się zmieniać, zyskując metody i zachowania, które nie zostały zdefiniowane w ich klasie. Klasa odpowiada za inicjowanie obiektu w ramach procesu określanego mianem *tworzenia instancji*. Później jednak obiekt staje się niezależny.

Możliwość adaptowania przez obiekty zachowań, jakie nie zostały zapewnione przez ich klasę, to jedna z najważniejszych zasad definiujących projekt Ruby jako język. Jak możesz się domyślić, w ramach różnych kontekstów często będziemy do tego wracać. Na tym etapie bądź jedynie świadom tego, że choć każdy obiekt ma klasę, nie jest ona jedynym wyznacznikiem możliwości obiektu.

Gdy już dysponujesz podstawową wiedzą na temat języka Ruby (w razie wątpliwości możesz powrócić do przedstawionego wcześniej materiału), dokonajmy przeglądu kroków związanych z uruchamianiem programu.

### 1.1.4. Tworzenie i zapisywanie prostego programu

Na tym etapie możesz rozpocząć tworzenie plików programu w utworzonym wcześniej katalogu z przykładowym kodem Ruby. Pierwszym programem będzie konwerter jednostek temperatury ze stopni Celsjusza na stopnie Fahrenheita.

> **UWAGA** Oczywiście praktycznie wykorzystywany konwerter stopni temperatury będzie bazować na liczbach zmiennopozycyjnych. W danych wejściowych i wyjściowych pozostaniemy przy liczbach całkowitych, aby skoncentrować się na kwestiach związanych ze strukturą programu i jego wykonywaniem.

Poniższy przykład zostanie kilkakrotnie zastosowany. Będzie on stopniowy rozszerzany i modyfikowany. W kolejnych iteracjach zostaną zrealizowane następujące działania:

- porządkowanie danych wyjściowych programu,
- akceptowanie danych wejściowych wprowadzonych przez użytkownika za pomocą klawiatury,
- wczytywanie wartości z pliku,
- zapisywanie wyniku programu w pliku.

Pierwsza wersja jest prosta. Skoncentrowano się w niej na procesach tworzenia pliku i uruchamiania programu, a nie na jakiejkolwiek wyszukanej logice programu.

#### TWORZENIE PIERWSZEGO PLIKU PROGRAMU

Za pomocą zwykłego edytora tekstu wpisz kod z listingu 1.1 w pliku tekstowym i zapisz go pod nazwą `c2f.rb` w katalogu z przykładowym kodem.

**Listing 1.1. Prosty konwerter jednostek temperatury ze stopni Celsjusza na stopnie Fahrenheita o ograniczonym zastosowaniu (plik c2f.rb)**

```
celsius = 100
fahrenheit = (celsius * 9 / 5) + 32
puts "Oto wynik: "
puts fahrenheit
puts "."
```

> **UWAGA** Zależnie do używanego systemu operacyjnego możesz mieć możliwość autonomicznego uruchamiania plików programu Ruby, czyli korzystać wyłącznie z nazwy pliku lub nazwy skróconej (np. *c2f*) bez rozszerzenia pliku. Miej jednak świadomość tego, że rozszerzenie pliku *.rb* jest obowiązkowe w niektórych sytuacjach. Przede wszystkim dotyczy to programów, które uwzględniają więcej niż jeden plik (więcej na ten temat dowiesz się w dalszej części rozdziału), a ponadto wymagają mechanizmu wzajemnego znajdowania plików. W książce wszystkie nazwy plików programu Ruby zakończone są rozszerzeniem *.rb* w celu zapewnienia, że przykłady będą działać w przypadku wielu platform, wymagając w jak najmniejszym stopniu działań administracyjnych.

Dysponujesz teraz na dysku kompletnym (choć niewielkim) programem Ruby, który możesz uruchomić.

### *1.1.5. Kierowanie programu do interpretera języka Ruby*

Uruchamianie programu Ruby wiąże się z przekazaniem jego pliku źródłowego (lub plików) do interpretera języka Ruby o nazwie `ruby`. W pewnym sensie wykonasz teraz taką operację. Przekażesz program do interpretera `ruby`, ale zamiast żądać od niego uruchomienia programu, poprosisz o sprawdzenie kodu programu pod kątem błędów składni.

#### SPRAWDZANIE POD KĄTEM BŁĘDÓW SKŁADNI

Jeśli we wzorze konwersji zamiast liczby 32 umieścisz liczbę 31, wystąpi błąd programistyczny. Interpreter języka Ruby w dalszym ciągu bez żadnych problemów uruchomi program i zwróci błędny wynik. Jeśli jednak przypadkiem w drugim wierszu kodu programu pominiesz nawias domykający, wystąpi błąd składni i interpreter nie wykona programu:

```
$ ruby broken_c2f.rb
broken_c2f.rb:5: syntax error, unexpected end-of-input, expecting ')'
```

Błąd został zgłoszony w piątym, czyli ostatnim wierszu programu, ponieważ interpreter języka Ruby oczekuje cierpliwie na stwierdzenie, czy w ogóle zamierzasz domknąć nawias, zanim uzna, że tak nie jest.

W wygodny sposób interpreter języka może sprawdzać programy pod kątem błędów składni bez ich uruchamiania. Interpreter wczytuje plik i informuje o tym, czy składnia jest poprawna. Aby dla pliku przeprowadzić sprawdzanie składni, wykonaj następujące polecenie:

```
$ ruby -cw c2f.rb
```

Opcja wiersza poleceń `-cw` stanowi zapis skrócony dwóch opcji: `-c` i `-w`. Opcja `-c` powoduje *sprawdzanie pod kątem błędów składni*. Opcja `-w` aktywuje wyższy poziom ostrzeżeń: interpreter języka Ruby będzie zgłaszać zastrzeżenia, jeśli wykonano działania przez niego dopuszczane, ale budzące wątpliwości z innych punktów widzenia niż składnia.

Przy założeniu, że poprawnie wpisano nazwę pliku, na ekranie powinien zostać wyświetlony następujący komunikat:

```
Syntax OK
```

### URUCHAMIANIE PROGRAMU

Aby uruchomić program, przekaż jeszcze raz plik interpreterowi, lecz tym razem bez połączonych opcji `-c` i `-w`:

```
$ ruby c2f.rb
```

Jeśli wszystko się powiedzie, zostaną zwrócone dane wyjściowe obliczeń:

```
Oto wynik:
212
.
```

Wynik obliczeń jest poprawny, ale nieładnie wyglądają dane wyjściowe rozmieszczone w trzech wierszach.

### DRUGA ITERACJA KONWERTERA

Problem może zostać sprowadzony do różnicy między poleceniami `puts` i `print`. Polecenie `puts` dodaje znak nowego wiersza na końcu wyświetlonego łańcucha, jeśli nie jest on już zakończony takim znakiem. Z kolei polecenie `print` wyświetla żądany łańcuch, a następnie kończy działanie. Polecenie nie powoduje automatycznego przejścia do następnego wiersza.

Aby usunąć ten problem, zmień pierwsze dwa polecenia `puts` na polecenie `print`:

```
print "Oto wynik: "
print fahrenheit
puts "."
```

Zwróć uwagę na znak spacji po znaku dwukropka; zapewnia ona, że między dwukropkiem i liczbą pojawi się odstęp. Dane wyjściowe mają teraz następującą postać:

```
Oto wynik: 212.
```

`puts` to skrót od słów *put* (wyświetl) *string* (łańcuch). Choć słowo *put* może nie wskazywać intuicyjnie przejścia do następnego wiersza, właśnie to powoduje polecenie `puts`. Podobnie jak polecenie `print` wyświetla ono żądane dane, ale też automatycznie zapewnia przejście do następnego wiersza. Jeśli zażądasz od polecenia `puts` wyświetlenia wiersza, który jest już zakończony znakiem nowego wiersza, nie doda ono takiego wiersza.

Jeżeli korzystano z narzędzi wyświetlania danych w językach, w których nie jest automatycznie dodawany znak nowego wiersza (np. funkcja `print` w języku Perl), to w przypadku języka Ruby może zdarzyć się, że napiszesz kod podobny do następującego, aby wyświetlić wartość z występującym po niej znakiem nowego wiersza:

```
print fahrenheit, "\n"
```

Nie będzie to jednak prawie nigdy wymagane, ponieważ polecenie `puts` automatycznie dodaje znak nowego wiersza. Z czasem przywykniesz do sposobu działania tego polecenia, a także do innych idiomów i konwencji związanych z językiem Ruby.

> **OSTRZEŻENIE** W przypadku niektórych platform (dotyczy to zwłaszcza systemu Windows) na końcu działania programu wyświetlany jest dodatkowy znak nowego wiersza. Oznacza to, że trudne będzie do wykrycia polecenie `print`, które w rzeczywistości powinno być zastąpione poleceniem `puts`, ponieważ polecenie `print` będzie działać jak polecenie `puts`. Świadomość różnicy występującej między tymi poleceniami, a ponadto wybranie tego z nich, które jest wymagane na podstawie zwykłego sposobu działania, powinno być wystarczające do zapewnienia uzyskania żądanych wyników.

Po przyjrzeniu się danym wyjściowym prezentowanym na ekranie rozszerzmy trochę operacje wejścia-wyjścia w celu uwzględnienia danych wprowadzanych za pomocą klawiatury i operacji na plikach.

### *1.1.6. Operacje wejścia-wyjścia związane z plikami i danymi wprowadzanymi przy użyciu klawiatury*

Język Ruby oferuje wiele metod odczytywania danych podczas wykonywania programu, zarówno wprowadzonych z wykorzystaniem klawiatury, jak i znajdujących się w plikach na dysku. Metody te okażą się przydatne, jeśli nie w przypadku pisania każdej aplikacji, to prawie na pewno w trakcie tworzenia kodu Ruby. Metody umożliwią w środowisku roboczym wykonywanie operacji związanych z konserwacją, konwertowaniem, porządkowaniem lub wprowadzaniem zmian w inny sposób. W dalszej części rozdziału przyjrzymy się niektórym metodom przetwarzania danych wejściowych. W rozdziale 12. w szerszym zakresie omówiono operacje wejścia-wyjścia.

#### DANE WPROWADZANE ZA POMOCĄ KLAWIATURY

Program, który bez końca informuje o tym, że stu stopniom Celsjusza odpowiada 212 stopni Fahrenheita, ma ograniczoną przydatność. Bardziej wartościowy program pozwala podać temperaturę w stopniach Celsjusza i uzyskać odpowiadającą jej wartość wyrażoną w stopniach Fahrenheita.

Modyfikowanie programu w celu zapewnienia takiej funkcjonalności obejmuje dodanie kilku kroków oraz zastosowanie po jednej metodzie z tabel 1.1 i 1.2: `gets` (pobiera wiersz danych wprowadzonych przy użyciu klawiatury) i `to_i` (dokonuje konwersji na liczbę całkowitą). Druga z tych metod została już wcześniej przedstawiona. Ponieważ jest to nowy program, a nie tylko modyfikacja, w nowym pliku umieść wersję kodu podaną w listingu 1.2. Plikowi nadaj nazwę *c2fi.rb* (*i* jest skrótem od słowa *interaktywny*).

**Listing 1.2. Interaktywny konwerter temperatur (plik c2fi.rb)**

```
print "Witaj. Proszę podać wartość w stopniach Celsjusza: "
celsius = gets
fahrenheit = (celsius.to_i * 9 / 5) + 32
print "Odpowiednik w stopniach Fahrenheita wynosi "
print fahrenheit
puts "."
```

Kilka przykładowych uruchomień demonstruje działanie nowego programu:

```
$ ruby c2fi.rb
Witaj. Proszę podać wartość w stopniach Celsjusza: 100
Odpowiednik w stopniach Fahrenheita wynosi 212.
$ ruby c2fi.rb
Witaj. Proszę podać wartość w stopniach Celsjusza: 23
Odpowiednik w stopniach Fahrenheita wynosi 73.
```

**Skracanie kodu**

Możliwe jest znaczne skrócenie kodu z listingu 1.2 przez skonsolidowanie operacji związanych z danymi wejściowymi, obliczeniami i danymi wyjściowymi. Bardziej zwięzły kod po przebudowie ma następującą postać:

```
print "Witaj. Proszę podać wartość w stopniach Celsjusza: "
print "Odpowiednik w stopniach Fahrenheita wynosi ", gets.to_i * 9 / 5 + 32, ".\n"
```

W tej wersji kodu dokonano oszczędności kosztem zmiennych. Nie ma już żadnych zmiennych, ale wymagane jest prześledzenie przez osobę czytającą kod bardziej zwartego (lecz krótszego!) zestawu wyrażeń. W dowolnym programie występuje zwykle kilka lub wiele miejsc, w przypadku których konieczne jest podjęcie decyzji dotyczącej tego, czy kod będzie dłuższy (ale być może bardziej przejrzysty?), czy krótszy (ale raczej mało zrozumiały). Czasami coś krótszego może być bardziej przejrzyste. Wszystko to stanowi część stylu tworzenia kodu w języku Ruby.

Dysponujemy teraz uogólnionym, jeśli nie cechującym się szczególną subtelnością rozwiązaniem problemu związanego z konwersją stopni Celsjusza na stopnie Fahrenheita. Rozszerzmy rozważania o dane wejściowe z pliku.

### ODCZYT Z PLIKU

Odczytywanie danych z pliku w programie Ruby nie jest wiele trudniejsze, a przynajmniej w wielu przypadkach, niż odczytywanie wiersza danych wprowadzonych za pomocą klawiatury. Następna wersja konwertera temperatur będzie wczytywać z pliku jedną liczbę i wykonywać dla niej konwersję ze stopni Celsjusza na stopnie Fahrenheita.

Najpierw utwórz nowy plik o nazwie *temp.dat* (dane z temperaturą), który zawiera jeden wiersz z jedną liczbą:

```
100
```

Utwórz trzeci plik programu o nazwie *c2fdwe.rb* (*dwe* to skrót od słów *dane wejściowe*), którego zawartość prezentuje listing 1.3.

**Listing 1.3. Konwerter temperatur korzystający z danych wejściowych pliku (plik c2fin.rb)**

```
puts "Odczytywanie wartości temperatury w stopniach Celsjusza z pliku danych..."
num = File.read("temp.dat")
celsius = num.to_i
fahrenheit = (celsius * 9 / 5) + 32
puts "Liczba to " + num
print "Wynik: "
puts fahrenheit
```

Tym razem przykładowe uruchomienie programu powoduje zwrócenie następujących danych wyjściowych:

```
$ ruby c2fin.rb
Odczytywanie wartości temperatury w stopniach Celsjusza z pliku danych...
Liczba to 100
Wynik: 212
```

Jeśli zmienisz liczbę w pliku, to oczywiście wynik będzie inny.

A może by tak zapisać wynik obliczeń w pliku?

### ZAPIS W PLIKU

Najprostsza operacja zapisu w pliku jest tylko odrobinę bardziej wyszukana niż najprostsza operacja odczytu z pliku. Jak widać w listingu 1.4, w przypadku zapisywania w pliku podstawowym dodatkowym krokiem jest określenie *trybu* pliku. W tym przypadku jest to tryb `w` (`w` to skrót od słowa *write*). Zapisz w pliku *c2fdwy.rb* wersję programu z kodem z tego listingu, a następnie uruchom program.

**Listing 1.4. Konwerter temperatur umieszczający dane wyjściowe w pliku (plik c2fdwy.rb)**

```
print "Witaj. Proszę podać wartość w stopniach Celsjusza: "
celsius = gets.to_i
fahrenheit = (celsius * 9 / 5) + 32
puts "Zapisywanie wyniku w pliku danych wyjściowych temp.out"
fh = File.new("temp.out", "w")
fh.puts fahrenheit
fh.close
```

Wywołanie metody `fh.puts fahrenheit` powoduje zapisanie wartości zmiennej `fahrenheit` w pliku, dla którego obiekt `fh` to uchwyt operacji zapisu. Jeśli sprawdzisz plik *temp.out*, powinno być widoczne, że zawiera odpowiednik wyrażony w stopniach Fahrenheita dla dowolnej wpisanej liczby.

W ramach ćwiczenia możesz spróbować połączyć wcześniejsze przykłady do postaci programu Ruby, który odczytuje liczbę z pliku i zapisuje w innym pliku wynik konwersji na stopnie Fahrenheita. Tymczasem po zaznajomieniu się z podstawową składnią języka Ruby w dalszej kolejności zajmiemy się omówieniem jego instalacji. Z kolei to pozwoli dowiedzieć się, jak w języku tym zarządzane są rozszerzenia i biblioteki.

## 1.2. Anatomia instalacji języka Ruby

Zainstalowanie języka Ruby w systemie oznacza, że na dysku istnieje kilka katalogów zawierających wiele bibliotek i plików pomocniczych. Przeważnie interpreter języka Ruby potrafi znaleźć to, czego potrzebuje, bez przekazywania mu dodatkowych informacji. Jednak znajomość przebiegu instalacji języka Ruby stanowi część odpowiedniego zaznajomienia się z podstawami dotyczącymi tego języka.

**Sprawdzanie kodu źródłowego języka Ruby**

Oprócz dostępu do drzewa katalogowego instalacji języka Ruby możesz też mieć dostęp do umieszczonego na komputerze drzewa kodu źródłowego tego języka. Jeśli taki dostęp nie istnieje, masz możliwość pobrania takiego kodu ze strony internetowej języka Ruby. Drzewo kodu źródłowego zawiera wiele plików języka Ruby trafiających do ostatecznej instalacji, a także mnóstwo plików języka C, które są kompilowane do postaci plików obiektów, a następnie instalowane. Ponadto drzewo to przechowuje pliki informacyjne, takie jak ChangeLog (dziennik zmian) i licencje oprogramowania.

Język Ruby potrafi określić miejsce, w którym znajdują się jego pliki instalacyjne. Aby uzyskać taką informację, gdy otwarto sesję narzędzia irb, musisz w niej załadować wcześniej pakiet bibliotek języka Ruby o nazwie `rbconfig`. Pakiet jest interfejsem zapewniającym dostęp do wielu uwzględnionych podczas kompilacji informacji konfiguracyjnych dotyczących instalacji języka Ruby. W celu załadowania tych informacji przez narzędzie irb użyj jego flagi wiersza poleceń `-r` i nazwy pakietu:

```
$ irb --simple-prompt -rrbconfig
```

Możesz teraz zażądać informacji. Na przykład masz możliwość stwierdzenia, gdzie zostały zainstalowane pliki wykonywalne języka Ruby (w tym pliki interpretera ruby i narzędzia irb):

```
>> RbConfig::CONFIG["bindir"]
```

`RbConfig::CONFIG` to *stała* odnosząca się do *tablicy asocjacyjnej* (ang. *hash*; rodzaj struktury danych), w której w przypadku języka Ruby są przechowywane informacje konfiguracyjne. Łańcuch `"bindir"` to *klucz* tablicy asocjacyjnej. Odpytywanie tablicy asocjacyjnej przy użyciu tego klucza pozwala uzyskać odpowiadającą mu *wartość* tablicy, która jest nazwą katalogu instalacyjnego z plikami binarnymi.

Reszta informacji o konfiguracji jest udostępniana w ten sam sposób, czyli w postaci wartości znajdujących się wewnątrz struktury danych konfiguracyjnych, które są dostępne za pomocą konkretnych kluczy tablicy asocjacyjnej. Aby uzyskać dodatkowe informacje konfiguracyjne, w poleceniu `irb` musisz zastąpić łańcuch `bindir` innymi terminami. Jednak za każdym razem używana jest ta sama podstawowa formuła: `rbConfig::CONFIG["termin"]`. W tabeli 1.5 wyszczególniono terminy i katalogi, do których się one odwołują.

Poniżej podsumowano główne katalogi instalacji oraz ich zawartość. Nie musisz ich wszystkich pamiętać, ale w razie potrzeby (lub jeśli masz zamiar przejrzeć je i sprawdzić wybrane przykłady kodu Ruby) należy wiedzieć, jak je znaleźć.

**Tabela 1.5.** Kluczowe katalogi instalacji języka Ruby oraz powiązane z nimi terminy pakietu RbConfig

| Termin | Zawartość katalogu |
|---|---|
| `rubylibdir` | Standardowe biblioteki języka Ruby. |
| `bindir` | Narzędzia wiersza poleceń języka Ruby. |
| `archdir` | Rozszerzenia i biblioteki specyficzne dla architektury (skompilowane pliki binarne). |
| `sitedir` | Własne lub zewnętrzne rozszerzenia i biblioteki (utworzone w języku Ruby). |
| `vendordir` | Zewnętrzne rozszerzenia i biblioteki (utworzone w języku Ruby). |
| `sitelibdir` | Własne rozszerzenia języka Ruby (utworzone w tym języku). |
| `sitearchdir` | Własne rozszerzenia języka Ruby (utworzone w języku C). |

### 1.2.1. Podkatalog standardowych bibliotek języka Ruby

W katalogu *rubylibdir* znajdziesz pliki programów napisanych w języku Ruby. Pliki te udostępniają narzędzia standardowych bibliotek, które mogą być wymagane przez Twoje programy, jeśli potrzebne będą zapewniane przez nie funkcje.

Oto niektóre pliki znajdujące się w tym katalogu:

- *cgi.rb*. Narzędzia ułatwiające programowanie z wykorzystaniem interfejsu CGI.
- *fileutils.rb*. Narzędzia pozwalające na łatwe modyfikowanie plików programów Ruby.
- *tempfile.rb*. Mechanizm automatyzacji tworzenia plików tymczasowych.
- *drb.rb*. Narzędzie używane na potrzeby programowania rozproszonego z wykorzystaniem języka Ruby.

Niektóre spośród bibliotek standardowych, takie jak biblioteka drb (wymieniona jako ostatnia na powyższej liście), złożone są z więcej niż jednego pliku. W katalogu *rubylibdir* znajdziesz zarówno plik *drb.rb*, jak i cały podkatalog *drb* zawierający komponenty biblioteki drb.

Przejrzenie katalogu *rubylibdir* pozwoli się dobrze zorientować (jeśli nawet początkowo będzie to przytłaczające), dla jak wielu zadań język Ruby zapewnia narzędzia programistyczne. Choć większość programistów korzysta jedynie z podzbioru tych narzędzi, nawet taka część ogromnej kolekcji bibliotek programistycznych oferuje wiele możliwości podczas pracy.

### 1.2.2. Katalog rozszerzeń języka C (RbConfig::CONFIG[archdir])

Katalog *archdir*, który zwykle zlokalizowany jest jeden poziom niżej niż katalog *rubylibdir*, zawiera rozszerzenia i biblioteki specyficzne dla architektury. Pliki w tym katalogu mają zazwyczaj nazwy zakończone rozszerzeniami *.so*, *.dll* lub *.bundle* (zależnie od używanego sprzętu i systemu operacyjnego). Pliki te są rozszerzeniami języka C, czyli plikami binarnymi ładowanymi w środowisku wykonawczym, które są generowane przy użyciu kodu rozszerzeń języka C utworzonego w języku Ruby. Pliki są kompilowane do postaci binarnej w ramach procesu instalacji języka Ruby.

Podobnie jak jest w przypadku plików programów języka Ruby w katalogu *rubylibdir*, pliki w katalogu *archdir* zawierają komponenty standardowych bibliotek, które

mogą być ładowane we własnych programach (w katalogu znajdziesz między innymi plik rozszerzenia *rbconfig* używanego wraz z narzędziem irb do ujawniania nazw katalogów). Choć pliki te nie mają formatu możliwego do odczytania przez użytkownika, interpreter języka Ruby potrafi załadować je, gdy zostanie to od niego zażądane. Z punktu widzenia programisty korzystającego z języka Ruby wszystkie standardowe biblioteki są w równym stopniu przydatne, niezależnie od tego, czy utworzono je w języku Ruby, czy w języku C, a następnie skompilowano do postaci binarnej.

Pliki instalowane w katalogu *archdir* różnią się w przypadku poszczególnych instalacji, zależnie od tego, jakie rozszerzenia zostały skompilowane. Z kolei to jest zależne od tego, czego zażądała osoba przeprowadzająca kompilację, a także od tego, jakie rozszerzenia interpreter języka Ruby był w stanie skompilować.

### 1.2.3. Katalogi site_ruby (RbConfig::CONFIG[sitedir]) i vendor_ruby (RbConfig::CONFIG[vendordir])

Instalacja języka Ruby obejmuje podkatalog o nazwie *site_ruby*, w którym programista i/lub administrator systemu przechowują zewnętrzne rozszerzenia i biblioteki. Część z nich może zawierać napisany przez Ciebie kod, a część może być narzędziami pobranymi z witryn innych osób oraz archiwami bibliotek języka Ruby.

Katalog *site_ruby* stanowi analogię dla głównego katalogu instalacji języka Ruby w tym sensie, że zawiera własne podkatalogi przeznaczone dla rozszerzeń języków Ruby i C (są to odpowiednio katalogi *sitelibdir* i *sitearchdir* w terminach `RbConfig::CONFIG`). Gdy zażądasz rozszerzenia, interpreter języka Ruby sprawdza, czy znajduje się ono w tych podkatalogach katalogu *site_ruby*, a także w katalogach *rubylibdir* i *archdir*.

Oprócz katalogu *site_ruby* znajdziesz katalog *vendor_ruby*. Niektóre rozszerzenia zewnętrzne automatycznie instalują się w tym katalogu. Po raz pierwszy katalog pojawił się w wersji 1.9 języka Ruby. W dalszym ciągu rozwijana jest standardowa metoda określająca, w którym z tych dwóch katalogów zostaną umieszczone poszczególne pakiety.

### 1.2.4. Katalog gems

Narzędzie RubyGems zapewnia standardową metodę tworzenia pakietów bibliotek języka Ruby i dystrybuowania ich. Podczas instalacji pakietów *gem* pliki bibliotek wyłączone z pakunku trafiają do katalogu *gems*. Katalog ten nie jest wyszczególniony w strukturze danych konfiguracyjnych, ale zwykle znajduje się na tym samym poziomie co katalog *site_ruby*. Jeśli udało się znaleźć ten katalog, sprawdź, co jeszcze innego zostało obok niego zainstalowane. W punkcie 1.4.5 zamieszczono więcej informacji o pakietach *gem*.

Przyjrzyjmy się teraz mechanizmom i semantyce związanym z tym, jak język Ruby korzysta z własnych rozszerzeń, a także z tych, które możesz sam utworzyć lub zainstalować.

## 1.3. Rozszerzenia i biblioteki programistyczne języka Ruby

Pierwsza ważna uwaga, o jakiej trzeba pamiętać w trakcie lektury tego podrozdziału, dotyczy tego, że nie pełni on roli zestawienia standardowych bibliotek. Jak wspomniano we wprowadzeniu, książka nie ma na celu dokumentowania języka Ruby. Ma ona umożliwić naukę tego języka i stanie się członkiem związanej z nim społeczności, aby możliwe było ciągłe poszerzanie własnych horyzontów.

A zatem celem niniejszego podrozdziału jest zaprezentowanie sposobu działania rozszerzeń, a także tego, jak spowodować uruchomienie przez interpreter języka Ruby jego rozszerzeń. Ponadto wyjaśniono różnice między umożliwiającymi to metodami oraz omówiono architekturę, która pozwala tworzyć własne rozszerzenia i biblioteki.

Rozszerzenia dołączone do języka Ruby są zwykle określane zbiorczo jako *standardowa biblioteka*. Obejmuje ona rozszerzenia przeznaczone dla bardzo różnych projektów i zadań, takich jak zarządzanie bazami danych, obsługa sieci, specjalistyczne operacje matematyczne, przetwarzanie danych XML itp. Dokładna zawartość standardowej biblioteki zazwyczaj się zmienia, przynajmniej w niewielkim stopniu, przy okazji każdej nowej wersji języka Ruby. Jednak większość powszechnie używanych bibliotek raczej pozostaje, gdy potwierdzą swoją przydatność.

Kluczem do zastosowania rozszerzeń i bibliotek jest metoda `require` wraz z blisko powiązaną z nią metodą `load`. Metody te umożliwiają ładowanie rozszerzeń w środowisku wykonawczym, w tym własnoręcznie utworzonych rozszerzeń. Najpierw przyjrzymy się im ogólnie, a następnie poszerzymy omówienie o wykorzystanie metod do ładowania rozszerzeń wbudowanych.

### 1.3.1. Ładowanie plików i rozszerzeń zewnętrznych

Przechowywanie programu w jednym pliku może być poręczne, ale zaczyna być raczej utrudnieniem niż korzyścią, gdy kod liczy setki, tysiące albo setki tysięcy wierszy. Rozdzielenie gdzieś w obrębie kodu programu na osobne pliki nabiera sporego sensu. Język Ruby ułatwia ten proces za pomocą metod `require` i `load`. Zajmiemy się najpierw metodą `load`, która w porównaniu z drugą metodą ma prostszą konstrukcję.

**Składnik, rozszerzenie lub biblioteka?**

Dane ładowane w programie w czasie jego działania są określane przy użyciu kilku różnych nazw. *Składnik* to termin najbardziej abstrakcyjny i rzadko spotykany, z wyłączeniem specjalistycznych zastosowań wymagających składnika (w tym przypadku stosowana jest metoda `require`). *Biblioteka* to bardziej konkretny i częściej używany termin. Odwołuje się on do faktycznego kodu, a także do podstawowego faktu określającego, że istnieje zestaw narzędzi programistycznych, które mogą być ładowane. *Rozszerzenie* może odnosić się do dowolnej biblioteki dodatkowej możliwej do załadowania, ale często termin ten identyfikuje bibliotekę języka Ruby napisaną za pomocą języka programowania C, a nie języka Ruby. Jeśli powiesz komuś, że utworzyłeś rozszerzenie języka Ruby, prawdopodobnie ta osoba przyjmie, że masz na myśli to, że zostało ono napisane w języku C.

Aby sprawdzić poniższe przykłady, niezbędny będzie program podzielony na dwa pliki. Pierwszy plik o nazwie *loaddemo.rb* powinien zawierać następujący kod Ruby:

```
puts "To jest pierwszy (główny) plik programu."
load "loadee.rb"
puts "I z powrotem do pierwszego pliku."
```

Gdy interpreter języka Ruby napotka wywołanie metody `load`, wczytuje drugi plik. Zawartość tego pliku o nazwie *loadee.rb* powinna być następująca:

```
puts "> To jest drugi plik."
```

Dwa pliki powinny znajdować się w tym samym katalogu (prawdopodobnie w katalogu z przykładowym kodem). Po uruchomieniu programu `loaddemo.rb` z poziomu wiersza poleceń zostaną wyświetlone następujące dane wyjściowe:

```
To jest pierwszy (główny) plik programu.
> To jest drugi plik.
I z powrotem do pierwszego pliku.
```

Dane wyjściowe pozwalają stwierdzić, jakie wiersze kodu z poszczególnych plików są wykonywane, a także w jakiej kolejności.

Wywołanie metody `load` w pliku *loaddemo.rb* zawiera nazwę pliku *loadee.rb* jako argument tej metody:

```
load "loadee.rb"
```

Jeśli ładowany plik znajduje się w bieżącym katalogu roboczym, interpreter języka Ruby będzie w stanie znaleźć go przy użyciu nazwy. W przeciwnym razie interpreter poszuka pliku w *ścieżce ładowania*.

### 1.3.2. Ładowanie pliku określonego w domyślnej ścieżce ładowania

Ścieżka ładowania interpretera języka Ruby jest listą katalogów, w których szukane są pliki do załadowania. W celu wyświetlenia nazw tych katalogów należy sprawdzić zawartość specjalnej zmiennej globalnej `$:`. Udostępniona zawartość zależy od używanej platformy. Dane uzyskane podczas inspekcji typowej ścieżki ładowania w systemie Mac OS X są podobne do przedstawionych w poniższym przykładzie uwzględniającym katalog *.rvm*, w którym narzędzie Ruby Version Manager przechowuje wybrane wersje języka Ruby:

```
$ ruby -e 'puts $:'   <------ Flaga -e wskazuje, że interpreterowi przekazujesz skrypt wstawiany
/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/site_ruby/2.1.0
/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/site_ruby/2.1.0/x86_64-
darwin12.0
/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/site_ruby
/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/vendor_ruby/2.1.0
/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/vendor_ruby/2.1.0/x86_64-
darwin12.0
/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/vendor_ruby
/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/2.1.0
/Users/dblack/.rvm/rubies/ruby-2.1.0/lib/ruby/2.1.0/x86_64-darwin12.0
```

Choć w przypadku używanego komputera część ścieżki widoczna po lewej stronie łańcucha `ruby-2.1.0` może być inna (np. `/usr/local/lib/`), podstawowy wzorzec podkatalogów pozostanie taki sam. Podczas ładowania pliku interpreter języka Ruby szuka go w każdym z wyszczególnionych katalogów, od góry do dołu listy.

> **UWAGA** Bieżący katalog roboczy reprezentowany zwykle przez znak kropki w rzeczywistości nie jest uwzględniany w ścieżce ładowania. Polecenie ładujące działa tak, jakby ten znak był obecny, ale jest to szczególny przypadek programistyczny.

Nawigacja dotycząca katalogów względnych w poleceniu `load` jest możliwa za pomocą tradycyjnego symbolu złożonego z dwóch kropek, który powoduje przejście w górę hierarchii katalogów:

```
load "../extras.rb"
```

Zauważ, że jeśli w czasie działania programu zostanie zmieniony bieżący katalog, zmienią się również odwołania katalogów względnych.

> **UWAGA** Pamiętaj o tym, że `load` to metoda wykonywana w momencie napotkania jej w pliku przez interpreter języka Ruby. Szukając dyrektyw ładujących, interpreter nie przeszukuje całego pliku. Natrafia na nie w trakcie działania. Oznacza to, że możesz ładować pliki, których nazwy są określane dynamicznie podczas pracy programu. Masz nawet możliwość opakowania wywołania metody `load` za pomocą instrukcji warunkowej. W tym przypadku wywołanie zostanie wykonane tylko wtedy, gdy warunek będzie prawdziwy.

Możliwe jest też wymuszenie znalezienia pliku przez metodę `load`, niezależnie od zawartości ścieżki ładowania. W tym celu należy zapewnić metodzie pełną ścieżkę do pliku:

```
load "/home/users/dblack/book/code/loadee.rb"
```

Jest to oczywiście mniej elastyczne rozwiązanie niż zastosowanie ścieżki ładowania lub względnych ścieżek katalogów, ale może okazać się przydatne zwłaszcza wtedy, gdy ścieżka bezwzględna jest przechowywana jako łańcuch w zmiennej, a ponadto ma zostać załadowany reprezentowany przez nią plik.

Wywołanie metody `load` zawsze powoduje załadowanie żądanego pliku, niezależnie od tego, czy plik ten został już załadowany, czy nie. Jeśli pomiędzy operacjami ładowania plik ulegnie zmianie, priorytet ma jego nowa wersja, która nadpisuje cokolwiek w oryginalnej wersji. Może to być przydatne szczególnie wtedy, gdy otwarta jest sesja narzędzia irb podczas jednoczesnego modyfikowaniu pliku w edytorze, a ponadto pożądane jest natychmiastowe sprawdzenie efektów zmian.

Druga metoda służąca do ładowania plików — `require` — również przeszukuje katalogi podane w domyślnej ścieżce ładowania. Jednak ta metoda oferuje kilka opcji, których pozbawiona jest metoda `load`.

### *1.3.3. Żądanie składnika*

Główną różnicą między metodami `load` i `require` jest to, że w przypadku wywołania więcej niż raz z tymi samymi argumentami druga z nich nie ładuje ponownie plików, które zostały już załadowane. Interpreter języka Ruby śledzi, jakie pliki zostały zażądane, i nie duplikuje działań.

Metoda `require` jest bardziej abstrakcyjna niż metoda `load`. *Mówiąc wprost, nie żądasz pliku, lecz składnika*. Zwykle odbywa się to nawet bez podawania rozszerzenia nazwy pliku. Aby przekonać się, jak to działa, następujący wiersz w pliku *loaddemo.rb*:

```
load "loadee.rb"
```

zmień do postaci:

```
require "./loadee.rb"
```

Po uruchomieniu pliku *loaddemo.rb* uzyskasz taki sam wynik jak wcześniej, nawet pomimo tego, że nie została podana pełna nazwa pliku do załadowania.

Postrzegając `loadee` jako składnik, a nie jako plik, metoda `require` umożliwia traktowanie rozszerzeń napisanych w języku Ruby w taki sam sposób, w jaki są traktowane rozszerzenia utworzone w języku C. Inaczej mówiąc, pliki zakończone rozszerzeniem *.rb* mogą być traktowane identycznie jak pliki z rozszerzeniem *.so*, *.dll* lub *.bundle*.

**Określanie katalogu roboczego**

Metoda `require` nie ma informacji na temat bieżącego katalogu roboczego (`.`). Możesz go określić jawnie w następujący sposób:

```
require "./loadee.rb"
```

Możliwe jest też dołączenie tego katalogu do ścieżki ładowania za pomocą tablicowego operatora dołączania:

```
$: << "."
```

Oznacza to, że nie jest konieczne podawanie bieżącego katalogu roboczego w wywołaniach metody `require`:

```
require "loadee.rb"
```

Istnieje też, tak jak w przypadku metody `load`, możliwość przekazania metodzie `require` pełnej ścieżki. Metoda ta pobierze plik/składnik. Możliwy jest wariant mieszany. Na przykład zaprezentowana poniżej składnia zadziała nawet pomimo tego, że łączy specyfikację ścieżki statycznej z bardziej abstrakcyjną składnią składnika na końcu ścieżki:

```
require "/home/users/dblack/book/code/loadee.rb"
```

Choć metoda `load` jest przydatna, a zwłaszcza w przypadku zamiaru załadowania pliku więcej niż raz, metoda `require` zapewnia stosowane na co dzień rozwiązanie, które posłuży do ładowania rozszerzeń i bibliotek języka Ruby, zarówno tych standardowych, jak i innych. Ładowanie składników standardowych bibliotek nie jest w żadnym stopniu trudniejsze niż ładowanie pliku *loadee*. Wystarczy zażądać wymaganych danych. Gdy to nastąpi, i oczywiście w zależności od tego, czym jest rozszerzenie, dostępne będą nowe klasy i metody. Oto kompletny przykład w sesji narzędzia irb:

```
>> "David Black".scanf("%s%s")
NoMethodError: undefined method `scanf' for "David Black":String  ←❶
>> require "scanf"        ←❷
=> true
>> "David Black".scanf("%s%s")
=> ["David", "Black"]     ←❸
```

Pierwsze wywołanie metody `scanf` kończy się błędem ❶. Jednak po wywołaniu metody `require` ❷, a ponadto bez żadnej dodatkowej interwencji ze strony programisty obiekty łańcuchowe (np. `"David Black"`) odpowiadają na komunikat metody `scanf`. W tym przykładzie ❸ żądamy wyodrębnienia z oryginalnego łańcucha dwóch kolejnych łańcuchów ze znakiem odstępu jako niejawnym operatorem.

### 1.3.4. Polecenie require_relative

Trzeci sposób ładowania plików zapewnia polecenie `require_relative`. Ładuje ono składnik, przeprowadzając wyszukiwanie względem katalogu zawierającego plik, z którego polecenie zostało wywołane. A zatem wynik z poprzedniego przykładu możesz uzyskać w następujący sposób:

```
require_relative "loadee"
```

Nie wymaga to modyfikowania ścieżki ładowania w celu uwzględnienia bieżącego katalogu. Polecenie `require_relative` jest wygodne przy nawigacji w obrębie lokalnej hierarchii katalogów. Oto przykład:

```
require_relative "lib/music/sonata"
```

Rozdział zakończymy omówieniem narzędzi wiersza poleceń dołączonych do języka Ruby.

## 1.4. Standardowe narzędzia i aplikacje języka Ruby

Po zainstalowaniu języka Ruby uzyskujesz zestaw ważnych narzędzi wiersza poleceń, które są instalowane w dowolnym katalogu skonfigurowanym jako katalog danych binarnych. Zwykle jest to katalog */usr/local/bin*, */usr/bin* lub */opt* (w celu potwierdzenia tego możesz użyć polecenia `require "rbconfig"` i sprawdzić stałą `RbConfig::CONFIG["bindir"]`). Są to następujące narzędzia:

- `ruby` — interpreter.
- `irb` — interaktywny interpreter języka Ruby.
- `rdoc` i `ri` — narzędzia dokumentacji języka Ruby.
- `rake` — odpowiednik narzędzia `make` w języku Ruby służący do zarządzania zadaniami.
- `gem` — narzędzie do zarządzania pakietami aplikacji i bibliotek języka Ruby.
- `erb` — system tworzenia szablonów.
- `testrb` — narzędzie wysokiego poziomu używane na potrzeby środowiska testowania kodu Ruby.

W tym podrozdziale przyjrzymy się wszystkim tym narzędziom, z wyjątkiem narzędzi `erb` i `testrb`. Są one przydatne w określonych sytuacjach, ale nie są najważniejsze podczas zdobywania podstawowych umiejętności związanych z korzystaniem z języka Ruby.

Nie musisz od razu zapamiętywać wszystkich metod przedstawionych w podrozdziale. Zamiast tego po prostu go przeczytaj, aby zorientować się ogólnie w ich przeznaczeniu. Wkrótce będziesz często korzystał z części zamieszczonego tutaj materiału

(dotyczy to zwłaszcza wybranych opcji wiersza poleceń oraz narzędzia `ri`). Część treści podrozdziału będzie stopniowo stawać się przydatna w miarę coraz lepszego poznawania języka Ruby.

### 1.4.1. Opcje wiersza poleceń interpretera

Przy uruchamianiu interpretera języka Ruby z poziomu wiersza poleceń możesz podać nie tylko nazwę pliku programu, ale też co najmniej jedną opcję, co już zostało zaprezentowane w rozdziale. Wybrane opcje nakazują interpreterowi działanie w określony sposób i/lub wykonywanie konkretnych operacji.

Język Ruby oferuje ponad 20 opcji wiersza poleceń. Niektóre z nich są rzadko używane, natomiast inne codziennie przez wielu programistów korzystających z tego języka. W tabeli 1.6 zestawiono najczęściej używane opcje.

**Tabela 1.6.** Podsumowanie często stosowanych opcji wiersza poleceń języka Ruby

| Opcja | Opis | Przykład użycia |
|---|---|---|
| `-c` | Powoduje sprawdzenie składni kodu w pliku programu bez uruchamiania go. | `ruby -c c2f.rb` |
| `-w` | Wyświetla komunikaty ostrzeżenia podczas wykonywania programu. | `ruby -w c2f.rb` |
| `-e` | Powoduje wykonanie kodu podanego w znakach cudzysłowu w wierszu poleceń. | `ruby -e 'puts "Demo kodu!"'` |
| `-l` | Tryb wiersza: powoduje wyświetlenie znaku nowego wiersza po każdym wierszu danych wyjściowych. | `ruby -le 'print "+ znak nowego wiersza!"'` |
| `-r`*nazwa* | Opcja wymaga podania składnika z nazwą. | `ruby -rprofile` |
| `-v` | Wyświetla informacje o wersji języka Ruby, a ponadto powoduje wykonanie programu w trybie szczegółowych informacji. | `ruby -v` |
| `--version` | Wyświetla informacje o wersji języka Ruby. | `ruby --version` |
| `-h` | Wyświetla informacje o wszystkich opcjach wiersza poleceń interpretera. | `ruby -h` |

Omówmy bardziej szczegółowo każdą z powyższych opcji.

#### SPRAWDZANIE SKŁADNI (-C)

Opcja `-c` nakazuje interpreterowi języka Ruby sprawdzenie kodu w co najmniej jednym pliku pod kątem dokładności składniowej bez wykonywania kodu. Opcja jest zwykle stosowana w połączeniu z opcją `-w`.

#### WŁĄCZANIE OSTRZEŻEŃ (-W)

Uruchamianie programu z opcją `-w` powoduje załadowanie go przez interpreter w trybie ostrzeżeń. Oznacza to, że na ekranie wyświetlanych jest więcej ostrzeżeń niż w innych przypadkach. Ostrzeżenia zwracają uwagę na miejsca w programie, które

choć nie zawierają błędów składni, wzbudzają podejrzenia z punktu widzenia stylistycznego lub logicznego. W ten sposób w języku Ruby przekazuje się następujące informacje: „To, co zostało wykonane, jest poprawne składniowo, ale wygląda dziwnie. Czy na pewno coś takiego miało zostać osiągnięte?”. Nawet bez tej opcji interpreter języka Ruby generuje określone ostrzeżenia, ale mniej niż w przypadku pełnego trybu ostrzeżeń.

### WYKONYWANIE WPROWADZONEGO SKRYPTU Z KODEM (-E)

Opcja `-e` informuje interpreter o tym, że wiersz poleceń zawiera kod Ruby ujęty w znaki cudzysłowu, który powinien wykonać w podanej postaci, a nie jako kod umieszczony w pliku. Może to być przydatne przy szybkim wykonywaniu zadań skryptowych, w przypadku których wprowadzanie kodu do pliku i uruchamianie dla niego interpretera `ruby` może nie być warte zachodu.

Dla przykładu załóżmy, że chcesz zobaczyć na ekranie swoje imię zapisane odwrotnie. Oto jedno polecenie pozwalające szybko to osiągnąć z wykorzystaniem opcji `-e`:

```
$ ruby -e 'puts "Jan A. Nowak".reverse'
kawoN .A naJ
```

To, co zostało ujęte w znaki cudzysłowu, stanowi cały (choć krótki) program Ruby. Aby opcji `-e` przekazać program liczący więcej niż jeden wiersz, wewnątrz miniprogramu możesz użyć znaków podziału wiersza (przez naciśnięcie klawisza *Enter*):

```
$ ruby -e 'print "Wprowadź nazwę: "
puts gets.reverse'
Wprowadź nazwę: Jan A. Nowak
kawoN .A naJ
```

Wiersze możesz też oddzielić średnikami:

```
$ ruby -e 'print "Wprowadź nazwę: "; print gets.reverse'
```

> **UWAGA** Dlaczego występuje pusty wiersz między kodem programu i danymi wyjściowymi w przykładzie użycia metody `reverse` z dwoma wierszami? Ponieważ wiersz wprowadzany za pomocą klawiatury zakończony jest znakiem nowego wiersza, w przypadku odwracania danych wejściowych nowy łańcuch rozpoczyna się znakiem nowego wiersza! Interpreter języka Ruby traktuje bardzo dosłownie żądanie użytkownika dotyczące przetwarzania i wyświetlania danych.

### URUCHAMIANIE W TRYBIE WIERSZY (-L)

Opcja `-l` powoduje, że każdy łańcuch zwracany przez program umieszczany jest w swoim własnym wierszu, nawet jeśli standardowo nie ma to miejsca. Oznacza to zwykle, że wiersze wyświetlane przez polecenie `print`, a nie za pomocą polecenia `puts`, które nie są automatycznie zakończone znakiem nowego wiersza, po zastosowaniu tej opcji będą miały na końcu znak nowego wiersza.

Wykorzystaliśmy różnicę między poleceniami `print` i `puts`, aby zapewnić, że program do konwersji temperatury nie wstawi dodatkowych znaków nowego wiersza

w środku swoich danych wyjściowych (zajrzyj do punktu 1.1.5). Opcja `-l` umożliwia uzyskanie odwrotnego efektu: powoduje ona, że nawet dane wyświetlane przez polecenie `print` pojawiają się w osobnych wierszach. Oto różnica:

```
$ ruby c2f-2.rb
Oto wynik: 212.
$ ruby -l c2f-2.rb
Oto wynik:
212
.
```

W tym przypadku wynik uzyskany po zastosowaniu opcji `-l` jest dokładnie taki, jakiego nie żądano. Przykład ten ilustruje jednak efekt użycia opcji.

*Jeśli wiersz jest już zakończony znakiem nowego wiersza, zastosowanie dla niego opcji `-l` zakończy się dodaniem kolejnego takiego znaku.* Ogólnie rzecz biorąc, opcja ta nie jest powszechnie wykorzystywana lub spotykana. Wynika to głównie z możliwości zapewnienia przez polecenie `puts` działania powodującego w razie potrzeby dodanie nowego wiersza. Dobrze jednak wiedzieć o istnieniu opcji `-l`, a ponadto mieć możliwość rozpoznania efektów jej użycia.

#### WYMAGANIE PLIKU LUB ROZSZERZENIA Z NAZWĄ (-RNAZWA)

Opcja `-r` powoduje wywołanie metody `require` dla własnego argumentu. Polecenie `ruby -rscanf` będzie wymagać użycia pliku `scanf` podczas uruchamiania interpretera. W pojedynczym wierszu poleceń opcję `-r` możesz umieścić więcej niż raz.

#### URUCHAMIANIE W TRYBIE SZCZEGÓŁOWYCH INFORMACJI (-V, --VERBOSE)

Uruchamianie programu z opcją `-v` powoduje wykonanie dwóch działań: wyświetlenie informacji o używanej wersji języka Ruby, a następnie aktywowanie mechanizmu ostrzeżeń tego samego co w przypadku opcji `-w`. Najczęstszym zastosowaniem opcji `-v` jest znajdowanie numeru wersji języka Ruby:

```
$ ruby -v
ruby 2.1.0p0 (2013-12-25 revision 44422) [x86_64-darwin12.0]
```

W tym przypadku używana jest wersja 2.1.0 języka Ruby (poziom poprawek 0) opublikowana 25 grudnia 2013 r. i skompilowana dla komputera z procesorem i686 i systemem Mac OS X. Ponieważ nie określono żadnego programu lub kodu do uruchomienia, interpreter języka Ruby zakończy działanie od razu po wyświetleniu informacji o wersji.

#### WYŚWIETLANIE WERSJI JĘZYKA RUBY (--VERSION)

Opcja ta powoduje wyświetlenie przez interpreter języka Ruby łańcucha z informacjami o wersji, a następnie zakończenie pracy. W przypadku tej opcji nie jest wykonywany żaden kod, nawet jeśli podano kod lub nazwę pliku. Być może pamiętasz, że opcja `-v` wyświetla informacje o wersji, po czym uruchamia kod (jeśli go podano) w trybie szczegółowych informacji. Można powiedzieć, że opcja `-v` w ukryty sposób powiązana jest zarówno z *wersją*, jak i z *trybem szczegółowych informacji*, natomiast opcja `--version` dotyczy tylko *wersji*.

### WYŚWIETLANIE NIEKTÓRYCH INFORMACJI POMOCY (-H, --HELP)

Opcje te zapewniają zestawienie w postaci tabeli z wszystkimi dostępnymi opcjami wiersza poleceń, a także podsumowują ich przeznaczenie.

Oprócz użycia pojedynczych opcji w ramach jednego wywołania interpretera języka Ruby możesz też łączyć dwie lub większą liczbę opcji.

### ŁĄCZENIE OPCJI (-CW)

Zaprezentowano już kombinację opcji `-cw`, która powoduje sprawdzenie składni pliku bez uruchamiania go, a jednocześnie wyświetlenie ostrzeżeń:

```
$ ruby -cw nazwa_pliku
```

Inna kombinacja, z jaką często się spotkasz, złożona jest z opcji `-v` i `-e`. Umożliwiają one wyświetlenie używanej wersji języka Ruby, a następnie wykonanie kodu ujętego w znaki cudzysłowu. Kombinacja ta będzie obecna w wielu dyskusjach związanych z językiem Ruby, na listach adresowych i w różnych innych miejscach. Osoby korzystają z tych opcji, aby zademonstrować, jak ten sam kod może działać inaczej w różnych wersjach języka Ruby. Aby na przykład pokazać wyraźnie, że metoda łańcuchowa o nazwie `start_with?` nie była obecna w wersji 1.8.6 języka Ruby, ale istnieje w wersji 2.1.0, możesz uruchomić przykładowy program, używając najpierw pierwszej wersji języka, a następnie drugiej:

```
$ ruby-1.8.6-p399 -ve "puts 'abc'.start_with?('a')"
ruby 1.8.6 (2010-02-05 patchlevel 399) [x86_64-linux]
-e:1: undefined method `start_with?' for "abc":String (NoMethodError)  ←❶
$ ruby-2.1.0p0 -ve "puts 'abc'.start_with?('a')"
ruby 2.1.0p0 (2013-12-25 revision 44422) [x86_64-linux]
true                                                   ←❷
```

Oczywiście w systemie muszą być zainstalowane obie wersje języka Ruby. Komunikat `undefined method 'start_with?'` ❶ uzyskany przy pierwszym uruchomieniu (z wykorzystaniem wersji 1.8.6) oznacza, że podjęto próbę wykonania nazwanej operacji, która nie istnieje. Jednak w przypadku uruchomienia tego samego fragmentu kodu Ruby za pomocą wersji 2.1.0 języka Ruby operacja działa ❷: wyświetlana jest wartość `true`. Jest to wygodny sposób udostępniania informacji i formułowania pytań dotyczących zmian w działaniu języka Ruby między jego poszczególnymi wersjami.

W tym miejscu cofniemy się i przyjrzymy dokładniej interaktywnemu interpreterowi języka Ruby o nazwie irb. Być może zajrzałeś już do tego punktu, gdy wspomniano o nim na początku rozdziału. Jeśli nie, masz teraz możliwość uzyskania więcej informacji o tym wyjątkowo przydatnym narzędziu języka Ruby.

**Określanie opcji**

Do interpretera języka Ruby możesz przekazywać opcje osobno w następujący sposób:

```
$ ruby -c -w
```

lub

```
$ ruby -v -e "puts 'abc'.start with?('a')"
```

Częstą sytuacją jest jednak podawanie ich razem, tak jak to zaprezentowano w treści rozdziału.

### 1.4.2. Omówienie interaktywnego interpretera języka Ruby irb

Jak już wspomniano, irb to interaktywny interpreter języka Ruby. Oznacza to, że zamiast przetwarzania pliku zajmuje się tym, co zostanie wpisane w trakcie trwania sesji. irb to znakomite narzędzie służące do testowania kodu Ruby oraz nauki języka Ruby.

Aby rozpocząć sesję narzędzia irb, użyj polecenia `irb`. Spowoduje ono wyświetlenie swojej zachęty:

```
$ irb
2.1.0 :001 >
```

Jak już pokazano, możesz też zastosować opcję `--simple-prompt` w celu skrócenia danych wyjściowych narzędzia irb:

```
$ irb --simple-prompt
>>
```

Po uruchomieniu narzędzia irb możesz wprowadzać polecenia języka Ruby. Możliwe jest nawet uruchomienie wersji programu do konwersji stopni Celsjusza na stopnie Fahrenheita. Jak się okaże w omawianym przykładzie, pod względem działania narzędzie irb przypomina kieszonkowy kalkulator: wykonuje obliczenia dla wszystkiego, co zostanie wprowadzone, i wyświetla wynik. Nie musisz używać polecenia `print` lub `puts`:

```
>> 100 * 9 / 5 + 32
=> 212
```

Aby dowiedzieć się, z ilu minut składa się rok (jeśli nie masz pod ręką płyty CD z odpowiednim przebojem z musicalu *Rent*), wpisz następujące wyrażenie z operacją mnożenia:

```
>> 365 * 24 * 60
=> 525600
```

Oczywiście narzędzie irb będzie również przetwarzać wszystkie wprowadzone instrukcje języka Ruby. Aby na przykład przypisać do zmiennych liczby dni, godzin i minut, a następnie pomnożyć te zmienne, w narzędziu irb możesz to zrealizować w następujący sposób:

```
>> days = 365
=> 365
>> hours = 24
=> 24
>> minutes = 60
=> 60
>> days * hours * minutes
=> 525600
```

Ostatnie obliczenie jest tym, czego można oczekiwać. Spójrz jednak na pierwsze trzy wiersze z widocznych powyżej. Po wpisaniu przypisania `days = 365` w odpowiedzi narzędzie irb wyświetla liczbę `365`. Dlaczego tak jest?

Wyrażenie `days = 365` to wyrażenie przypisania: wartość `365` przypisujesz zmiennej o nazwie `days`. Głównym celem takiego wyrażenia jest przypisanie wartości zmiennej, aby możliwe było później użycie zmiennej. Wyrażenie przypisania (cały wiersz `days = 365`) ma jednak wartość. Wartość wyrażenia przypisania widoczna jest po jego

prawej stronie. Gdy narzędzie irb napotka dowolne wyrażenie, wyświetla jego wartość. A zatem gdy narzędzie to natrafi na wyrażenie `days = 365`, wyświetli wartość `365`. Choć może się to wydać przesadnym działaniem, stanowi nieodłączną część przetwarzania wyrażeń. Jest to identyczne działanie, które pozwala wpisać wyrażenie `2 + 2` w narzędziu irb i zobaczyć wynik bez potrzeby jawnego używania instrukcji `print`.

Nawet z wywołaniem metody `puts` powiązana jest wartość zwracana, czyli `nil`. Jeśli wpiszesz instrukcję `puts` w narzędziu irb, zostanie ona przez nie od razu wykonana, a ponadto zostanie wyświetlona wartość zwracana tej instrukcji:

```
$ irb --simple-prompt
>> puts "Witaj"
Witaj
=> nil
```

Istnieje możliwość ograniczenia ilości danych wyjściowych generowanych przez narzędzie irb; zapewnia to opcja `--noecho`. Działanie tej opcji jest następujące:

```
$ irb --simple-prompt --noecho
>> 2 + 2
>> puts "Cześć"
Cześć
```

Dzięki opcji `--noecho` wyrażenie sumy nie zwraca swojego wyniku. Polecenie `puts` jest wykonywane (dlatego pojawia się łańcuch `"Cześć"`), ale nie jest wyświetlana jego wartość zwracana (`nil`).

**Przerywanie działania narzędzia irb**

Możliwe jest zawieszenie wykonywania pętli w narzędziu irb lub wystąpienie braku odpowiedzi ze strony sesji (często oznacza to, że wpisano otwierający znak cudzysłowu, lecz nie znak domykający, lub coś innego w obrębie wierszy). To, w jaki sposób odzyskasz ponownie kontrolę nad wykonywaniem kodu, w pewnym stopniu zależy od systemu. W przypadku większości systemów kombinacja klawiszy *Ctrl+C* okaże się skuteczna. W innych systemach może być konieczne użycie kombinacji klawiszy *Ctrl+Z*. Najlepszym rozwiązaniem jest wykorzystanie bezpośrednio w odniesieniu do narzędzia irb wszelkich ogólnych informacji o operacji przerywania pracy programów, która może zostać wykonana w używanym systemie. Oczywiście, jeśli narzędzie irb naprawdę całkowicie się zawiesi, możesz skorzystać z narzędzi zarządzających procesami lub zadaniami, aby zakończyć proces narzędzia irb.

W celu zakończenia działania narzędzia irb w normalny sposób możesz wpisać polecenie `exit`. W wielu systemach sprawdzi się również kombinacja klawiszy *Ctrl+D*.

Sporadycznie narzędzie irb może zupełnie skapitulować (czyli zgłosić błąd krytyczny i samo zakończyć działanie). Przeważnie jednak narzędzie wychwytuje własne błędy i pozwala kontynuować pracę.

Po zaznajomieniu się ze sposobem wyświetlania przez narzędzie irb wartości wszystkiego, a także sposobem zamykania go, gdy okaże się to konieczne, stwierdzisz, że jest to niezmiernie przydatne narzędzie (i „zabawka”).

Kod źródłowy Ruby jest oznaczany w sposób zapewniający możliwość automatycznego generowania dokumentacji. `ri` i Rdoc to narzędzia niezbędne do interpretowania i wyświetlania takiej dokumentacji. Pora im się przyjrzeć.

### 1.4.3. Narzędzia ri i Rdoc

Narzędzia `ri` (*Ruby Index*) i RDoc (*Ruby Documentation*), które oryginalnie zostały stworzone przez Dave'a Thomasa, to blisko powiązana para programów służących do udostępniania dokumentacji dotyczącej programów Ruby. `ri` to narzędzie wiersza poleceń. System RDoc obejmuje narzędzie wiersza poleceń `rdoc`. `ri` i `rdoc` są programami autonomicznymi uruchamianymi z poziomu wiersza poleceń (możesz też skorzystać z rozwiązań zapewnianych w obrębie programów Ruby, choć nie będziemy w tym miejscu zajmować się tym zagadnieniem).

RDoc to system dokumentacji. Jeśli w plikach programu (napisanego w języku Ruby lub C) umieścisz komentarze w ustalonym formacie RDoc, narzędzie `rdoc` przeprowadzi skanowanie plików, wyodrębni komentarze, uporządkuje jej w inteligentny sposób (komentarze są indeksowane zgodnie z tym, czego dotyczą) i na ich podstawie utworzy ładnie sformatowaną dokumentację. Znaczniki systemu RDoc są obecne w wielu plikach źródłowych (utworzonych zarówno za pomocą języka Ruby, jak i języka C), w drzewie kodu źródłowego Ruby, a także w wielu plikach wchodzących w skład instalacji języka Ruby.

Narzędzie `ri` współgra z systemem RDoc: umożliwia ono wyświetlanie informacji wyodrębnionych i uporządkowanych przez ten system. Dokładniej rzecz biorąc, narzędzie `ri` jest skonfigurowane pod kątem wyświetlania informacji systemu RDoc z plików źródłowych Ruby (po dostosowaniu narzędzie może jednak służyć nie tylko do tego). Oznacza to, że w dowolnym systemie z pełną instalacją języka Ruby możesz uzyskać szczegółowe informacje o nim, używając zwykłego wywołania narzędzia `ri` z poziomu wiersza poleceń.

Oto przykład żądania informacji o metodzie `upcase` obiektów łańcuchowych:

```
$ ri String#upcase
```

Polecenie to zwraca następujące dane wyjściowe:

```
= String#upcase
(from ruby core)
------------------------------------------------------------------------------
  str.upcase   -> new_str
------------------------------------------------------------------------------
Returns a copy of str with all lowercase letters replaced with their
uppercase counterparts. The operation is locale insensitive---only characters
``a'' to ``z'' are affected. Note: case replacement is effective only in
ASCII region.
  "wItaJ".upcase   #=> "WITAJ"
```

Znak # znajdujący się między obiektem `String` i metodą `upcase` w poleceniu `ri` wskazuje, że w odróżnieniu od metody klasy szukana jest metoda instancji. W przypadku metody klasy należałoby użyć separatora `::` zamiast znaku #. W rozdziale 3. zajmiemy się odróżnianiem metody klasy i metody instancji. W tym miejscu ważne jest to, że z poziomu wiersza poleceń masz dostęp do dużej ilości dokumentacji.

**WSKAZÓWKA** Domyślnie polecenie `ri` przetwarza swoje dane wyjściowe za pomocą narzędzia stronicującego (np. narzędzie `more` w systemie Unix). Narzędzie to może wstrzymać pracę przy końcu danych wyjściowych, oczekując na

naciśnięcie przez użytkownika klawisza spacji lub innego klawisza w celu wyświetlenia następnego ekranu z informacjami lub całkowitego zakończenia działania narzędzia, jeśli wszystkie informacje zostały pokazane. To, co dokładnie ma zostać naciśnięte w tym przypadku, zależy od używanego systemu operacyjnego i narzędzia stronicującego. Równie dobrze może to być klawisz spacji, klawisz *Enter* lub *Esc* albo kombinacje klawiszy *Ctrl+C*, *Ctrl+D* i *Ctrl+Z*. Aby polecenie `ri` zapisało dane wyjściowe bez filtrowania ich za pomocą narzędzia stronicującego, możesz użyć opcji wiersza poleceń `-T` (`ri -T temat`).

`rake` to następne z narzędzi wiersza poleceń języka Ruby.

### 1.4.4. Narzędzie do zarządzania zadaniami rake

Jak sugeruje nazwa narzędzia `rake` (skrót od słów *Ruby make*), jest to narzędzie do zarządzania zadaniami inspirowane narzędziem `make`. Narzędzie `rake` zostało napisane przez Jima Weiricha. Podobnie jak narzędzie `make` wczytuje ono zadania zdefiniowane w pliku *Rakefile* i je wykonuje. Jednak w przeciwieństwie do narzędzia `make` do definiowania swoich zadań narzędzie `rake` używa składni języka Ruby.

Listing 1.5 prezentuje plik *Rakefile*. Jeśli zawartość listingu zapiszesz w pliku o nazwie *Rakefile*, możesz następnie z poziomu wiersza poleceń wykonać polecenie:

```
$ rake admin:clean_tmp
```

Polecenie `rake` wykonuje zadanie `clean_tmp` zdefiniowane w obrębie przestrzeni nazw `admin`.

**Listing 1.5. Plik Rakefile definiujący zadania clean_tmp w przestrzeni nazw admin**

```
namespace :admin do
  desc "Interaktywne usuwanie wszystkich plików w katalogu /tmp"
  task :clean_tmp do                          ← Deklaruje zadanie clean_tmp
    Dir["/tmp/*"].each do |f|                 ← ❶
      next unless File.file?(f)               ← ❷
      print "Czy usunąć plik #{f}? "          ← ❸
      answer = $stdin.gets
      case answer
      when /^y/
        File.unlink(f)                        ← ❹
      when /^q/
        break                                 ← ❺
      end
    end
  end
end
```

Zdefiniowane w tym przypadku zadanie narzędzia `rake` stosuje kilka technik języka Ruby, których jeszcze nie omawiano, ale podstawowy algorytm jest naprawdę prosty:

1. Wykonanie pętli dla każdej pozycji katalogu `/tmp` ❶.
2. Pominięcie bieżącej iteracji pętli, jeśli dana pozycja nie jest plikiem. Zauważ, że ukryte pliki nie są usuwane, ponieważ operacja wyświetlania zawartości katalogu nie uwzględnia ich ❷.

3. Wyświetlenie pytania o usunięcie pliku ❸.
4. Jeśli użytkownik wpisze literę `y` (lub dowolny łańcuch zaczynający się od niej), plik jest usuwany ❹.
5. Jeśli użytkownik wpisze literę `q`, następuje przerwanie pętli. Zadanie jest zatrzymywane ❺.

Podstawowa logika programistyczna bazuje na wykonywaniu pętli dla listy pozycji katalogu (przeczytaj treść ramki „Użycie instrukcji each do wykonywania pętli dla kolekcji"), a także na instrukcji `case`, czyli strukturze wykonywania warunkowego (obie techniki zostaną szczegółowo przedstawione w rozdziale 6.).

**Użycie instrukcji each do wykonywania pętli dla kolekcji**

Wyrażenie `Dir["/tmp/*"].each do |f|` to wywołanie metody `each` tablicy wszystkich nazw pozycji katalogu. Cały blok kodu zaczynający się od instrukcji `do` i zakończony instrukcją `end` (jest to ta, która ma identyczne wcięcie jak instrukcja `Dir`) jest jednokrotnie wykonywany dla każdego elementu tablicy. Każdorazowo w trakcie wykonywania kodu bieżący element jest wiązany z parametrem `f`. Takie znaczenie ma część `|f|` wyrażenia. W kolejnych rozdziałach metoda `each` pojawi się kilkakrotnie. Zostanie ona szczegółowo omówiona podczas prezentowania *iteratorów* (metody, które automatycznie dokonują przejścia przez kolekcje) w rozdziale 9.

Polecenie `desc` znajdujące się powyżej definicji zadania zapewnia jego opis. Jest to przydatne nie tylko podczas przeglądania pliku, ale też wtedy, gdy trzeba wyświetlić wszystkie zadania, które w dowolnym czasie mogą zostać wykonane przez narzędzie `rake`. Przejdź do katalogu zawierającego plik *Rakefile* (listing 1.5) i wykonaj następujące polecenie:

```
$ rake --tasks
```

Zostanie wyświetlony listing wszystkich zdefiniowanych zadań:

```
$ rake --tasks
(in /Users/ruby/hacking)
rake admin:clean_tmp # Interaktywne usuwanie wszystkich plików w katalogu /tmp
```

Na potrzeby przestrzeni nazw i zadań narzędzia `rake` możesz zastosować dowolne nazwy. Przestrzeń nazw nie jest nawet wymagana. Zadanie możesz zdefiniować w przestrzeni nazw najwyższego poziomu:

```
task :clean_tmp do
  # itp.
end
```

Wywołaj następnie zadanie za pomocą samej jego nazwy:

```
$ rake clean_tmp
```

Użycie dla zadań przestrzeni nazw jest jednak dobrym pomysłem, a zwłaszcza wtedy, gdy znacznie zwiększa się liczba definiowanych zadań. Przestrzeń nazw może mieć dowolną głębokość. Poprawna jest na przykład następująca struktura:

```
namespace :admin do
  namespace :clean do
```

```
    task :tmp do
      # itp.
    end
  end
end
```

Zdefiniowane powyżej zadanie jest wywoływane w następujący sposób:

```
$ rake admin:clean:tmp
```

Jak prezentuje przykład czyszczenia zawartości katalogu, zadania narzędzia `rake` nie muszą być ograniczone do działań powiązanych z programowaniem w języku Ruby. W przypadku tego narzędzia masz do dyspozycji wszystkie możliwości języka Ruby, z których możesz skorzystać podczas tworzenia dowolnych wymaganych zadań.

Następne omawiane narzędzie to polecenie `gem`, które bardzo upraszcza instalację zewnętrznych pakietów języka Ruby.

### 1.4.5. Instalowanie pakietów za pomocą polecenia gem

Biblioteka RubyGems i kolekcja narzędzi obejmują rozwiązania ułatwiające tworzenie pakietów oraz instalowanie bibliotek i aplikacji języka Ruby. Nie będziemy tutaj omawiać tworzenia pakietu za pomocą narzędzia `gem`, ale przyjrzymy się instalacji takiego pakietu i korzystaniu z niego.

Instalowanie pakietu przy użyciu narzędzia `gem` może, i zwykle tak jest, sprowadzać się do wykonania prostego polecenia `install`:

```
$ gem install prawn
```

Takie polecenie zwraca dane wyjściowe podobne do przedstawionych poniżej (zależnie od tego, jakie zostały zainstalowane pakiety narzędzia `gem` i jakie zależności muszą zostać spełnione przy instalacji nowych pakietów):

```
Fetching: Ascii85-1.0.2.gem (100%)
Fetching: ruby-rc4-0.1.5.gem (100%)
Fetching: hashery-2.1.0.gem (100%)
Fetching: ttfunk-1.0.3.gem (100%)
Fetching: afm-0.2.0.gem (100%)
Fetching: pdf-reader-1.3.3.gem (100%)
Fetching: prawn-0.12.0.gem (100%)
Successfully installed Ascii85-1.0.2
Successfully installed ruby-rc4-0.1.5
Successfully installed hashery-2.1.0
Successfully installed ttfunk-1.0.3
Successfully installed afm-0.2.0
Successfully installed pdf-reader-1.3.3
Successfully installed prawn-0.12.0
7 gems installed
```

Po tego rodzaju raportach statusu następuje kilka wierszy wskazujących, że instalowana jest dokumentacja narzędzi `ri` i RDoc dla różnych pakietów narzędzia `gem` (instalacja dokumentacji obejmuje przetwarzanie plików źródłowych pakietów narzędzia `gem` za pośrednictwem systemu RDoc, dlatego należy być cierpliwym; często jest to najdłużej trwająca faza instalacji pakietów narzędzia `gem`).

W trakcie procesu instalacji pakietów narzędzia `gem` w razie potrzeby pobiera ono pliki pakietów z witryny rubygems.org (*http://www.rubygems.org/*). Pliki te, z rozszerzeniem *.gem*, są zapisywane w podkatalogu pamięci podręcznej katalogu pakietów narzędzia `gem`. Możliwe jest też zainstalowanie pakietu z pliku *.gem* znajdującego się lokalnie na dysku twardym lub innym nośniku. Podaj instalatorowi nazwę pliku:

```
$ gem install /home/me/mygems/ruport-1.4.0.gem
```

W przypadku podania nazwy pakietu narzędzia `gem` bez rozszerzenia (np. *ruport*) szuka ono pliku pakietu w bieżącym katalogu, a także w lokalnej pamięci podręcznej utrzymywanej przez system biblioteki RubyGems. W instalacjach lokalnych nadal ma miejsce zdalne wyszukiwanie zależności, chyba że w poleceniu `gem` podano opcję wiersza poleceń `-l` (lokalne), która ogranicza wszystkie operacje do domeny lokalnej. Aby zostały zainstalowane wyłącznie zdalne pakiety narzędzia `gem`, w tym zależności, możesz użyć opcji `-r` (zdalne). W celu odinstalowania pakietu narzędzia `gem` zastosuj polecenie `gem uninstall` *`nazwa_pakietu_gem`*.

Po zainstalowaniu pakietu możesz go użyć za pomocą metody `require`.

### ŁADOWANIE I STOSOWANIE PAKIETÓW NARZĘDZIA GEM

Jeśli pakietów narzędzia `gem` nie ma w początkowej ścieżce ładowania (`$:`), możliwe jest jednak zażądanie ich za pomocą metody `require` i załadowanie. Oto przykład użycia tej metody dla pakietu `hoe` (narzędzie ułatwiające tworzenie własnych pakietów narzędzia `gem`) przy założeniu, że zainstalowano ten pakiet:

```
>> require "hoe"
=> true
```

Na tym etapie w ścieżce ładowania pojawi się odpowiedni katalog `hoe`, co możesz sprawdzić, wyświetlając wartość ścieżki `$:` i wynik polecenia `grep` (umożliwia wybór za pomocą wzorca dopasowywania) użytego dla wzorca `"hoe"`:

```
>> puts $:.grep(/hoe/)
/Users/dblack/.rvm/gems/ruby-2.1.0/gems/hoe-3.8.1/lib
```

Jeśli dla określonej biblioteki zainstalowano więcej niż jeden pakiet narzędzia `gem`, a ponadto ma zostać wymuszone użycie pakietu innego niż najnowszy, możesz skorzystać z metody `gem` (zauważ, że nie jest ona tym samym co narzędzie wiersza poleceń o nazwie `gem`). Oto przykład prezentujący, w jaki sposób możesz wymusić zastosowanie mniej aktualnej wersji pakietu hoe:

```
>> gem "hoe", "3.8.0"
=> true
>> puts $:.grep(/hoe/)
/Users/dblack/.rvm/gems/ruby-2.1.0/gems/hoe-3.8.0/lib
```

**W przypadku użycia metody `gem` nie ma potrzeby stosowania metody `require`**

Oczywiście przeważnie wskazane będzie użycie najnowszych wersji pakietów narzędzia `gem`. System obsługujący te pakiety zapewnia jednak w razie potrzeby narzędzia służące do dostosowywania sposobu korzystania z pakietów.

Po wspomnieniu o bibliotece RubyGems możemy zakończyć bieżące omówienie związane z katalogiem */bin*. W dalszej kolejności zajmiemy się bliżej podstawowymi elementami języka.

## 1.5. Podsumowanie

W rozdziale dokonaliśmy przeglądu kilku ważnych i fundamentalnych zagadnień związanych z językiem Ruby. Oto one:

- Różnica między terminami Ruby (język) i ruby (interpreter języka Ruby).
- Typografia związana ze zmiennymi języka Ruby (wszystkie z nich będą znów prezentowane i dokładniej analizowane).
- Podstawowe operatory i wbudowane konstrukcje języka Ruby.
- Zapisywanie, przechowywanie i uruchamianie pliku programu Ruby.
- Dane wprowadzane za pomocą klawiatury i dane wyjściowe na ekranie.
- Modyfikowanie bibliotek języka Ruby za pomocą metod `require` i `load`.
- Anatomia instalacji języka Ruby.
- Narzędzia wiersza poleceń dołączone do języka Ruby.

Dysponujesz teraz dobrym planem prezentującym sposób działania języka Ruby oraz narzędzia zapewniane w jego środowisku programowania. Zaznajomiłeś się z kilkoma ważnymi technikami związanymi z językiem Ruby i sprawdziłeś je w praktyce. Jesteś przygotowany do systematycznego poznawania języka Ruby.

# 2 Obiekty, metody i zmienne lokalne

**W tym rozdziale:**

- Obiekty i obiektowość.
- Porównanie wbudowanych możliwości obiektów z nabytymi.
- Parametr metody, argument oraz składnia wywoływania.
- Przypisywanie zmiennych lokalnych i ich zastosowanie.
- Odwołania do obiektów.

W rozdziale rozpoczniemy eksplorowanie szczegółów języka programowania Ruby. Przede wszystkim przyjrzymy się najpierw pojęciu obiektu, z którym związany będzie niemal każdy wiersz kodu Ruby. Ogólnie rzecz biorąc, operacje wykonywane w powiązaniu z obiektami sprowadzają się do wysyłania im komunikatów. Większość z nich odpowiada nazwom metod, których wykonania żądasz od obiektu. Dość dokładnie przyjrzymy się połączonym procesom wysyłania komunikatów i wywoływania metod.

Obiekty języka Ruby często (być może najczęściej) są obsługiwane za pośrednictwem reprezentujących je zmiennych. W rozdziale w większym stopniu niż do tej pory zagłębimy się w techniczne szczegóły związane z naturą i działaniem zmiennych w języku Ruby. Lektura rozdziału pozwoli znaleźć punkt zaczepienia w krainie obiektów języka Ruby i związanych z nimi modyfikacji.

Jak zawsze przykładowy kod możesz wpisać w oknie narzędzia irb i/lub zapisać go w pliku, który następnie może zostać uruchomiony za pomocą interpretera języka Ruby. Niezależnie od wybranej metody postępowania warto tak na wszelki wypadek pozostawić otwartą sesję narzędzia irb.

## 2.1. Komunikowanie się z obiektami

W dowolnym programie Ruby większość elementów projektu, logiki i działań związana jest z obiektami. W trakcie pisania programów Ruby podstawowe operacje to tworzenie obiektów, zapewnianie im możliwości i żądanie od nich wykonywania działań. Obiekty pełnią rolę klamki w świecie programu. Gdy chcesz zrealizować jakiekolwiek działanie, czy to obliczenie, operację na danych wyjściowych, czy porównanie danych, żądasz tego od obiektu. Zamiast pytać na poziomie abstrakcyjnym, czy *a* jest równe *b*, pytasz, czy *a* uważa się samo za równe *b*. Aby dowiedzieć się, czy dany student uczestniczy w zajęciach danego nauczyciela, zadasz studentowi następujące pytanie: „Czy jesteś studentem tego nauczyciela?". To, jak dokładnie przebiega tego rodzaju odpytywanie, zależy od szczegółów projektu programu. Generalnie jednak tworzenie programu Ruby to głównie kwestia takiej inżynierii obiektów, aby każdy z nich pełnił wyraźną rolę, a ponadto mógł realizować powiązane z nią działania.

### 2.1.1. Język Ruby i obiektowość

W przypadku języka Ruby obowiązuje idea polegająca na przetwarzaniu danych za pośrednictwem obiektów z wykorzystaniem zasady projektowej języka programowania określanej mianem *obiektowości*. Wiele bardzo popularnych języków jest zorientowanych obiektowo (np. Java, C++ i Python, a także Ruby). Niektóre języki, które nie są w pełni zorientowane obiektowo, oferują narzędzia przeznaczone do tworzenia kodu obiektowego (np. język Perl opisany w książce Damiana Conwaya, *Object-Oriented Perl* [wydawnictwo Manning, 1999 r.]). W ramach programowania zorientowanego obiektowo wykonywane są obliczenia, modyfikacje danych oraz operacje na danych wejściowych/wyjściowych przez tworzenie obiektów i żądanie od nich zrealizowania działań i udostępnienia informacji.

W większości języków zorientowanych obiektowo, w tym w języku Ruby, każdy obiekt jest przykładem lub instancją konkretnej klasy. Działanie poszczególnych obiektów jest określane przynajmniej w pewnym zakresie przez definicje metod obecne w klasie obiektu. W rozdziale 3. obszernie zostaną omówione klasy. W tym miejscu skoncentrujemy się bezpośrednio na obiektach.

Projektowanie oprogramowania obiektowego sprowadza się w dużej mierze do określenia tego, jak mają wyglądać obiekty. Co powinny realizować? Jak będą ze sobą prowadzić interakcję? Ile obiektów każdego typu (np. wielu studentów i jeden kierownik dziekanatu) powinno istnieć? Jak się okaże, język Ruby zapewnia kompletny zestaw narzędzi służących do nadawania nazw, tworzenia, adresowania i modyfikowania obiektów, a za ich pośrednictwem danych, które są przez nie przetwarzane.

**Rzeczywisty świat**

Termin *rzeczywisty świat* pojawia się w wielu dyskusjach związanych z programowaniem. Można debatować (i *ma* to miejsce) nad tym, czy ten lub inny język programowania, a nawet ten lub inny rodzaj języka, w większym stopniu w porównaniu z innymi oddaje rzeczywisty świat. Wiele zależy od tego, jak postrzegasz świat. Czy uważasz go za zbiorowisko ludzi ze swoimi sprawami, gdzie każda osoba ma zadania do wykonania i oczekuje na zażądanie tego przez kogoś? Jeśli tak, możesz uznać, że języki zorientowane obiektowo najlepiej modelują świat. Czy traktujesz życie jako serię czynności do wykonania na liście, które zostaną zrealizowane po kolei? Jeśli tak, możesz stwierdzić, że rygorystyczny, proceduralny język programowania ma więcej wspólnego z właściwościami rzeczywistego świata.

Podsumowując, nie ma jednej odpowiedzi na pytanie, czym jest rzeczywisty świat. Oznacza to, że nie istnieje odpowiedź na pytanie dotyczące tego, czym dla języka programowania jest modelowanie rzeczywistego świata. Wbrew temu, co mogłoby się wydawać, nie jest to taka ważna kwestia. Świat konstruowany w programie komputerowym stanowi w zasadzie wyimaginowany świat. Ograniczanie się przez sprawianie wrażenia, że jest inaczej, może być przesadnym działaniem.

### 2.1.2. Tworzenie obiektu ogólnego

Przede wszystkim zagadnienie programowania zorientowanego obiektowo przejawia tendencję do bycia zarówno prostym (piszesz programy „zawierające" książki i butelki, samochody i domy, a ponadto aranżujesz swego rodzaju konwersację między tymi obiektami), jak i abstrakcyjnym (*Obiekt? Co to oznacza? Co mam wpisać w pliku mojego programu, aby utworzyć obiekt „domu"?*). Programowanie zorientowane obiektowo oferuje element prostoty: umożliwia wykorzystanie obiektów, jednostek, ról i zachowań jako źródła metody projektowania programów. Może to być pomocne. Jednocześnie do tworzenia i używania obiektów w swoich programach konieczne będzie uzyskanie informacji na temat tego, jak to przebiega w danym języku.

**UWAGA** Zależnie od posiadanego doświadczenia i oczekiwań możesz zastanawiać się, dlaczego nie zaczynamy omawiania obiektów od bliższego przyjrzenia się klasom, a nie obiektom. Klasy mają duże znaczenie w języku Ruby. Zapewniają sposób tworzenia pakietów zachowań i etykietowania ich (mogą istnieć klasy `Osoba`, `Zadanie` itp.), a także łatwego generowania wielu obiektów o podobnych zachowaniach. Jednak pod tym względem język Ruby różni się od innych języków obiektowych. Rzeczywiste działanie dotyczy poszczególnych obiektów: każdy obiekt ma potencjał „uczenia się" zachowań (metod), które nie zostały poznane przez jego klasę. Nie ma wątpliwości co do tego, że pojęcie klasy dopasowane jest do pojęcia obiektu. Okazuje się, że w języku Ruby klasa sama jest obiektem! Więcej na ten temat będzie napisane później, ale takie są podstawowe powody, dla których zaczynamy od obiektów.

Objaśnienie programowania zorientowanego obiektowo w przypadku konkretnego języka może ułatwić zrozumienie abstrakcyjnych elementów. W dalszej kolejności zajmiemy się przykładowym kodem Ruby. Zostanie utworzony nowy obiekt. Nie będzie on reprezentować ani modelować niczego konkretnego (np. dom, książka lub nauczyciel). Będzie to obiekt ogólny:

```
obj = Object.new
```

Powyższy kod tworzy obiekt i zmienną, za pomocą której możesz odwoływać się do obiektu.

Wszystkie obiekty języka Ruby są tworzone z określonymi możliwościami wbudowanymi. Są to metody, które obiekty potrafią wykonać, ponieważ są obiektami języka Ruby. Choć te możliwości są istotne, nie są aż tak ekscytujące, dlatego zajmiemy się nimi później. Bardziej ciekawe jest to, co ma miejsce po „nauczeniu" obiektu, jak ma realizować żądane przez nas działania.

### DEFINIOWANIE ZACHOWANIA OBIEKTU

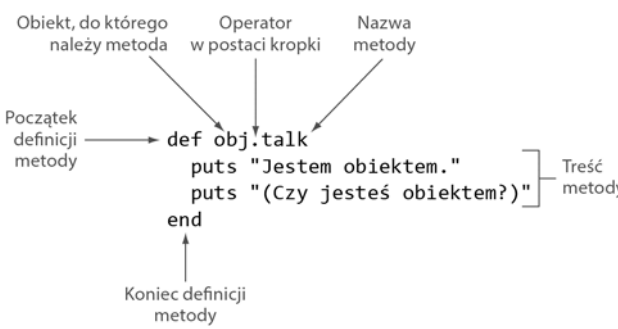
Załóżmy, że utworzono obiekt, który ma zrobić coś interesującego: chcesz, aby się komunikował. W tym celu musisz od obiektu tego zażądać. Zanim to jednak zrobisz, musisz „nauczyć" obiekt komunikowania się.

Dokładnie rzecz biorąc, z technicznego punktu widzenia niezbędne jest zdefiniowanie *metody* dla obiektu. W tym celu stosowany jest specjalny termin w postaci *słowa kluczowego* o nazwie `def`.

Oto przykład definiowania metody `talk` dla obiektu `obj`:

```
def obj.talk
  puts "Jestem obiektem."
  puts "(Czy jesteś obiektem?)"
end
```

Na rysunku 2.1 dokonano analizy powyższego fragmentu kodu.




**Rysunek 2.1.** Anatomia definicji metody

Obiekt `obj` potrafi teraz komunikować się, a ponadto możesz tego od niego zażądać.

### WYSYŁANIE KOMUNIKATÓW DO OBIEKTÓW

Aby zażądać komunikowania się od obiektu `obj`, użyj zamieszczonej w rozdziale 1. składni wywoływania metody lub wysyłania komunikatu:

```
obj.talk
```

Obiekt komunikuje się:

```
Jestem obiektem.
(Czy jesteś obiektem?)
```

Obiekt `obj` „rozumie” komunikat `talk` lub odpowiada na niego. Obiekt ma odpowiedzieć na komunikat, jeśli zawiera zdefiniowaną metodę, której nazwa odpowiada komunikatowi.

Oto kilka kwestii do rozważenia związanych ze składnią wysyłania komunikatu, która zawiera znak kropki:

- Znak kropki to operator wysyłania komunikatu. Komunikat po prawej stronie jest wysyłany do obiektu (lub *odbiorcy*; taki termin często jest używany w tym przypadku) po lewej stronie.
- Odbiorca może być, i często jest, reprezentowany przez zmienną, która go zastępuje. Odbiorca może być jednak też konstrukcją w postaci obiektu literału (na przykład łańcuch w znakach cudzysłowu).
- W praktyce wysyłany komunikat prawie zawsze jest nazwą metody (tak jak zdefiniowana wcześniej metoda `talk`). Obiekt zawsze próbuje działać przy założeniu, że komunikat jest nazwą metody. Jeśli nie istnieje metoda o podanej nazwie, podejmowane są działania w ramach obsługi błędów.

Semantyka wywołań metody pozwala na więcej niż w przypadku względnie jednowymiarowego przykładu z metodą `talk`. Dotyczy to zwłaszcza sytuacji, gdy rozpoczynasz wywoływanie metod z argumentami.

### 2.1.3. Metody pobierające argumenty

Metody w języku Ruby w dużym stopniu przypominają funkcje matematyczne: po przekazaniu danych wejściowych rozpoczyna się przetwarzanie, a następnie zwracany jest wynik. Aby przekazać dane wejściowe metodzie języka Ruby, wywołaj ją z co najmniej jednym argumentem.

W definicji metody po jej nazwie wskazujesz argumenty w postaci listy zmiennych umieszczonych w nawiasach okrągłych (argumenty mogą być wymagane lub opcjonalne; w tym miejscu przyjrzymy się argumentom obligatoryjnym, a w dalszej części rozdziału argumentom opcjonalnym). Podczas wywoływania metody w wywołaniu podajesz wartości odpowiadające tym zmiennym. Dokładniej rzecz biorąc, zmienne wyszczególnione w definicji metody są *formalnymi parametrami* metody, a wartości przekazywane metodzie podczas jej wywoływania to odpowiadające im *argumenty* (termin *argumenty* często jest używany nieformalnie w odniesieniu do parametrów metody, a także do argumentów wywołania metody, ale warto wiedzieć, że istnieje takie techniczne odróżnienie).

Załóżmy, że obiekt ma pełnić rolę konwertera stopni Celsjusza na stopnie Fahrenheita. Definiując następującą metodę, możesz dowiedzieć się, jak przeprowadzić taką konwersję:

```
def obj.c2f(c)
  c * 9.0 / 5 + 32
end
```

Tym razem w formule konwersji wartość 9 przybrała postać 9.0. Spowoduje to wymuszenie wyniku jako liczby zmiennoprzecinkowej, która jest dokładniejsza od liczby całkowitej. Metoda `obj.c2f` zawiera jeden formalny parametr, który oznacza, że pobiera ona jeden argument. W momencie wywoływania metody podajesz argument:

```
puts obj.c2f(100)
```

Oto wynik:

```
212.0
```

Jak widać, występuje bezpośrednie powiązanie między składnią listy parametrów w definicji metody i składnią listy argumentów podczas wywoływania metody. W obu przypadkach nawiasy okrągłe są opcjonalne. Możesz zastosować następujące wiersze kodu:

```
def obj.c2f c
```

i

```
obj.c2f 100
```

Nie są one jednak zawsze opcjonalne, a zwłaszcza wtedy, gdy łączonych jest ze sobą wiele wywołań metody. Z tego powodu warto bardziej korzystać z nawiasów niż z nich rezygnować. Wyjątek możesz zrobić odnośnie do typowych lub konwencjonalnych sytuacji, w których nawiasy okrągłe są zwykle pomijane (np. wywołania metody `puts`). W razie wątpliwości użyj jednak nawiasów.

Na drugim końcu procesu każde wywołanie metody *zwraca* wartość.

### 2.1.4. Wartość zwracana metody

Kod Ruby złożony jest z wyrażeń, dla których określana jest konkretna wartość. W tabeli 2.1 przedstawiono kilka przykładów wyrażeń oraz ich wartości (wraz z komentarzami objaśniającymi).

**Tabela 2.1.** Przykłady wyrażeń języka Ruby i wartości, które są dla nich określane

| Wyrażenie | Wartość | Komentarze |
|---|---|---|
| `2 + 2` | `4` | Dla wyrażeń arytmetycznych są określane wyniki. |
| `"Witaj"` | `"Witaj"` | W przypadku zwykłego łańcucha literału (w znakach cudzysłowu) wartością wynikową jest on sam. |
| `"Witaj," + " przyjacielu"` | `"Witaj, przyjacielu"` | Łańcuchy mogą być „dodawane" do siebie (operacja konkatenacji) za pomocą znaku plus. |
| `c = 100` | `100` | Przy przypisywaniu do zmiennej dla całego przypisania określana jest przypisana wartość. |
| `c * 9/5 + 32` | `212` | Zastosowanie mają zwykłe reguły pierwszeństwa: operacje mnożenia i dzielenia są ważniejsze niż dodawanie, dlatego wykonywane są w pierwszej kolejności. |
| `obj.c2f(100)` | `212` | Wywołanie metody jest wyrażeniem. |

Przyjrzyj się ostatniej pozycji w tabeli 2.1; jest to wywołanie metody `obj.c2f`. Każde wywołanie metody to wyrażenie. Przy wywoływaniu metody dla wywołania określany jest wynik. Taki wynik wywołania metody jest *wartością zwracaną* metody.

Wartość zwracana dowolnej metody jest taka sama jak wartość ostatniego wyrażenia obliczanego podczas wykonywania metody. W przypadku metody konwersji temperatur ostatnim obliczanym wyrażeniem jest jedynie wiersz, który jest treścią metody:

```
c * 9.0 / 5 + 32
```

Oznacza to, że wynik tego obliczenia zapewnia wartość zwracaną metody.

Język Ruby oferuje słowo kluczowe `return`, które gwarantuje jawność wartości zwracanych. Choć użycie tego słowa jest zwykle opcjonalne, wielu programistów lubi z niego korzystać, ponieważ dzięki temu zapewniają jawność tego, co w przeciwnym razie będzie niejawne:

```
def obj.c2f(c)
  return c * 9.0 / 5 + 32
end
```

Jest to odpowiednik wcześniejszej wersji metody, ale zapewnia więcej informacji o tym, co ma miejsce. Jednak ta wersja definicji jest bardziej rozwlekła. W ramach określania własnego stylu musisz zdecydować, czy chcesz korzystać ze słowa kluczowego `return`. Przy zwracaniu wielu wartości musi ono zostać zastosowane. Takie wartości będą automatycznie opakowywane w tablicy: `return a,b,c`, zamiast tylko `a,b,c` (choć możliwe jest też zwracanie wielu wartości w tablicy jawnej, takiej jak `[a,b,c]`, bez słowa kluczowego `return`). Konieczne jest również użycie tego słowa kluczowego, gdy wartość ma zostać zwrócona gdzieś w obrębie metody. Jednak niezależnie od tego, czy słowo kluczowe `return` jest wykorzystywane, każde wywołanie metody spowoduje zwrócenie *czegoś*. Nawet wywołanie metody bez treści, która zawiera tylko instrukcje `def` i `end`, zakończy się zwróceniem obiektu `nil`.

Na tym etapie obiekt działa zgodnie z wymaganiami: nasłuchuje komunikatów i wykonuje dla nich działania. Jest to dobra ilustracja sposobu działania języka Ruby, lecz niekompletna. Zaczęliśmy od obiektu ogólnego, a następnie sprawiliśmy, że mógł się komunikować i konwertować temperatury. Pozwoliło to zaznajomić się z mechanizmem definiowania i wywoływania metod, ale zakończyło się uzyskaniem raczej dziwnego obiektu. Przyjrzyjmy się obiektowi modelującemu coś, co uwzględnia w trochę większym stopniu strukturę. Utworzymy obiekt ogólny, dla którego będzie zrozumiałe działanie i logika biznesowa biletu na wydarzenie.

## 2.2. Tworzenie obiektu: działanie biletu

Bilet to znajomy obiekt ze znanym zestawem właściwości i działań. Zorientujmy się ogólnie, jakich działań oczekujemy od obiektu języka Ruby przypominającego bilet, a także jakie informacje o nim samym mają być dla niego dostępne.

### 2.2.1. Obiekt biletu — przede wszystkim działanie

Obiekt biletu powinien mieć możliwość zapewnienia danych dotyczących jego samego. Obiekt ten powinien odpowiadać na żądania udostępnienia informacji o wydarzeniu, z którym powiązany jest bilet. Informacje obejmują datę, miejsce i nazwę wydarzenia, wykonawcę, miejsce i cenę.

Po uzyskaniu żądania obiekt biletu zapewni następujące informacje, bazując na wymyślonym publicznym wieczorze literackim Marka Twaina z 1903 roku:

```
01/02/03
Sala Miejska
Wieczór literacki autora
Mark Twain
Balkon drugi, rząd J, miejsce 12
5,50 zł
```

Celem jest utworzenie obiektu, z którego z łatwością możemy uzyskać wszystkie te informacje.

**TWORZENIE OBIEKTU BILETU**

Obiekt ogólny będzie pełnić rolę podstawy biletu:

```
ticket = Object.new
```

Gdy obiekt `ticket` już istnieje, możemy rozpocząć wyposażanie go we właściwości i dane przez definiowanie metod, z których każda zwraca odpowiednią wartość:

```
def ticket.date
  "01/02/03"
end
def ticket.venue
  "Sala Miejska"
end
def ticket.event
  "Wieczór literacki autora"
end
def ticket.performer
  "Mark Twain"
end
def ticket.seat
  "Balkon drugi, rząd J, miejsce 12"
end
def ticket.price      ←❶
  5,50
end
```

Większość zdefiniowanych tutaj metod zwraca wartości łańcuchowe. Jest to od razu widoczne: metody te zwracają wartość wewnątrz znaków cudzysłowu. Metoda `price` zwraca liczbę zmiennoprzecinkową ❶. Gdy jeszcze obiekt `ticket` nie ma wielu informacji o sobie, zażądajmy od niego ich udostępnienia.

### *2.2.2. Odpytywanie obiektu biletu*

Zamiast generować nieprzetworzoną listę pozycji, utwórzmy przyjazne dla użytkownika podsumowanie szczegółów biletu. Użycie metod `print` i `puts` może być pomocne w zapewnieniu mniej lub bardziej narracyjnej formy informacji:

```
print "Bilet na wydarzenie: "
print ticket.event + ". Miejsce: "
print ticket.venue + ". Data: "
puts ticket.date + "."
```

**Wyświetlenie informacji o wydarzeniu**

```
print "Wykonawca: "
puts ticket.performer + "."
print "Miejsce: "
print ticket.seat + ". "
print "Cena: "
puts "%,2f." % ticket.price
```

**Wyświetlenie informacji o wykonawcy**

**Wyświetlenie informacji o miejscu**

**Wyświetlenie liczby zmiennoprzecinkowej z dokładnością do dwóch miejsc po przecinku**

Zapisz cały kod, począwszy od wiersza `ticket = Object.new`, w pliku o nazwie *ticket.rb* i uruchom go. Zostaną wyświetlone następujące dane wyjściowe:

```
Bilet na wydarzenie: Wieczór literacki autora. Miejsce: Sala Miejska. Data: 01/02/03.
Wykonawca: Mark Twain.
Miejsce: Balkon drugi, rząd J, miejsce 12. Cena: 5,5.
```

Kod tego przykładu składa się z serii wywołań zdefiniowanych wcześniej metod: `ticket.event`, `ticket.venue` itd. Kod wyświetlający osadza te wywołania, czyli innymi słowy, wartości zwracane tych metod (`"Wieczór literacki autora"`, `"Sala Miejska"` itd.) w serii poleceń danych wyjściowych, a ponadto dodaje łączniki (`". Miejsce: "`, `". Data: "` itd.), aby zapewnić dobrą czytelność i ładny wygląd tekstu.

Choć przykład biletu na wieczór literacki Twaina jest prosty, uwzględnia kilka ważnych procedur i zasad języka Ruby. Najważniejsza wynikająca z przykładu informacja jest taka, że *„wiedza" niezbędna do wykonania przez program czegokolwiek przydatnego znajduje się w obiekcie*. Obiekt biletu ma taką „wiedzę". W celu uzyskania do niej dostępu należy skierować do obiektu biletu odpowiednie żądanie za pośrednictwem wywołań metody. Nic bardziej niż to nie stanowi sedna programowania w języku Ruby. Sprowadza się to do żądania od obiektów wykonania działań i zwrócenia danych.

Kod obiektu biletu działa i zapewnia wartościowe informacje, ale jest rozwlekły. Język Ruby ma reputację języka wysokiego poziomu o dużych możliwościach. Powinno być możliwe zrealizowanie wielu działań z wykorzystaniem stosunkowo niewielkiej ilości kodu. Jednak do wygenerowania trzech wierszy danych wyjściowych przykład obiektu biletu wymaga dziesięciu wierszy z instrukcjami `print` i `puts`.

Poprawmy trochę tę proporcję.

### 2.2.3. Skracanie kodu obiektu biletu za pomocą interpolacji łańcuchów

Jedną z najbardziej przydatnych technik programowania dostępnych w języku Ruby jest *interpolacja łańcuchów*. Operator interpolacji łańcuchów zapewnia możliwość umieszczenia czegokolwiek w łańcuchu (na przykład zmiennej lub wartości zwracanej metody). Może to pozwolić zaoszczędzić wiele przełączeń między instrukcjami `print` i `puts`.

Co więcej, łańcuchy mogą być łączone za pomocą znaku plus (+). Poniżej zaprezentowano kod wyświetlający, w którym zastosowano interpolację łańcuchów do wstawienia wartości wyrażeń do łańcucha. Ponadto użyto operacji dodawania łańcuchów w celu skonsolidowania wielu wywołań metody `puts` w jedno.

```
puts "Bilet na wydarzenie: #{ticket.event}. Miejsce: #{ticket.venue}." +
  "Wykonawca: #{ticket.performer}." +
  "Miejsce: #{ticket.seat}. " +
  "Cena: $#{"%,2f." % ticket.price}"
```

Wszystko, co znajduje się w obrębie operatora interpolacji `#{...}`, obliczane jest osobno. Wyniki obliczenia są wstawiane do łańcucha. Po wykonaniu wierszy powyższego kodu na ekranie nie pojawi się operator `#{...}`. Zamiast niego widoczne będą wyniki obliczenia lub określania wartości tego, co znajdowało się w nawiasach klamrowych. Interpolacja pomogła wyeliminować sześć spośród dziesięciu wierszy kodu, a także sprawiła, że kod o wiele bardziej przypomina ostateczny format danych wyjściowych niż coś, co działa, lecz nie zapewnia zbyt wielu danych wizualnych.

Do tej pory żądano od obiektu biletu informacji w postaci łańcuchów i liczb. Bilety zawierają też trochę informacji o sobie w formie danych boolowskich (prawda/fałsz).

### 2.2.4. Dostępność biletu: wyrażanie stanu boolowskiego w metodzie

Za pomocą informacji boolowskich możesz rozpatrywać, czy bilet został sprzedany, czy nadal jest dostępny. Oto jeden ze sposobów zapewniania obiektowi biletu informacji o jego własnym statusie dostępności:

```
def ticket.availability_status
  "sprzedany"
end
```

Inny sposób polega na skierowaniu do obiektu biletu żądania dotyczącego jego dostępności i spowodowaniu przekazania przez obiekt wartości `true` (prawda) lub `false` (fałsz):

```
def ticket.available?
  false
end
```

Podobnie jak `true`, tak i `false` to specjalny termin w języku Ruby. `true` i `false` to obiekty. W języku tym obiekty te służą do reprezentowania między innymi wyników operacji porównywania (np. `x > y`). Obiektów możesz używać do reprezentowania prawdy i fałszu w tworzonych przez siebie metodach. Być może zauważyłeś, że nazwa metody `available?` zakończona jest pytajnikiem. Język Ruby zezwala na to, aby umożliwić tworzenie metod określających wartość `true` lub `false`, a ponadto zapewnić wywołaniom metody wygląd pytań:

```
if ticket.available?
  puts "Masz szczęście!"
else
  puts "Niestety, to miejsce zostało sprzedane."
end
```

Z prawdą i fałszem wiąże się więcej niż tylko obiekty `true` i `false`. Dla każdego wyrażenia w języku Ruby określana jest wartość w postaci obiektu, a każdy obiekt tego języka ma wartość prawda. Taka wartość w języku Ruby w przypadku prawie każdego obiektu jest wartością `true`. Jedynymi obiektami, dla których wartością prawda (lub wartością boolowską) jest `false`, jest obiekt `false` oraz specjalny niebędący encją obiekt `nil`. W rozdziale 7. bardziej szczegółowo omówiono wartości boolowskie i obiekt `nil`. Na razie możesz traktować wartość `false` i obiekt `nil` jako równorzędne pod względem funkcjonalnym wskaźniki wyniku testu negatywnego.

Eksperymentowanie z wyrażeniami instrukcji `if` w narzędziu irb jest dobrym sposobem na zorientowanie się, jak w języku Ruby wygląda logika warunkowa. Wypróbuj kilka przykładów podobnych do następujących:

```
>> if "abc"
>>   puts "Łańcuchy są obiektami true w języku Ruby!"
>> end
Łańcuchy są obiektami true w języku Ruby!
=> nil
>> if 123
>>   puts "To samo dotyczy liczb!"
>> end
To samo dotyczy liczb!
=> nil
>> if 0
>>   puts "Nawet 0 jest obiektem true, co nie ma miejsca w niektórych językach."
>> end
Nawet 0 jest obiektem true, co nie ma miejsca w niektórych językach.
=> nil
>> if 1 == 2
>>   puts "Jeden nie jest równe dwa, dlatego nie zostanie to wyświetlone."
>> end
=> nil
```

Pierwszy z powyższych przykładów, czyli instrukcja `if "abc"`, spowoduje wygenerowanie ostrzeżenia dotyczącego literałów łańcuchowych w warunkach. Ostrzeżenie możesz zignorować w odniesieniu do tych przykładów.

Zauważ, że narzędzie irb nie tylko dostosowuje się do wywołań metody `puts` ❶, ale też w ramach własnej inicjatywy zwraca wartość całego wyrażenia ❷. W przypadkach, w których występuje metoda `puts`, dla całego wyrażenia określana jest wartość w postaci obiektu `nil`, ponieważ wartość zwracana tej metody zawsze jest obiektem `nil`. W ostatnim przykładzie, w którym łańcuch nie jest wyświetlany (ponieważ nie jest spełniony warunek), wartością wyrażenia również jest obiekt `nil`, gdyż dla zakończonej niepowodzeniem instrukcji `if` (a ponadto pozbawionej gałęzi ratunkowej instrukcji `else`) także określana jest wartość jako obiekt `nil` ❸.

Pamiętając, że obiekt `nil` ma wartość boolowską `false`, w razie potrzeby możesz poeksperymentować za pomocą narzędzia irb. Wywołanie metody `puts` zwraca obiekt `nil`, dlatego jest fałszywe, nawet pomimo tego, że wyświetlany jest łańcuch. Jeśli metodę `puts` umieścisz w klauzuli `if`, klauzula będzie fałszywa. Jednak nadal będzie dla niej określana wartość. A zatem:

```
>> if puts "Zobaczysz to."
>>   puts "lecz nie to"
>> end
Zobaczysz to.
=> nil
```

Pierwsza metoda `puts` jest wykonywana, ale zwracana przez nią wartość, czyli obiekt `nil`, nie jest prawdziwa w sensie wartości boolowskich. Z tego powodu nie jest wykonywana druga metoda `puts`.

Choć jest to nienaturalny przykład, warto oswoić się z tym, że *wszystko w języku Ruby ma wartość boolowską*, a czasami nie jest to tym, czego możesz oczekiwać. Jak to często bywa, narzędzie irb może okazać się bardzo pomocne w zrozumieniu tego zagadnienia.

Gdy już obiekt `ticket` ma zdefiniowane pewne zachowania, cofnijmy się i zastanówmy nad tym, jakie zachowania są przypisywane każdemu obiektowi w języku Ruby podczas jego tworzenia.

## 2.3. Wbudowane zachowania obiektu

Nawet nowo utworzony obiekt nie jest zupełnie pusty. Od razu po utworzeniu obiekt odpowiada na kilka komunikatów. Każdy obiekt „rodzi się" z określonymi wbudowanymi możliwościami.

Aby zaznajomić się z listą wbudowanych metod, możesz wywołać metodę `methods` (i zastosować operację `sort` w celu ułatwienia przeglądania w sposób wizualny):

```
p Object.new.methods.sort
```

Wynikiem jest lista wszystkich komunikatów (metod), w jakie wyposażany jest nowo tworzony obiekt. Ostrzeżenie: dane wyjściowe są nieuporządkowane. Wynika to ze sposobu prezentowania tablic w języku Ruby. Ponadto metoda `methods` zapewnia tablicę nazw metod. Aby wyświetlić listę metod po jednej metodzie w każdym wierszu, w poleceniu użyj metody `puts` zamiast `p`.

```
[:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :class, :clone,
:define_singleton_method, :display, :dup, :enum_for, :eql?, :equal?, :extend,
:freeze, :frozen?, :hash, :inspect, :instance_eval, :instance_exec,
:instance_of?, :instance_variable_defined?, :instance_variable_get,
:instance_variable_set, :instance_variables, :is_a?, :kind_of?, :method,
:methods, :nil?, :object_id, :private_methods, :protected_methods,
:public_method, :public_methods, :public_send, :remove_instance_variable,
:respond_to?, :send, :singleton_class, :singleton_methods, :taint, :tainted?,
:tap, :to_enum, :to_s, :trust, :untaint, :untrust, :untrusted?]
```

Nie martw się, jeśli na razie większość powyższych metod nic Ci nie mówi. Możesz je wypróbować w narzędziu irb, jeśli jesteś ciekaw, jakie jest ich przeznaczenie (a także, jeśli nie obawiasz się wygenerowania komunikatów o błędzie).

Jednak kilka spośród tych wbudowanych metod, które zostaną tutaj szczegółowo przedstawione, jest wystarczająco popularnych i pomocnych, nawet w początkowych etapach zaznajamiania się z językiem Ruby. Dotyczy to następujących metod:

- `object_id`,
- `respond_to?`,
- `send` (synonim: `__send__`).

Dodanie tych metod do własnego zestawu narzędzi języka Ruby będzie czymś właściwym ze względu na ich możliwości, a także dlatego, że stanowią przykład metod wbudowanych.

**Porównanie obiektów ogólnych z obiektami podstawowymi**

Zażądanie za pomocą polecenia `Object.new` od interpretera języka Ruby utworzenia nowego obiektu powoduje uzyskanie tego, co wcześniej zostało nieformalnie określone mianem obiektu *ogólnego*. W języku Ruby istnieją też obiekty *podstawowe*, i jest to bardziej formalna nazwa. W przypadku wywołania metody `BasicObject.new` uzyskasz swego rodzaju proto-obiekt, który ma bardzo ograniczone możliwości. Nie możesz nawet zażądać od obiektu podstawowego wyświetlenia jego metod, ponieważ nie zawiera on metody `methods`! Okazuje się, że oferuje on tylko siedem metod, co wystarczy do tego, aby obiekt istniał i mógł zostać zidentyfikowany, i nic ponadto. W rozdziałach 3. i 13. dowiesz się więcej o tych obiektach podstawowych.

### 2.3.1. Unikatowe identyfikowanie obiektów za pomocą metody object_id

Z każdym obiektem w języku Ruby powiązany jest unikatowy identyfikator liczbowy. W celu wyświetlenia identyfikatora obiektu zażądaj od niego udostępnienia swojej metody `object_id` za pomocą następującego kodu lub podobnego do niego:

```
obj = Object.new
puts "Identyfikator obiektu obj to #{obj.object_id}."
str = "Łańcuchy również są obiektami, a to jest łańcuch!"
puts "Identyfikator obiektu łańcucha str to #{str.object_id}."
puts "Identyfikator liczby całkowitej 100 to #{100.object_id}."
```

Istnienie unikatowego identyfikatora liczbowego dla każdego obiektu może okazać się pomocne przy próbie określenia, czy dwa obiekty są takie same. W jaki sposób dwa obiekty mogą być identyczne? Cóż, obiekt liczby całkowitej `100` jest taki sam jak... obiekt liczby całkowitej `100` (zażądaj dwukrotnie od liczby 100 jej identyfikatora obiektu, a uzyskasz jednakowy wynik). Oto następny przypadek:

```
a = Object.new
b = a
puts "Identyfikator obiektu a to #{a.object_id}, a identyfikator obiektu b to
↪#{b.object_id}."
```

Nawet pomimo tego, że zmienne `a` i `b` są różne, obiekt, do którego się one odwołują, jest identyczny (w punkcie 2.5.1 zamieszczono więcej informacji na temat zagadnienia odwołań do obiektów). Może też mieć miejsce odwrotna sytuacja: choć czasami dwa obiekty wydają się być takie same, tak nie jest. Często dzieje się tak w przypadku łańcuchów. Rozważ następujący przykład:

```
string_1 = "Witaj"
string_2 = "Witaj"
puts "Identyfikator obiektu string_1 to #{string_1.object_id}."
puts "Identyfikator obiektu string_2 to #{string_2.object_id}."
```

**Identyfikator obiektu `string_1`:**

**Identyfikator obiektu `string_2`:**

Nawet pomimo tego, że te dwa łańcuchy zawierają ten sam tekst, z technicznego punktu widzenia nie są tym samym obiektem. Po wyświetleniu obu łańcuchów będzie widoczny identyczny wynik (`"Witaj"`). Same obiekty łańcuchów są jednak różne. Można to porównać do posiadania dwóch kopii tej samej książki: zawierają identyczną treść, ale nie są jedną rzeczą. Możesz zniszczyć jedną kopię, a druga pozostanie nienaruszona.

### IDENTYFIKATORY LICZBOWE I RÓWNOŚĆ OBIEKTÓW

Podobnie jak w przypadku instytucji grupującej ludzi jednym z aspektów nadawania obiektom identyfikatorów liczbowych w języku Ruby jest umożliwienie unikatowego identyfikowania, a w szczególności określania, kiedy dwa obiekty są tym samym obiektem.

Język Ruby oferuje różne metody porównywania obiektów pod kątem różnych typów równości. Jeśli istnieją dwa łańcuchy, możesz przeprowadzić test w celu sprawdzenia, czy zawierają one te same znaki. Możesz również sprawdzić, czy łańcuchy są tym samym obiektem (co, jak wcześniej pokazano, niekoniecznie jest regułą, nawet jeśli łańcuchy są złożone z tych samych znaków). To samo dotyczy (z niewielkimi różnicami) innych obiektów oraz ich typów.

Porównywanie identyfikatorów liczbowych pod kątem równości to tylko jeden ze sposobów określania jednakowości obiektów. Trochę dalej bardziej szczegółowo zajmiemy się tymi porównaniami. Na razie omówimy następną metodę wbudowaną na liście, czyli metodę `respond_to?`.

## *2.3.2. Uzyskiwanie możliwości obiektu za pomocą metody respond_to?*

Obiekty języka Ruby odpowiadają na komunikaty. W różnych momentach działania programu zależnie od obiektu i zdefiniowanych dla niego rodzajów metod obiekt może odpowiadać na dany komunikat albo nie. Na przykład następujący kod powoduje wygenerowanie błędu:

```
obj = Object.new
obj.talk
```

Język Ruby z wielką ochotą powiadomi o problemie:

```
NoMethodError: undefined method `talk' for #<Object:0x00000102836550>
```

Za pomocą metody `respond_to?` możesz wcześniej (przed zażądaniem od obiektu wykonania działania) określić, czy obiekt potrafi obsłużyć komunikat, który ma zostać mu wysłany. Metoda ta jest dostępna w przypadku wszystkich obiektów. Możesz sprawdzić, czy dowolny obiekt odpowiada na jakikolwiek komunikat. Metoda `respond_to?` pojawia się zwykle razem z logiką instrukcji warunkowej `if`:

```
obj = Object.new
if obj.respond_to?("talk")
  obj.talk
else
  puts "Niestety, obiekt nie rozpoznaje komunikatu talk."
end
```

Metoda `respond_to?` jest przykładem *introspekcji* lub *refleksji*. Są to dwa terminy, które odnoszą się do sprawdzania stanu działającego programu. Język Ruby oferuje kilka narzędzi związanych z introspekcją. Sprawdzanie metod obiektu za pomocą metody `methods` w zaprezentowany wcześniej sposób to kolejna technika *introspekcji* lub *refleksji* (w trzeciej części książki poznasz znacznie więcej tego typu technik).

Do tej pory korzystaliśmy z operatora kropki, aby wysyłać komunikaty do obiektów. Jest to zupełnie poprawne. Co jednak będzie, jeśli nie wiesz, jakie komunikaty mają zostać wysłane?

### 2.3.3. Wysyłanie komunikatów do obiektów za pomocą metody send

Załóżmy, że chcesz umożliwić użytkownikowi uzyskanie informacji z obiektu `ticket` przez wprowadzenie za pomocą klawiatury odpowiedniego terminu zapytania (`venue`, `performer` itp.). Oto wiersze, jakie należałoby dodać do istniejącego programu:

```
print "Żądane informacje: "
request = gets.chomp
```

Drugi wiersz kodu pobiera dane wpisane przy użyciu klawiatury, usuwa końcowy znak nowego wiersza i zapisuje wynikowy łańcuch w zmiennej `request`.

Na tym etapie możesz testować dane wejściowe dla kolejnych wartości za pomocą operatora porównania mającego postać dwóch znaków równości (==) porównującego łańcuchy na podstawie ich zawartości, a także wywołując metodę, której wartość zapewnia dopasowanie:

```
if request == "venue"
  puts ticket.venue
elsif request == "performer"
  puts ticket.performer
...
```

Aby jednak być dokładnym, konieczne byłoby kontynuowanie procesu dla całej listy właściwości obiektu `ticket`. Okazałoby się to zbyt czasochłonne.

Istnieje alternatywa: wyraz możesz wysłać bezpośrednio do obiektu `ticket`. Zamiast wcześniejszego kodu należałoby użyć następujących wierszy:

```
if ticket.respond_to?(request)   ←❶
  puts ticket.send(request)
else
  puts "Takie informacje są niedostępne"
end
```

W tej wersji kodu zastosowano metodę `send` jako uniwersalny sposób kierowania komunikatu do obiektu `ticket`. Eliminuje to konieczność przetwarzania całej listy możliwych żądań. Zamiast tego po sprawdzeniu, że obiekt `ticket` ma określoną procedurę postępowania ❶, przekazujesz mu komunikat i zezwalasz na dalsze działania.

Operator kropki będzie przeważnie używany do wysyłania komunikatów do obiektów. Alternatywa w postaci metody `send` może być jednak przydatna, a ponadto może oferować na tyle duże możliwości, a jednocześnie podatność na błędy, że prawie zawsze zasługuje przynajmniej na poziom zabezpieczeń reprezentowanych przez wywołanie metody `respond_to?`. W niektórych sytuacjach metoda ta może okazać się nawet zbyt obszerna, aby była bezpieczna. Za pomocą metody `send` do obiektu możesz wysłać komunikat tylko wtedy, gdy uwzględniono go na wcześniej ustalonej „białej liście" komunikatów. Ogólna wskazówka dotyczy ostrożności: zachowaj ją przy wysyłaniu dowolnych komunikatów do obiektów, a zwłaszcza w sytuacji, gdy komunikaty bazują na wyborze dokonanym przez użytkownika lub wprowadzonych przez niego danych.

**Użycie metody __send__ lub public_send zamiast metody send**

Wysyłanie to szerokie pojęcie: wysyłana jest wiadomość e-mail, dane do gniazd wejścia-wyjścia itd. Programy często definiują metodę o nazwie `send`, która powoduje konflikt z wbudowaną metodą `send` języka Ruby. Z tego powodu język zapewnia alternatywny sposób wywoływania tej metody w postaci nazwy `__send__`. Zwykle nikt nie tworzy metody o takiej nazwie, dlatego wbudowana wersja metody języka Ruby jest zawsze dostępna i nigdy nie dochodzi do konfliktu z nowo utworzonymi metodami. Nazwa `send` wygląda dziwnie, ale z punktu widzenia kolizji nazw metod jest bezpieczniejsza od zwykłej wersji `send`.

Ponadto istnieje bezpieczna (ale pod innym względem) wersja metody `send` (lub `send`) o nazwie `public_send`. Różnica między zwykłą metodą `send` i metodą `public_send` polega na tym, że pierwsza z nich może wywoływać metody prywatne obiektu, a druga nie. Metody prywatne zostaną omówione w dalszej części książki, ale jeśli jesteś ciekawy, dlaczego metoda `public_send` pojawiła się na liście metod już teraz, to z grubsza rzecz biorąc z tego właśnie powodu.

Zajmiemy się teraz bliżej składnią i semantyką argumentów metody.

## 2.4. Dokładna analiza argumentów metody

Metody tworzone w języku Ruby mogą pobierać wiele argumentów albo żadnego. Zezwalają one również na użycie zmiennej liczby argumentów. W podrozdziale na kilka różnych sposobów przyjrzymy się semantyce argumentów.

- Różnica między argumentami wymaganymi i opcjonalnymi.
- Sposób przypisywania wartości domyślnych do argumentów.
- Reguły określające kolejność wymuszającą uporządkowanie parametrów w sygnaturze metody, aby interpreter języka Ruby mógł przetworzyć listy argumentów w wywołaniach metod i poprawnie powiązać parametry.
- Działania niedozwolone w przypadku argumentów w języku Ruby.

Na końcu podrozdziału — w tabeli 2.2 — zostanie dokonane podsumowanie argumentów.

**UWAGA** Z semantyką listy argumentów związanych jest więcej informacji, niż tutaj podano. Dokładniej rzecz biorąc, w języku Ruby występuje takie pojęcie jak argumenty słów kluczowych (lub nazwane parametry). Pojęcie to jest silnie powiązane z użyciem tablic asocjacyjnych jako argumentów metody. Z tego właśnie powodu ich pełne omówienie pojawi się przy obszernym prezentowaniu tablic asocjacyjnych w rozdziale 9.

### 2.4.1. Argumenty wymagane i opcjonalne

Przy wywoływaniu metody języka Ruby musisz podać poprawną liczbę argumentów. W przeciwnym razie interpreter języka zgłosi wystąpienie problemu. Na przykład wywołanie jednoargumentowej metody z trzema argumentami:

```
obj = Object.new
def obj.one_arg(x)
```

```
  puts "Wymagam jednego i tylko jednego argumentu!"
end
obj.one_arg(1,2,3)
```

spowoduje wygenerowanie następującego komunikatu:

```
ArgumentError: wrong number of arguments (3 for 1)
```

Możliwe jest utworzenie metody zezwalającej na dowolną liczbę argumentów. W tym celu na początku nazwy pojedynczego argumentu umieść znak gwiazdki (*):

```
def obj.multi_args(*x)
  puts "Mogę pobrać wiele argumentów albo żadnego!"
end
```

Notacja `*x` oznacza, że przy wywoływaniu metody możesz podać dowolną liczbę argumentów (lub nie podawać żadnego). W tym przypadku zmiennej `x` przypisywana jest tablica wartości odpowiadających dowolnym przekazanym argumentom. Dokonując przejścia tablicy, możesz następnie sprawdzić kolejno wartości (w rozdziale 9. przyjrzymy się bliżej tablicom).

Możliwe jest dopasowanie liczby argumentów przez połączenie argumentów wymaganych i opcjonalnych:

```
def two_or_more(a,b,*c)
  puts "Wymagam co najmniej dwóch argumentów!"
  puts "I otrzymałem wystarczającą ich ilość: "
  p a, b, c
end
```

W tym przykładzie `a` i `b` to argumenty wymagane. Ostatni argument `*c` spowoduje pobranie wszystkich innych argumentów, które mogą zostać wysłane, a następnie umieszczenie ich w tablicy w zmiennej `c`. Po wywołaniu metody `two_or_more(1,2,3,4,5)` uzyskasz następujący raport z informacjami o tym, co zostało przypisane zmiennym `a`, `b` i `c`:

```
Wymagam co najmniej dwóch argumentów!
I otrzymałem wystarczającą ich ilość:
1
2
[3, 4, 5]
```

Użycie litery `p` zamiast nazwy `print` lub `puts` powoduje wyświetlenie tablicy z wykorzystaniem powiązanej z nią notacji. W przeciwnym razie każdy element tablicy pojawiłby się w osobnym wierszu, co utrudniłoby stwierdzenie, że w ogóle ma się do czynienia z tablicą.

Opcjonalność argumentu można też zapewnić przez określenie dla niego wartości domyślnej.

### 2.4.2. Wartości domyślne argumentów

Podanie wartości domyślnej dla argumentu powoduje, że w sytuacji, gdy argument nie zostanie określony, odpowiadającej mu zmiennej zostanie przekazana wartość domyślna.

Argumenty domyślne są identyfikowane przez znak równości i wartość. Oto przykład:

```
def default_args(a,b,c=1)
  puts "Wartości zmiennych: ",a,b,c
end
```

Efektem utworzenia wywołania:

```
default_args(3,2)
```

będzie następujący wynik:

```
Wartości zmiennych:
3
2
1
```

Ponieważ w wywołaniu metody dla argumentu `c` nie podano żadnej wartości, ustawiona została dla niego wartość domyślna `1` określona dla tego argumentu na liście parametrów. Jeśli podasz trzeci argument, jego wartość nadpisze domyślne przypisanie w postaci wartości `1`. Następujące wywołanie:

```
default_args(4,5,6)
```

wygeneruje wynik:

```
Wartości zmiennych:
4
5
6
```

Naprawdę ciekawie zaczyna się w momencie łączenia i dopasowywania różnych elementów składni argumentów, gdy konieczne jest ustalenie, w jakiej kolejności wszystko ma zostać ustawione.

### *2.4.3. Kolejność parametrów i argumentów*

Jakich danych wyjściowych będziesz oczekiwać w przypadku poniższego fragmentu kodu?

```
def mixed_args(a,b,*c,d)
  puts "Argumenty:"
  p a,b,c,d
end
mixed_args(1,2,3,4,5)
```

Wcześniej pokazano, że parametr ze znakiem gwiazdki (np. `*c`) powoduje pobranie pozostałych argumentów, a przynajmniej tak było w metodzie `two_or_more`, w której `*c` wystąpiło na końcu listy parametrów. Co ma miejsce, gdy po parametrze z gwiazdką pojawi się kolejny argument?

Zasadniczo interpreter języka Ruby próbuje przypisać wartości do jak największej liczby zmiennych. Parametry ze znakiem gwiazdki uzyskują najniższy priorytet: jeśli metodzie zabraknie argumentów po wykonaniu operacji przypisań argumentów wymaganych, parametr przechwytujący wszystko, taki jak `*c`, przyjmie postać pustej tablicy. Argumenty wymagane, znajdujące się zarówno przed parametrem `*c`, jak i po nim, są przetwarzane przed nim.

Oto dane wyjściowe poprzedniego fragmentu kodu:

```
Argumenty:
1
2
[3, 4]
5
```

Parametry `a` i `b` stają się pierwszymi dwoma argumentami, odpowiednio `1` i `2`. Ponieważ parametr `d` znajdujący się na końcu listy reprezentuje argument wymagany, pobiera pierwszą dostępną wartość z prawego końca listy argumentów, czyli wartość `5`. Wszystko, co pozostanie w środku `(3, 4)`, zostanie pobrane przez argument `c`.

Jeśli tylko podasz liczbę argumentów wystarczającą do dopasowania do argumentów wymaganych metody, tablica parametru z gwiazdką będzie pusta. Następujące wywołanie metody:

```
mixed_args(1,2,3)
```

zwróci następujące dane wyjściowe:

```
1
2
[]
3
```

W tym przykładzie parametrowi `c` nie powiodło się. Nie pozostały dla niego żadne argumenty.

Składnia parametrów może okazać się dość wyszukana. Poniżej zaprezentowano metodę, która pobiera argument wymagany, argument opcjonalny z wartością domyślną 1, dwa dodatkowe argumenty wymagane pobierane z prawej strony oraz argument z gwiazdką umieszczony gdzieś w środku listy argumentów.

```
def args_unleashed(a,b=1,*c,d,e)
  puts "Argumenty:"
  p a,b,c,d,e
end
```

Poniżej przedstawiono przebieg sesji narzędzia irb, w ramach której metoda została sprawdzona. Zauważ, że w każdym przypadku wartością zwracaną wywołania metody jest tablica złożona z wszystkich wartości. Jest to wartość zwracana wywołania metody `p`. Wartość stanowi reprezentację tablicową tych samych wartości, które są widoczne po wyświetleniu w osobnych wierszach jako niezależne wartości:

```
>> args_unleashed(1,2,3,4,5)      ←❶
1
2
[3]
4
5
=> [1, 2, [3], 4, 5]
>> args_unleashed(1,2,3,4)        ←❷
1
2
[]
3
4
```

```
=> [1, 2, [], 3, 4]
>> args_unleashed(1,2,3)          ←——❸
1
1
[]
2
3
=> [1, 1, [], 2, 3]
>> args_unleashed(1,2,3,4,5,6,7,8)  ←——❹
1
2
[3, 4, 5, 6]
7
8
=> [1, 2, [3, 4, 5, 6], 7, 8]
>> args_unleashed(1,2)            ←——❺
ArgumentError: wrong number of arguments (2 for 3+)
```

Pierwsze wywołanie metody `args_unleashed` zawiera pięć argumentów ❶. Oznacza to, że jest wystarczająca liczba argumentów: nadpisywana jest wartość domyślna argumentu `b`, a tablica `c` ma jeden element. Drugie wywołanie jest bardziej ograniczone ❷. W jego przypadku argument `c` jest na przegranej pozycji: nadpisywana jest wartość domyślna argumentu `b`, a argument `c` pozostaje pusty (ponieważ dwa ostatnie argumenty są rezerwowane przez argumenty wymagane `d` i `e`).

W trzecim wywołaniu ma miejsce jeszcze większe „zaciskanie pasa” ❸. Tym razem liczba argumentów wystarcza jedynie do spełnienia podstawowych wymagań. Oznacza to, że wartości mogą zostać przypisane argumentom `a`, `d` i `e`. Parametr `b` sięga po swoją wartość domyślną, a parametr `c` jest pusty.

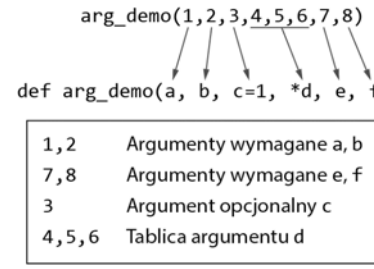
W przypadku czwartego wywołania sytuacja wygląda inaczej ❹: tym razem istnieje więcej argumentów niż wymaga metoda, a ponadto ich liczba jest wystarczająca do przypisania wartości parametrom opcjonalnym i parametrom z wartością domyślną. Dla parametru `c` to prawdziwa „żyła złota”, która spełnia swoją rolę, pobierając wszystkie argumenty niepotrzebne w innym miejscu. Ostatecznie parametr ten pobiera cztery elementy.

Z kolei piąte wywołanie nie wysyła liczby argumentów wystarczającej do spełnienia podstawowych wymagań ❺. Ponieważ nie mogą zostać utworzone powiązania zmiennych, wystąpi błąd argumentów.

Oprócz praktycznego znaczenia składni argumentów najważniejszą rzeczą, jaką zapewniają przedstawione przykłady, jest być może oczywista informacja: niezależnie od realizowanych działań ostatecznie każdy parametr zostaje powiązany z wartością. Nie ma miejsca sytuacja, w której parametr po prostu trafia w „nicość”. Gdy znajduje się na liście, stanie się zmienną lokalną w obrębie metody, nawet jeśli jest on tylko powiązany z pustą tablicą (czasami tak było w przypadku parametru `c`). Nie musisz używać każdej takiej zmiennej, ale powiązania zawsze są tworzone.

Rysunek 2.2 przedstawia graficzną reprezentację podstawowej logiki przypisywania argumentów. Zestawienie przypisań w oknie prezentuje kolejność priorytetów: jako pierwsze obsługiwane są argumenty wymagane, a następnie argument opcjonalny z wartością domyślną i parametr z gwiazdką.

W przypadku złożonych wymagań względem argumentów musisz uważnie przestrzegać reguł, a także pamiętać o tym, czego nie możesz robić.




**Rysunek 2.2.** Logika przypisywania argumentów w praktyce

### 2.4.4. Działania niedozwolone w przypadku list argumentów

Parametry cechują się hierarchią w obrębie grupy. Parametry wymagane mają najwyższy priorytet, niezależnie od tego, czy występują po lewej, czy po prawej stronie listy. Wszystkie parametry opcjonalne muszą znaleźć się w środku, który może być środkiem „nicości”:

```
def all_optional(*args)
```
← **Brak argumentów wymaganych z lewej lub prawej strony**

Argumenty wymagane mogą występować tylko z lewej lub prawej strony albo z obu stron.

Niedozwolone jest umieszczanie parametru pobierającego argumenty z lewej strony jakiegokolwiek argumentu z wartością domyślną. Zastosowanie następującego kodu:

```
def broken_args(x,*y,z=1)
end
```

spowoduje wygenerowanie błędu składni, ponieważ w żaden sposób nie może to być poprawne. Po przypisaniu wartości argumentowi `x` i pobraniu wszystkich pozostałych argumentów w tablicy `y` nic nie pozostanie dla argumentu `z`. Jeśli argument `z` stanie się argumentem znajdującym się po prawej stronie, pozostawiając resztę dla argumentu `y`, nie będzie miało sensu opisywanie argumentu `z` jako „opcjonalnego” lub „mającego wartość domyślną”. Sytuacja stanie się jeszcze bardziej problematyczna przy próbie zastosowania czegoś równie niedozwolonego, na przykład `(x, *y, z=1, a, b)`. Na szczęście w języku Ruby na liście parametrów nie może występować więcej niż jeden argument z gwiazdką. Upewnij się, że argumenty zostały sensownie uporządkowane, a w razie możliwości zachowaj listy argumentów w dość prostej postaci.

W tabeli 2.2 podsumowano dotychczas przedstawione informacje o składni i semantyce argumentów. Tabelę możesz traktować bardziej jako zasób referencyjny niż coś, co wymaga zapamiętania i regularnego sprawdzania, pod warunkiem że zrozumiałe są zasady działania poszczególnych przykładów.

Jak widać w tabeli 2.2, argumenty wysyłane do metod są przypisywane zmiennym, a dokładniej rzecz biorąc, zmiennym lokalnym, które są widoczne i możliwe do użycia w czasie działania metody. Przypisywanie do zmiennych lokalnych przez powiązanie argumentów metody to tylko jeden wariant ogólnego procesu przypisywania do zmiennych lokalnych. Zagadnieniu temu przyjrzymy się szczegółowo jako następnemu.

## 2.5. Zmienne lokalne i przypisywanie do zmiennych

Nazwy zmiennych lokalnych rozpoczynają się od małej litery lub znaku podkreślenia, a ponadto złożone są ze znaków alfanumerycznych i znaków podkreślenia. Wszystkie następujące nazwy zmiennych lokalnych są poprawne, włącznie z nazwą zawierającą tylko znak podkreślenia:

**Tabela 2.2.** Przykładowe sygnatury metody z argumentami wymaganymi, argumentami opcjonalnymi i argumentami z wartością domyślną

| Typy argumentów | Sygnatura metody | Przykładowe wywołania | Przypisania do zmiennej |
|---|---|---|---|
| Wymagany | `def m(a,b,c)` | `m(1,2,3)` | `a = 1, b = 2, c = 3` |
| Opcjonalny | `def m(*a)` | `m(1,2,3)` | `a = [1,2,3]` |
| Z wartością domyślną | `def m(a=1)` | `m` | `a = 1` |
| | | `m(2)` | `a = 2` |
| Wymagany/opcjonalny | `def m(a,*b)` | `m(1)` | `a = 1, b = [ ]` |
| Wymagany/z wartością domyślną | `def m(a,b=1)` | `m(2)` | `a = 2, b = 1` |
| | | `m(2,3)` | `a = 2, b = 3` |
| Z wartością domyślną/opcjonalny | `def m(a=1,*b)` | `m` | `a = 1, b = [ ]` |
| | | `m(2)` | `a = 2, b = [ ]` |
| Wymagany/z wartością domyślną/opcjonalny | `def m(a,b=2,*c)` | | `a = 1, b = 2, c = [ ]` |
| | | | `a = 1, b = 3, c = [ ]` |
| | | | `a = 1, b = 3, c = [5,7]` |
| Wymagany/z wartością domyślną/opcjonalny/wymagany | `def m(a,b=2,*c,d)` | | `a = 1, b = 2, c = [ ], d = 3` |
| | | | `a = 1, b = 3, c = [ ], d = 5` |
| | | | `a = 1, b = 3, c = [5], d = 7` |
| | | | `a = 1, b = 3, c = [5,7], d = 9` |

```
x
_x
name
first_name
plan9
user_ID
_
```

Słowo *lokalne* w terminie *zmienne lokalne* dotyczy faktu, że mają one ograniczony *zasięg*: zmienna lokalna jest widoczna tylko w ograniczonej części programu, takiej jak definicja metody. Nazwy zmiennych lokalnych mogą być ponownie wykorzystywane w różnych zasięgach. Na przykład nazwy zmiennej `x` możesz użyć w więcej niż jednym miejscu. Dopóki mają one różne zasięgi, dwie zmienne `x` są traktowane jako zupełnie osobne (pamiętaj o tym, że w tradycyjnym stylu związanym z językiem Ruby w przypadku zmiennych lokalnych preferowane są nazwy `znak_podkreślenia` zamiast nazw `znakiRóżnejWielkości`).

Pojęcie zasięgu jest ważne samo w sobie. Obszerniej zajmiemy się nim w rozdziale 5. Możesz jednak już teraz zacząć zaznajamiać się z niektórymi kluczowymi aspektami związanymi z zasięgiem, analizując proces przetwarzania zmiennych lokalnych. Klasycznym przypadkiem zasięgu lokalnego jest definicja metody. Zaobserwuj w tym przykładzie, co się dzieje ze zmienną `x`:

```
def say_goodbye
  x = "Żegnaj"   ←❶
  puts x
```

```
end
def start_here
  x = "Witaj"  ←❷
  puts x
  say_goodbye                                  ←❸
  puts "Sprawdźmy, czy zmienna x nie zmieniła się:"
  puts x                                       ←❹
end
start_here     ←❺
```

Oto dane wyjściowe tego programu:

```
Witaj
Żegnaj
Sprawdźmy, czy zmienna x nie zmieniła się:
Witaj
```

Po wywołaniu metody `start_here` ❺ zostanie ona wykonana. W obrębie tej metody zmiennej `x` przypisywany jest łańcuch `Witaj` ❷, czyli *tej* zmiennej `x` o zasięgu wewnątrz metody.

Metoda `start_here` wyświetla wartość swojej zmiennej `x` (`Witaj`), a następnie wywołuje metodę `say_goodbye` ❸. W tej metodzie ma miejsce coś podobnego: zmiennej `x` ❶ przypisywany jest łańcuch (`Żegnaj`). Jest to jednak inna zmienna `x`, co możesz zauważyć po zakończeniu wywołania metody `say_goodbye` i zwróceniu sterowania metodzie `start_here`: interpreter języka Ruby wyświetla wartość tej zmiennej `x`, a wartością jest nadal łańcuch `Witaj` ❹. Użycie zmiennej `x` jako nazwy zmiennej lokalnej w zasięgu jednej metody nie wpłynęło na wartość tej zmiennej w zasięgu innej metody.

Zmienne lokalne w ostatnim przykładzie są tworzone w ramach jawnego przypisania (jak pokazano wcześniej, takie zmienne mogą zostać utworzone przez powiązanie argumentów metody z jej parametrami). Co jednak ma miejsce w przypadku przypisywania lub inicjalizacji? Jaka dokładnie jest relacja między zmienną i reprezentującym ją obiektem?

### 2.5.1. Zmienne, obiekty i odwołania

Przypisania do zmiennych zapewniają widoczny efekt powodujący, że zmienna po lewej stronie zostanie ustawiona jako równa obiektowi po prawej stronie przypisania. Na przykład w wyniku zastosowania następującego przypisania:

```
str = "Witaj"
```

instrukcje, takie jak `puts str`, zapewnią łańcuch `"Witaj"`, który zostanie wyświetlony i przetworzony.

Przyjrzyj się teraz następującemu przykładowi:

```
str = "Witaj"
abc = str
puts abc
```

Powyższy kod również wyświetla łańcuch `"Witaj"`. Jak widać, zmienna `abc` też zawiera ten łańcuch dzięki temu, że została do niej przypisana zmienna `str`.

To jednak nie wszystko. W następnym przykładzie zastosowano metodę o nazwie `replace`, która nowym tekstem zastępuje lokalnie zawartość łańcucha:

```
str = "Witaj"
abc = str
str.replace("Żegnaj")
puts str
puts abc
def say_goodbye
  str = "Witaj"
  abc = str
  str.replace("Żegnaj")
  puts str
  puts abc
end

say_goodbye
```

Przyjrzyj się dokładniej danym wyjściowym.

```
Żegnaj
Żegnaj
```

Pierwszy i drugi łańcuch `"Żegnaj"` to odpowiednio wartości zmiennych `str` i `abc`. Zastąpiono jednak tylko wartość zmiennej `str`. W jaki sposób został zastąpiony łańcuch w zmiennej `abc`?

### WPROWADZENIE DO ODWOŁAŃ

Odpowiedzią na powyższe pytanie jest to, że zmienne w języku Ruby (z kilkoma wyjątkami, wśród których najbardziej zauważalne są zmienne powiązane z liczbami całkowitymi) nie przechowują wartości obiektu. Zmienna `str` nie zawiera łańcucha `"Witaj"`. Zamiast niego zmienna przechowuje odwołanie do obiektu łańcuchowego. Jest to obiekt cechujący się tym, że składa się z liter tworzących łańcuch `"Witaj"`.

W przypisaniu z nazwą zmiennej po lewej stronie i obiektem po prawej stronie zmienna odbiera odwołanie do obiektu. W przypisaniu między dwoma zmiennymi (`abc = str`) zmienna po lewej stronie otrzymuje kopię odwołania przechowywanego w zmiennej po prawej stronie. Efektem takiego przypisania jest to, że obie zmienne zawierają odwołania do tego samego obiektu.

To, że zmienne przechowują odwołania do obiektów, ma wpływ na operacje, które zmieniają obiekty. Następująca operacja zastępowania łańcuchów:

```
str.replace("Żegnaj")
```

Zastępuje łańcuchem `"Żegnaj"` znaki łańcucha, do którego odwołaniem jest zmienna `str`. Zmienna `abc` zawiera kolejne odwołanie do tego samego obiektu łańcuchowego. Nawet pomimo tego, że komunikat `replace` kierowany jest do zmiennej `str`, powoduje zmianę obiektu, do którego odwołuje się odwołanie w zmiennej `abc`. Po wyświetleniu wartości zmiennej `abc` wynikiem będzie zawartość zmienionego łańcucha.

**Usuwanie odwołania: wartości bezpośrednie**

Niektóre obiekty w języku Ruby są przechowywane w zmiennych jako wartości bezpośrednie. Obejmują one liczby całkowite, symbole (np. `:this`) oraz obiekty specjalne `true`, `false` i `nil`. W momencie przypisania jednej z tych wartości do zmiennej (`x = 1`) przechowuje ona samą wartość, a nie odwołanie do niej.

Z praktycznego punktu widzenia ta kwestia nie jest istotna (w zawartych w książce omówieniach odwołań i powiązanych z nimi zagadnieniach często będzie ona traktowana jako domniemana). W języku Ruby proces usuwania odwołań do obiektów jest obsługiwany automatycznie. Przeciwnie niż w przypadku obiektu z bezpośrednią wartością całkowitoliczbową nie musisz wykonywać żadnych dodatkowych działań, aby wysłać komunikat do obiektu, który zawiera na przykład odwołanie do łańcucha.

Z regułą reprezentowania wartości bezpośrednich związanych jest jednak kilka interesujących konsekwencji, a zwłaszcza w przypadku liczb całkowitych. Podstawowa sprawa to fakt, że dowolny obiekt reprezentowany jako wartość bezpośrednia zawsze jest dokładnie tym samym obiektem, niezależnie od tego, ile zmiennych zostało do niego przypisanych. Istnieje tylko jeden obiekt `100`, tylko jeden obiekt `false` itd.

Unikatowa natura zmiennych powiązanych z liczbami całkowitymi, która cechuje się bezpośredniością, wynika z braku w języku Ruby operatorów preinkrementacji i postinkrementacji. Oznacza to, że w języku tym nie jest możliwe zastosowanie następujących wierszy:

```
x = 1
x++     # Brak takiego operatora
```

Przyczyną jest to, że na skutek bezpośredniej obecności wartości `1` w zmiennej `x` wiersz `x++` przyjąłby postać `1++`. Oznacza to, że liczba 1 zostałaby zmieniona na liczbę 2, co nie ma sensu.

Dla każdego obiektu w języku Ruby może i musi występować co najmniej jedno odwołanie, które się do niego odwołuje. Jeśli nie ma żadnych odwołań, obiekt uważany jest za nieaktywny. Spowoduje to zwolnienie i ponowne wykorzystanie jego obszaru pamięci.

Jeżeli występują co najmniej dwie zmienne zawierające odwołania do pojedynczego obiektu, możesz użyć dowolnej z nich (zmienne są równorzędne), aby wysłać komunikaty do obiektu. Odwołania tworzą ze swoimi obiektami relację typu „wiele do jednego”. Jeśli jednak przypiszesz całkowicie nowy obiekt do zmiennej, która już odwołuje się do obiektu, sytuacja ulegnie zmianie.

### 2.5.2. Odwołania w przypisaniu do zmiennej i ponowne przypisanie

Każdorazowo przy przypisywaniu do zmiennej (czyli w sytuacji, gdy nazwę zmiennej umieszczasz po lewej stronie znaku równości, a coś innego po jego prawej stronie) proces jest realizowany od początku: zmienna jest czyszczona i tworzone jest nowe przypisanie.

Oto nowa wersja wcześniejszego przykładu, która ilustruje tę kwestię:

```
str = "Witaj"
abc = str
str = "Żegnaj"
puts str
puts abc
```

Tym razem dane wyjściowe są następujące:

```
Żegnaj
Witaj
```

Drugie przypisanie do zmiennej `str` zapewnia jej odwołanie do innego obiektu łańcuchowego. W tym przypadku zmienne `str` i `abc` rozdzielają się od siebie. Zmienna `abc` nadal odwołuje się do starego łańcucha (z zawartością `"Witaj"`), ale zmienna `str` odwołuje się teraz do innego łańcucha (z zawartością `"Żegnaj"`).

Pierwsza wersja programu modyfikowała pojedynczy łańcuch, ale jego druga wersja zawiera dwa osobne łańcuchy. Po ponownym zastosowaniu zmienna `str` nie ma nic więcej wspólnego z obiektem, do którego odwoływała się poprzednio. Jednak ponowne użycie zmiennej `str` nie ma żadnego wpływu na zmienną `abc`, która nadal zawiera odwołanie do oryginalnego łańcucha.

> **UWAGA** W przykładach użyto zmiennych lokalnych w celu zademonstrowania tego, co ma miejsce, a co nie, w momencie przypisywania do zmiennej, dla której wykonano już operację przypisania. Zaprezentowane tutaj reguły i zachowania nie dotyczą jednak wyłącznie zmiennych lokalnych. Te same reguły obowiązują w przypadku zmiennych klas, globalnych i instancji (dziwną rzeczą jest to, że dotyczy to również tak zwanych stałych, dla których przypisanie możesz wykonać więcej niż raz!). Wszystkie te kategorie identyfikatora są *wartościami l*, czyli mogą być lewą stroną przypisania lub jego celem (porównaj to na przykład z przypisaniem `100 = 10`, które nie jest poprawne, ponieważ liczba `100` nie jest wartością *l*). Wszystkie wartości *l* zachowują się tak samo pod względem sposobu wiązania z ich prawą stroną, a także tego, co ma miejsce, gdy dana wartość *l* zostanie użyta więcej niż raz.

W języku Ruby zmienne są często opisywane jako etykiety lub nazwy obiektów. Jest to przydatne porównanie. Załóżmy, że dla psa są używane dwa imiona. Zdania „Zabieram Fido do weterynarza" i „Zabieram Reksia do weterynarza" odnoszą się do tego samego zwierzęcia. Jeśli jednak staniesz się posiadaczem nowego psa i nadasz mu imię Fido, zmienią się powiązania nazwy z psem. Imiona Fido i Reksio nie odwołują się już do tego samego psa, a imię Fido przestało być powiązane z pierwszym psem.

Nowy Fido nie musi nawet być psem. Mógłbyś przestać wołać na psa Fido i zacząć używać tego imienia w odniesieniu do własnego samochodu. Odpowiada temu użycie przypisania `x = 1`, po którym następuje `x = "A string"`. Identyfikator `x` jest ponownie używany dla całkowicie nowej klasy obiektu (`String` zamiast `Fixnum`). W przeciwieństwie do niektórych języków Ruby nie oferuje zmiennych z określonym typem. Dowolna zmienna może być powiązana z dowolnym obiektem dowolnej klasy w dowolnym momencie.

Z semantyką odwołań i przypisania (ponownego przypisania) związane są istotne konsekwencje wpływające na sposób postępowania przy wywoływaniu metody z argumentami. Co otrzymuje metoda? Jak metoda może postąpić z tym, co otrzyma?

### 2.5.3. Odwołania i argumenty metody

Pozostańmy przy przykładzie bazującym na łańcuchach, ponieważ można je łatwo zmieniać i śledzić. Oto metoda, która pobiera jeden argument:

```
def change_string(str)
  str.replace("Nowa zawartość łańcucha!")
end
```

Utwórz następnie łańcuch i wyślij go do metody `change_string`:

```
s = "Oryginalna zawartość łańcucha!"
change_string(s)
```

Sprawdź teraz zmienną `s`:

```
puts s
```

Okazuje się, że zawartość łańcucha, do którego odwołuje się zmienna `s`, zmieniła się na następującą:

```
Nowa zawartość łańcucha!
```

Oznacza to, że w obrębie metody `change_string` zmiennej `str` przypisano odwołanie do łańcucha, do którego odwołuje się też zmienna `s`. Przy wywoływaniu metody z argumentami w rzeczywistości ma miejsce „przemycanie" odwołań do obiektów. Gdy metoda będzie w posiadaniu odwołania, wszelkie zmiany dokonane przez nią w obiekcie za pośrednictwem odwołania będą widoczne podczas sprawdzania obiektu z wykorzystaniem dowolnego z jego odwołań.

Język Ruby zapewnia techniki służące do ochrony obiektów przed modyfikowaniem w sytuacji, gdy tego oczekujesz lub wymagasz.

#### UŻYCIE DLA OBIEKTÓW METOD DUP I FREEZE

Aby ochronić obiekty przed modyfikowaniem w obrębie metod, do których są one wysyłane, możesz zastosować metodę `dup` przeznaczoną do duplikowania obiektu:

```
s = "Oryginalna zawartość łańcucha!"
change_string(s.dup)
puts s          ←——— Wyświetla łańcuch „Oryginalna zawartość łańcucha!"
```

Możliwe jest też użycie dla obiektu metody `freeze`, która uniemożliwia wprowadzenie w nim dodatkowych zmian:

```
s = "Oryginalna zawartość łańcucha!"
s.freeze                          Błąd środowiska uruchamiania: nie można
change_string(s)  ←———┘           zmodyfikować zablokowanego łańcucha
```

Zauważ, że metoda `freeze` nie ma odpowiednika w postaci metody o odwrotnym działaniu. Proces blokowania nie może zostać cofnięty.

Całość obrazu dopełnia metoda o nazwie `clone`. Przypomina ona bardzo metodę `dup`. Różnicą jest to, że w przypadku użycia metody `clone` dla zablokowanego obiektu uzyskany klon również jest zablokowany. Jeśli natomiast dla zablokowanego obiektu zastosujesz metodę `dup`, duplikat nie będzie zablokowany.

Dysponując takimi narzędziami jak `dup`, `clone` i `freeze`, możesz chronić obiekty przed większością szkodliwych operacji modyfikujących. Część zagrożeń nadal jednak „czai się w ukryciu". Jeśli nawet zablokujesz tablicę, w dalszym ciągu możliwe jest zmodyfikowanie obiektów *wewnątrz* tablicy (przy założeniu, że nie są zablokowane):

```
>> numbers = ["jeden", "dwa", "trzy"]
=> ["jeden", "dwa", "trzy"]
>> numbers.freeze
=> ["jeden", "dwa", "trzy"]
>> numbers[2] = "cztery"
RuntimeError: can't modify frozen array
>> numbers[2].replace("cztery")
=> "cztery"
>> numbers
=> ["jeden", "dwa", "cztery"]
```

W tym przykładzie zablokowanie tablicy `numbers` oznacza, że nie możesz jej zmienić ❶. Łańcuchy wewnątrz tablicy nie są jednak zablokowane. Jeśli dla łańcucha `"trzy"` wykonasz operację `replace`, złośliwie zamieniając go w łańcuch `"cztery"` ❷, podczas ponownego sprawdzania (nadal zablokowanej!) tablicy ❸ pojawi się nowa zawartość łańcucha.

Zachowaj ostrożność w przypadku odwołań i pamiętaj, że odwołanie do obiektu w obrębie kolekcji nie jest tym samym co odwołanie do kolekcji (gdy szczegółowo przyjrzymy się kolekcjom w rozdziale 9., mocno przekonasz się do istnienia kolekcji jako pełnoprawnych obiektów).

Ostatnia uwaga dotycząca zmiennych, a w szczególności zmiennych lokalnych, odnosi się do ich fizycznego podobieństwa do wywołań metody, a także tego, jak w języku Ruby określany jest zamiar użytkownika, gdy zastosuje on zwykły identyfikator.

### 2.5.4. Zmienne lokalne i przypominające je elementy

Gdy interpreter języka Ruby napotka w kodzie zwykłe słowo lub identyfikator, taki jak `s`, `ticket`, `puts` lub `user_name`, interpretuje to jako jeden z trzech następujących elementów:

- zmienna lokalna,
- słowo kluczowe,
- wywołanie metody.

*Słowa kluczowe* to specjalne słowa zastrzeżone, których nie można użyć jako nazw zmiennych. `def` to słowo kluczowe. Może ono zostać zastosowane tylko do rozpoczęcia definicji metody (mówiąc wprost, możesz „oszukać" interpreter języka Ruby, który dopuści nadanie metodzie nazwy `def`, ale lepiej tego nie robić). `if` to również słowo kluczowe. Spora część kodu Ruby uwzględnia klauzule warunkowe, które rozpoczynają się słowem kluczowym `if`, dlatego zezwolenie też na użycie go jako nazwy zmiennej mogłoby wprowadzić zamieszanie. Analizowanie sekwencji takiej jak `if = 3` byłoby utrudnione dla interpretera języka Ruby.

Podobnie jak zmienne lokalne wywołania metody mogą być zwykłymi słowami. Zaprezentowano kilka przykładów, w tym wykorzystujące metody `puts` i `print`. Jeśli wywołanie metody uwzględnia argumenty w nawiasach okrągłych, a nawet puste

nawiasy, oczywiste jest, że nie jest to zmienna lokalna. W innych przypadkach może wystąpić niejednoznaczność, którą interpreter języka Ruby musi wyjaśnić.

Oto w jaki sposób interpreter języka Ruby decyduje, co zobaczy w momencie napotkania zwykłego identyfikatora:

1. Jeśli identyfikator to słowo kluczowe, za takie zostanie uznany (w języku Ruby znajduje się wewnętrzna lista takich słów, które są rozpoznawane).
2. Jeśli po prawej stronie identyfikatora znajduje się znak równości (=), jest to zmienna lokalna objęta przypisaniem.
3. W przeciwnym razie przyjmowane jest, że identyfikator będzie wywołaniem metody.

Jeśli użyjesz identyfikatora, który nie spełnia żadnego z tych trzech punktów, interpreter języka Ruby wyświetli ostrzeżenie i wstrzyma wykonywanie, generując błąd krytyczny. Uzyskany w takiej sytuacji komunikat o błędzie jest pouczający:

```
$ ruby -e "x"
-e:1:in `<main>': undefined local variable or method 'x' for main:Object
(NameError)
```

Zauważ, że interpreter języka Ruby nie może określić, czy stosując `x`, użytkownik miał na myśli zmienną, czy metodę. Interpreter „wie", że `x` nie jest słowem kluczowym, ale może to być zmienna lub metoda. Z tego powodu komunikat o błędzie uwzględnia oba elementy.

Na tym etapie dysponujesz dużym zasobem wiedzy dotyczącej obiektów i zmiennych, a także tego, jak są one ze sobą powiązane. W rozdziale 3. zajmiemy się tym, jak za pomocą klas tworzyć obiekty w sposób zapewniający strukturę i skalowalność.

## 2.6. Podsumowanie

W rozdziale 2. przedstawiono wiele podstawowej wiedzy. W rozdziale zaprezentowano następujące zagadnienia:

- Sposób tworzenia nowego obiektu i definiowania dla niego metod.
- Podstawy dotyczące mechanizmu wysyłania komunikatów, który służy do wysyłania żądań do obiektów w celu uzyskania informacji lub wykonania działania.
- Kilka ważnych metod wbudowanych, które są oferowane przez każdy obiekt języka Ruby: `object_id`, `respond_to?` i `send`.
- Szczegóły składni list argumentów metody, w tym użycie argumentów wymaganych, argumentów opcjonalnych i argumentów z wartością domyślną.
- Sposób działania zmiennych lokalnych i przypisań do zmiennych.
- Zastosowanie w języku Ruby odwołań do obiektów oraz sposób działania odwołań, gdy wiele zmiennych odwołuje się do tego samego obiektu.

Pisanie programu Ruby może uwzględniać zastanowienie się nad możliwym sposobem odwzorowania elementów domeny (może to być nawet skromna jednoelementowa domena, taka jak „bilet na wydarzenie") na system obiektów, aby mogły one przechowywać informacje i wykonywać zadania. Jednocześnie ważne jest, aby nie traktować

zbyt ściśle relacji między obiektami i rzeczywistym światem. Języki obiektowe z pewnością oferują rozbudowany komponent modelowania bazującego na rzeczywistym świecie. Jednocześnie jednak język Ruby jest wyjątkowo elastyczny pod względem własnych ułatwień związanych z modelowaniem, o czym możesz się przekonać na podstawie tego, jak łatwe jest rozszerzanie działania danego obiektu. Głównym celem podczas projektowania programu i zawartych w nim obiektów jest zapewnienie działającego systemu, który cechuje się wewnętrzną spójnością.

Oczywiście język oferuje mnóstwo ułatwień służących do tworzenia struktury programu. Tworzenie obiektów po kolei, tak jak zostało to zaprezentowane w rozdziale, stanowi trochę więcej niż wierzchołek góry lodowej. W dalszej części książki w znacznym stopniu rozszerzymy omawianie zagadnienia, przybliżając sposoby tworzenia wielu obiektów za pomocą klas języka Ruby z wykorzystaniem większego stopnia automatyzacji.

# 3 Organizowanie obiektów za pomocą klas

**W tym rozdziale:**

- Tworzenie wielu obiektów za pomocą klas.
- Ustawianie i odczytywanie stanu obiektu.
- Automatyzowanie tworzenia metod odczytujących i zapisujących atrybuty.
- Mechanizmy dziedziczenia klas.
- Składnia i semantyka stałych języka Ruby.

Tworzenie nowego obiektu za pomocą metody `Object.new`, a także zapewnianie mu jego własnych metod (po jednej na raz) to znakomity sposób zaznajomienia się z obiektowością towarzyszącą programowaniu w języku Ruby. Jednak takie rozwiązanie nie jest właściwie skalowane. Jeśli prowadzisz internetową kasę biletową i baza danych musi przetwarzać setki rekordów biletów, musisz znaleźć inny sposób tworzenia i modyfikowania we własnych programach Ruby obiektów podobnych do obiektów biletu.

Z całą pewnością język Ruby zapewnia pełny zestaw technik programistycznych służących do tworzenia obiektów w trybie wsadowym. Nie musisz definiować dla każdego obiektu biletu osobnej metody `price`. Zamiast tego możesz zdefiniować *klasę* obiektów biletów mającą taką konstrukcję, która powoduje, że każdy obiektu biletu automatycznie uzyskuje metodę `price`.

Definiowanie klasy pozwala grupować zachowania (metody) w ramach wygodnych pakietów. Dzięki temu możesz szybko utworzyć wiele obiektów, które w zasadzie działają w taki sam sposób. Możliwe jest też dodanie metod do poszczególnych obiektów,

jeśli okaże się to odpowiednie w odniesieniu do tego, co próbujesz zrealizować we własnym programie. Nie jest to jednak konieczne w przypadku każdego obiektu, jeśli modelujesz domenę za pomocą klas.

Wszystko, co jest obsługiwane w języku Ruby, ma postać obiektu lub konstrukcji, która jest przekształcana w obiekt. Każdy obiekt jest instancją pewnej klasy. Fakt ten jest prawdziwy nawet w sytuacjach, w których początkowo może się to wydać trochę dziwne. Liczby całkowite są instancjami klasy, a same klasy to obiekty. W rozdziale dowiesz się, na czym polega ten dominujący aspekt projektu języka Ruby.

Omawianie klas nie oznacza, że nie ma mowy o obiektach. Z tego właśnie powodu niniejszy rozdział ma taki tytuł, a nie na przykład „Klasy języka Ruby". Spora część rozdziału dotyczy obiektów i metod, co jednak wynika z tego, że klasy przede wszystkim służą do organizowania obiektów i metod. Przyjrzymy się rodzajom możliwych działań, jakie będą realizowane w obrębie klas, a także wyjaśnimy, czym są same klasy.

## 3.1. Klasy i instancje

Typowa klasa składa się z kolekcji definicji metod. Klasy istnieją zwykle w celu wykonywania procesu *tworzenia instancji*, czyli tworzenia obiektów, która są instancjami klasy.

Tworzenie instancji zostało już wcześniej zaprezentowane. Mowa o poczciwym kodzie wprowadzającym:

```
obj = Object.new
```

`Object` to klasa wbudowana języka Ruby. Użycie znaku kropki po nazwie klasy oznacza wysyłanie komunikatu do klasy. Tak jak obiekty, również klasy mogą odpowiadać na komunikaty. Okazuje się, że klasy są obiektami, i warto mieć tego świadomość w każdej sytuacji. Metoda `new` to *konstruktor*. Celem tej metody jest tworzenie nowej instancji klasy (nowo generowanego obiektu) i zwracanie jej użytkownikowi.

Klasa jest definiowana za pomocą słowa kluczowego `class`. Klasy są nazywane przy użyciu *stałych*, czyli specjalnego typu identyfikatora rozpoznawanego dzięki temu, że rozpoczyna się od dużej litery. Stałe służą do przechowywania informacji i wartości, które nie zmieniają się w czasie działania programu.

> **OSTRZEŻENIE** Stałe mogą się zmieniać. Wbrew nazwie stałe nie są niezmienne. Jeśli jednak przypiszesz do stałej nową wartość, interpreter języka Ruby wyświetli ostrzeżenie. Najlepszym rozwiązaniem jest unikanie przypisywania nowych wartości do stałych, do których została już przypisana wartość (w punkcie 3.7.2 zamieszczono więcej informacji o ponownym przypisywaniu do stałych).

Zdefiniujmy klasę `Ticket`. W obrębie definicji klasy definiujemy następującą prostą metodę:

```
class Ticket
  def event                                  ←—— Definiowanie metody event dla klasy Ticket
    "Tak naprawdę nie może być jeszcze określona..."
```

```
  end
end
```

Możemy teraz utworzyć nowy obiekt `ticket` i zażądać od niego (choć nie ma to żadnej większej wartości, pozwala zilustrować proces) opisania swojego wydarzenia:

```
ticket = Ticket.new
puts ticket.event
```

Wywołanie metody `ticket.event` powoduje wykonanie metody `event`, a w konsekwencji wyświetlenie łańcucha (nie zawiera raczej zbyt wielu informacji) podanego w obrębie tej metody:

```
Tak naprawdę nie może być jeszcze określona...
```

Informacja jest mało precyzyjna, ale proces jest w pełni wykonywalny: utworzyliśmy i wykonaliśmy metodę instancji.

Jakie dokładnie ma to znaczenie?

### 3.1.1. Metody instancji

Przykłady definicji metod z rozdziału 2. uwzględniały definiowanie metod bezpośrednio dla poszczególnych obiektów:

```
def ticket.event
```

Metoda `event` w poprzednim przykładzie jest jednak definiowana w ogólny sposób w obrębie klasy `Ticket`:

```
def event
```

Wynika to stąd, że ta metoda `event` będzie współużytkowana przez wszystkie bilety, czyli przez wszystkie instancje klasy `Ticket`. Tego rodzaju metody definiowane wewnątrz klasy i przeznaczone do użycia przez wszystkie instancje klasy są nazywane *metodami instancji*. Nie należą one tylko do jednego obiektu. Mogą być wywoływane przez dowolną instancję klasy.

> **UWAGA** Metody definiowane dla jednego konkretnego obiektu, tak jak w przypadku definicji `def ticket.price`, są nazywane *metodami pojedynczego obiektu*. Wcześniej zaprezentowano odpowiednie przykłady. W rozdziale 13. obszerniej przyjrzymy się działaniu tego typu metod. Dla obiektu z metodą `price` obojętne jest to, czy jest ona nazywana metodą pojedynczego obiektu, czy metodą instancji jego klasy. Rozróżnienie jest jednak ważne z punktu widzenia programisty.

Po zdefiniowaniu metody instancji w klasie nic nie powstrzyma Cię przed ponownym zdefiniowaniem jej, czyli przesłonięciem pierwszej definicji nową.

### 3.1.2. Przesłanianie metod

Oto przykład dwukrotnego definiowania tej samej metody w jednej klasie:

```
class C
  def m
    puts "Pierwsza definicja metody m"
```

```
  end
def m
    puts "Druga definicja metody m"
  end
end
```

Co będzie mieć miejsce w przypadku tych dwóch definicji, gdy metoda `m` zostanie wywołana w instancji klasy `C`? Zapytajmy o to obiekt:

```
C.new.m
```

Wyświetlony wynik to łańcuch `Druga definicja metody m`. Druga definicja „zwyciężyła": widoczne są dane wyjściowe z tej definicji, a nie z pierwszej. W momencie przesłonięcia metody pierwszeństwo będzie mieć nowa wersja.

Poprzedni przykład świadomie uproszczono, ponieważ ilustruje on coś, co normalnie nie byłoby realizowane dokładnie w takiej postaci. Przesłanianie metody jest zwykle spowodowane tym, że utworzono klasę, która dziedziczy z oryginalnej klasy, a pożądane jest, aby klasa działała inaczej. Wkrótce zajmiemy się zagadnieniem dziedziczenia.

Możliwe jest poszerzanie metod klasy lub przesłanianie ich przez ponowne otwieranie definicji klasy.

### 3.1.3. Ponowne otwieranie klas

W większości przypadków podczas definiowania klasy tworzysz następujący pojedynczy blok definicji klasy:

```
class C
  # w tym miejscu znajduje się kod klasy
end
```

Możliwe jest jednak ponowne otwarcie klasy, a następnie poszerzenie jej lub wprowadzenie zmian. Oto przykład:

```
class C
  def x
  end
end
class C
  def y
  end
end
```

Otwieramy treść definicji klasy, dodajemy jedną metodę (`x`) i zamykamy tę treść. W dalszej kolejności ponownie otwieramy treść definicji, dodajemy drugą metodę (`y`) i zamykamy treść. Wynik jest taki sam jak dla następującego kodu:

```
class C
  def x
  end
  def y
  end
end
```

W przypadku tego kodu definicja klasy jest otwierana tylko raz i dodawane są obie metody. Oczywiście definicje klasy nie są rozdzielane na osobne bloki tylko dla

zabawy. Musi być ku temu powód, i to dobry, ponieważ rozdzielanie definicji klas może utrudnić programistom czytanie kodu w celu prześledzenia, co ma miejsce, lub używanie go.

Jednym z powodów rozdzielania definicji klas jest rozmieszczenie ich w wielu plikach. Jeśli za pomocą metody `require` żądasz pliku, który zawiera definicję klasy (być może ładujesz ją z innego pliku na dysku w czasie działania programu, a ponadto częściowa definicja tej samej klasy istnieje też w pliku, z poziomu którego wymagany jest drugi plik), dwie definicje są scalane. Nie jest to proces, jaki można przeprowadzić w dowolny sposób: musi wystąpić sytuacja, w której projekt programu wymaga, aby klasa została częściowo zdefiniowana w jednym miejscu, a częściowo w drugim.

Pora na praktyczny przykład. Język Ruby oferuje klasę `Time`. Umożliwia ona modyfikowanie czasu, formatowanie go na potrzeby znaczników czasu itp. W celu uzyskania żądanego formatu możesz zastosować łańcuchy formatu daty w stylu systemu Unix. Na przykład polecenie:

```
puts Time.new.strftime("%m-%d-%y")
```

wyświetla łańcuch `"02-09-14"` reprezentujący datę użycia wywołania metody.

Oprócz klasy wbudowanej `Time` język Ruby oferuje również plik programu o nazwie *time.rb*, w obrębie którego znajdują się różne rozszerzenia i dodatki klasy `Time`. Plik ten osiąga swój cel, rozszerzając klasę `Time` przez jej ponowne otwarcie. Jeśli poszukasz pliku *time.rb* w podkatalogu *lib* drzewa kodu źródłowego Ruby lub w instalacji tego języka, w wierszu 87. lub w wierszu znajdującym się w jego pobliżu zobaczysz następujący kod:

```
class Time
```

Kod powoduje ponowne otwarcie klasy `Time` w celu dodania nowych metod.

Aby zobaczyć efekt, najlepiej spróbować wykonać ten kod w narzędziu irb. Umożliwia ono wywołanie nieistniejącej metody bez powodowania zakończenia sesji. Dzięki temu w jednej sesji możesz ujrzeć efekty zastosowania polecenia `require`:

```
>> t = Time.new
=> 2014-02-09 09:41:29 -0500
>> t.xmlschema
NoMethodError: undefined method 'xmlschema' for 2014-02-09 09:41:29 -  ❶
     0500:Time
     from (irb):2
     from /Users/dblack/.rvm/rubies/ruby-2.1.0/bin/irb:11:in `<main>'
>> require 'time'                                                    ❷
=> true
>> t.xmlschema
=> "2014-02-09T09:41:29-05:00"
```

W kodzie nierozpoznany komunikat `xmlschema` wysyłany jest do obiektu `Time`. Operacja ta kończy się niepowodzeniem ❶. Później ładowany jest plik *time.rb* ❷. Z całą pewnością obiekt `Time` zawiera teraz metodę `xmlschema` (zgodnie z jej dokumentacją zwraca ona łańcuch, który reprezentuje czas jako datę i godzinę zdefiniowane przez schemat XML).

Kod jednej klasy możesz rozmieścić w wielu plikach lub miejscach w obrębie tego samego pliku. Bądź jednak świadom, że bardziej zalecane jest niedecydowanie się na coś takiego, jeśli to możliwe. W przypadku rozszerzeń obiektu `Time` często sugerowana jest możliwość unifikacji, czyli przede wszystkim zapewnienie obiektom `Time` wszystkich metod rozszerzeń, a nie wydzielanie tych metod do osobnej biblioteki. Możliwe jest, że taka unifikacja pojawi się w późniejszej wersji języka Ruby.

Język Ruby powiązany jest z obiektami, które są instancjami klasy. W dalszej kolejności zajmiemy się zmiennymi instancji, czyli specjalnym elementem języka przewidzianym do tego, aby każda instancja każdej klasy języka Ruby mogła ustawiać i utrzymywać własny, prywatny schowek z informacjami.

## 3.2. Zmienne instancji i stan obiektu

Po utworzeniu poszczególnych obiektów i wymaganych metod dla każdego działania lub wartości za pośrednictwem metod na stałe umieściliśmy wartość w obiekcie. Jeśli w przypadku tego rozwiązania bilet kosztuje 117,50 zł, obiekt biletu zawiera metodę o nazwie `price`, która zwraca dokładnie taką kwotę:

```
ticket = Object.new
def ticket.price
  117.50
end
```

Zamiast realizowaniem procesu tworzenia po jednym obiekcie na raz za pomocą metody `Object.new` zajmiemy się teraz praktyczną stroną projektowania klas i tworzenia wielu obiektów z ich wykorzystaniem.

Oznacza to, że zmieniamy reguły gry, mając do czynienia z takimi informacjami jak cena biletu. Jeśli tworzysz klasę `Ticket`, nie możesz zapewnić jej metody `price`, która zwraca cenę 117,50 zł. Wynika to z prostego powodu, a mianowicie takiego, że nie każdy bilet ma taką cenę. Nie możesz też powiązać z każdym biletem wydarzenia o nazwie koncert dobroczynny ani nie może być tak, że każdy bilet jest dla miejsca 33. w rzędzie G.

Zamiast wprowadzania na stałe wartości do każdego obiektu niezbędne jest rozwiązanie informujące różne obiekty, że mają inne wartości. Musi być możliwe utworzenie nowego obiektu `Ticket` i przechowywanie razem z nim informacji o wydarzeniu, cenie i innych właściwościach. Podczas tworzenia kolejnego obiektu biletu konieczne jest przechowywanie z nim innych informacji. Wymagana jest też możliwość osiągnięcia tego bez potrzeby tworzenia metody z umieszczoną w niej na stałe właściwością.

Informacje i dane powiązane z konkretnym obiektem stanowią *stan* obiektu. Niezbędna jest możliwość wykonania następujących działań:

- Ustawianie lub resetowanie stanu obiektu (przekazanie obiektowi biletu informacji: „kosztujesz 11,99 zł”).
- Ponowne wczytywanie stanu (zadanie obiektowi biletu pytania: „ile kosztujesz?”).

Tak się składa, że obiekty języka Ruby dysponują własnym mechanizmem przechowywania i pobierania wartości. Mowa o *zmiennych instancji*.

Zmienna instancji umożliwia poszczególnym obiektom zapamiętywanie stanu. Zmienne instancji działają bardzo podobnie do innych zmiennych: są im przypisywane wartości, które są ponownie wczytywane, dodawane do siebie, wyświetlane itd. Jednak w przypadku zmiennych instancji występuje kilka następujących różnic:

- Nazwy zmiennych instancji zawsze rozpoczynają się pojedynczym znakiem `@`. Umożliwia to natychmiastowe rozpoznanie takich zmiennych.
- Zmienne instancji są widoczne tylko w obiekcie, do którego należą (termin „widoczne w obiekcie" ma techniczną definicję mającą związek z domyślnym obiektem `self`, o którym więcej informacji zamieszczono w rozdziale 5.).
- Zmienna instancji zainicjowana w jednej metodzie w obrębie klasy może być używana przez dowolną metodę zdefiniowaną wewnątrz tej klasy.

Listing 3.1 zawiera prosty przykład ilustrujący, w jaki sposób przypisana wartość zmiennej instancji pozostaje aktywna między kolejnymi wywołaniami metody.

**Listing 3.1. Zmienna instancji utrzymująca swoją wartość między wywołaniami metody**

```
class Person
  def set_name(string)
    puts "Ustawianie imienia osoby..."
    @name = string                  ←❶
  end
  def get_name
    puts "Zwracanie imienia osoby..."
    @name
  end
end
joe = Person.new
joe.set_name("Jan")             ←❷
puts joe.get_name               ←❸
```

Dzięki przypisaniu ❶ będącemu wynikiem wywołania metody `set_name` ❷ po zapytaniu o imię osoby ❸ w odpowiedzi uzyskasz to, czego oczekujesz, czyli łańcuch `"Jan"`. W przeciwieństwie do zmiennej lokalnej zmienna instancji `@name` zachowuje przypisaną jej wartość nawet po zakończeniu metody, w której zmienna została zainicjowana. Taka właściwość zmiennych instancji (zachowywanie między wywołaniami metod) sprawia, że są odpowiednie do utrzymywania stanu w obiekcie.

Wkrótce poznasz lepsze i bardziej idiomatyczne sposoby przechowywania i pobierania wartości w obiektach. Wszystkie one bazują jednak na ustawianiu i pobieraniu wartości zmiennych instancji, dlatego warto dobrze zorientować się, jak działają zmienne instancji.

Podejmowane działania będą mieć na celu zrealizowanie czegoś przydatnego z wykorzystaniem klasy `Ticket`. Brakujący krok, który zostanie teraz omówiony, to proces inicjowania obiektu.

### *3.2.1. Inicjowanie obiektu ze stanem*

Tworząc klasę (np. `Ticket`), w razie potrzeby możesz zdefiniować specjalną metodę o nazwie `initialize`. W takiej sytuacji metoda ta będzie wykonywana każdorazowo przy tworzeniu nowej instancji klasy.

Na przykład dla następującej metody `initialize` wyświetlającej komunikat:

```
class Ticket
  def initialize
    puts "Tworzenie nowego biletu!"
  end
end
```

zostanie pokazany komunikat `"Tworzenie nowego biletu!"` za każdym razem podczas tworzenia nowego obiektu biletu przez wywołanie metody `Ticket.new`.

Taki zautomatyzowany proces inicjowania możesz zastosować do ustawienia stanu obiektu w momencie jego tworzenia. Załóżmy, że podczas tworzenia każdego obiektu biletu chcemy powiązać z nim miejsce i datę. Wysyłamy poprawne wartości jako argumenty do metody `Ticket.new`. Te same argumenty zostaną automatycznie wysłane do metody `initialize`. W jej obrębie uzyskamy dostęp do informacji o miejscu wydarzenia i dacie, które możemy zapisać za pomocą zmiennych instancji:

```
class Ticket
  def initialize(venue,date)
    @venue = venue
    @date = date
  end
```

Przed zamknięciem definicji klasy przy użyciu dyrektywy `end` należy dodać coś jeszcze, czyli rozwiązanie umożliwiające ponowne odczytanie informacji o miejscu i dacie. Zrezygnujmy z nazwy metody w postaci z przedrostkiem `get_` użytym dla metody `get_name` (listing 3.1) na rzecz nazwy nadawanej metodom pobierającym zgodnie z nazwą zmiennych instancji, których wartości są zwracane przez te metody. Do wcześniejszych wierszy kodu dodaj następujący kod (uwzględnia on dyrektywę `end` dla definicji klasy):

```
def venue
    @venue
  end
  def date
    @date
  end
end
```

Każda z tych metod przekazuje wartość zmiennej instancji. W każdym przypadku zmienna ta jest ostatnim (i jedynym) wyrażeniem w metodzie, dlatego też spełnia rolę jej wartości zwracanej.

> **UWAGA** Nazwy zmiennych instancji, metod i argumentów metody `initialize` nie muszą być zgodne. Zamiast nazwy `@venue` możesz użyć nazwy `@v`, aby na przykład zapisać wartość przekazywaną w argumencie `venue`. Możesz wywołać

drugą metodę `event_date` i nadal używać w niej nazwy `@date`. Niemniej jednak zwykle dobrą praktyką jest zachowanie zgodności nazw w celu zapewnienia przejrzystości odnośnie do tego, co jest z czym powiązane.

Jesteśmy teraz gotowi do utworzenia kilku biletów za pomocą dynamicznie ustawianych wartości dla argumentów miejsca i daty wydarzenia, a nie przez określanie na stałe wartości w kodzie, jak to miało miejsce we wcześniejszych przykładach:

```
th = Ticket.new("Sala Miejska", "11/12/13")
cc = Ticket.new("Centrum Kongresowe", "12/13/14")
puts "Utworzono dwa bilety."
puts "Pierwszy dotyczy wydarzenia w miejscu: #{th.venue}. Data: #{th.date}."
puts "Drugi dotyczy wydarzenia o dacie: #{cc.date}. Miejsce: #{cc.venue}."
```

Po uruchomieniu powyższego kodu wraz z wcześniejszą definicją klasy obiektu `Ticket` zostaną uzyskane następujące dane wyjściowe:

```
Utworzono dwa bilety.
Pierwszy dotyczy wydarzenia w miejscu: Sala Miejska. Data: 11/12/13.
Drugi dotyczy wydarzenia o dacie: 12/13/14. Miejsce: Centrum Kongresowe.
```

Fraza `Miejsce: Centrum Kongresowe` brzmi trochę zbyt poważnie, ale dzięki zmiennym instancji proces zapisywania i pobierania informacji dla poszczególnych obiektów przebiega doskonale. Każdy obiekt biletu ma własny stan (zapisane informacje), co zawdzięcza temu, co realizuje metoda `initialize`. Każdy taki obiekt umożliwia skierowanie do niego zapytania dotyczącego miejsca i daty, co jest możliwe dzięki metodom `venue` i `date`.

Zapewnia to ogromne perspektywy. W tym samym czasie możemy tworzyć, modyfikować, porównywać i sprawdzać dowolną liczbę obiektów biletów bez konieczności tworzenia osobnych metod dla każdego z nich. Wszystkie te obiekty współużytkują zasoby klasy `Ticket`. Jednocześnie każdy obiekt biletu ma własny zestaw zmiennych instancji służących do przechowywania informacji o stanie.

Do tej pory działania były realizowane w ten sposób, że wartości zmiennych instancji były ustawiane w chwili tworzenia obiektu, a ponadto wartości te mogły później zostać pobrane w dowolnym momencie istnienia obiektu. Takie rozwiązanie często jest odpowiednie, ale niesymetryczne. Co będzie, gdy zechcesz ustawić wartości zmiennych instancji w innym momencie niż podczas tworzenia obiektu? Co będzie, jeśli postanowisz zmienić stan obiektu, gdy już został raz ustawiony?

## 3.3. Metody ustawiające

Gdy musisz ustawić lub zmienić stan obiektu w innym miejscu programu niż metoda `initialize`, kluczowe znaczenie ma przypisanie (lub ponowne przypisanie) wartości do zmiennych instancji. Możesz oczywiście zmienić wartość każdej zmiennej instancji w dowolnej metodzie. Jeśli na przykład bilety mają oferować możliwość automatycznego uwzględniania upustu, można utworzyć metodę instancji podobną do następującej metody znajdującej się wewnątrz definicji klasy `Ticket`:

```
def discount(percent)
  @price = @price * (100 - percent) / 100.0
end
```

Okazuje się jednak, że najbardziej typowy przypadek jest najprostszy: wywołanie metody ustawiającej z argumentem i ustawienie dla argumentu odpowiedniej zmiennej instancji. Właśnie to realizuje metoda `set_name` w przykładzie klasy `Person`.

To jednak nie wszystko. Język Ruby oferuje specjalne konwencje dotyczące nadawania nazw metodom, umożliwiające tworzenie metod ustawiających w sposób, który jest bardziej elegancki niż umieszczanie przedrostka `set_` na początku opisowego słowa, takiego jak *nazwa*. Ponownie wykorzystamy klasę `Ticket`, ale tym razem skoncentrujemy się na metodach ustawiających i dostępnych technikach, które je ulepszają.

### 3.3.1. Znak równości (=) w nazwach metod

Załóżmy, że pożądany jest sposób ustawiania ceny biletu. Na początku cena może być ustawiana wraz z innymi informacjami w momencie tworzenia obiektu:

```
class Ticket
  def initialize(venue,date,price)
    @venue = venue
    @date = date
    @price = price
  end
  # itd.
def price
    @price
  end
  # itd.
end
th = Ticket.new("Sala Miejska", "11/12/13", 63.00)
```

Polecenie inicjalizacji staje się jednak wyjątkowo długie, a ponadto wymaga pamiętania, w jakiej kolejności ma zostać wstawionych wiele argumentów, a ostatecznie i tak nie uzyskuje się ceny dla biletu `"Sala Miejska"`. Poza tym w dalszym ciągu nie mamy możliwości późniejszej zmiany ceny biletu.

Rozwiążmy początkowo problem za pomocą metody `set_price`, która umożliwia ustawianie lub zmienianie ceny istniejącego biletu. Zostanie również przebudowana metoda `initialize`, tak aby nie oczekiwała kwoty ceny:

```
class Ticket
  def initialize(venue, date)
    @venue = venue
    @date = date
  end
  def set_price(amount)
    @price = amount
  end
  def price
    @price
  end
end
```

Oto kilka operacji zmiany ceny w praktyce:

```
ticket = Ticket.new("Sala Miejska", "11/12/13")
ticket.set_price(63.00)
puts "Cena biletu wynosi #{"%.2f" % ticket.price} zł."
ticket.set_price(72.50)
puts "Ojej! -- to po prostu zadziałało. Cena biletu wynosi teraz #{"%.2f" % ticket.price}
zł."
```

**Formatowanie ceny do dwóch miejsc po przecinku**

Oto dane wyjściowe:

```
Cena biletu wynosi 63 zł.
Ojej! -- to po prostu zadziałało. Cena biletu wynosi teraz 72.50 zł.
```

Takie rozwiązanie działa: możesz tworzyć wszystkie niezbędne metody `set_property` oraz oparte na zmiennych instancji metody pobierania, które zostaną użyte razem z tymi metodami. Dostępny jest jednak lepszy sposób.

> **WSKAZÓWKA** Technika wykorzystująca znak procentu zaprezentowana w ostatnim przykładzie umożliwia formatowanie łańcuchów za pomocą składni przypominającej składnię metody `sprintf`. Język Ruby też oferuje tę metodę `sprintf` (dostępna również przy formatowaniu nazw). Możliwe jest przebudowanie przykładu z ceną biletu przy użyciu kodu `sprintf("%.2f", ticket.price)`. Dostępne specyfikatory formatu (elementy ze znakiem `%` w obrębie łańcucha wzorca) obejmują specyfikatory `%d` (liczby dziesiętne), `%s` (łańcuchy), `%f` (liczby zmiennoprzecinkowe) i `%x` (liczby szesnastkowe). W celu uzyskania pełnej dokumentacji wykonaj polecenie `ri sprintf`.

Język Ruby pozwala na definiowanie metod zakończonych znakiem równości (`=`). Zastąpmy metodę `set_price` metodą o nazwie `price=` (nazwa `price` oraz znak równości):

```
def price=(amount)
  @price = amount
end
```

Metoda `price=` realizuje dokładnie to samo co metoda `set_price`. Pomimo trochę dziwnej nazwy metoda może zostać wywołana tak jak dowolna inna metoda:

```
ticket.price=(63.00)
```

Znak równości zapewnia znajome poczucie „przypisywania wartości do czegoś”, dlatego wiesz, że masz do czynienia z metodą ustawiającą. Choć jednak to nadal wygląda dziwnie, interpreter języka Ruby również tym się zajmuje.

### 3.3.2. „Lukier” składniowy dla metod przypominających przypisania

Programiści używają terminu *„lukier” składniowy* w odniesieniu do specjalnych reguł, które umożliwiają pisanie kodu w sposób niezgodny ze standardowymi regułami, ale lepiej prezentujący się i łatwiejszy do zapamiętania.

Język Ruby oferuje *„lukier” składniowy* w przypadku wywoływania metod ustawiających. Zamiast kodu:

```
ticket.price=(63.00)
```

możesz użyć następującego kodu:

```
ticket.price = 63.00
```

Gdy interpreter napotka taką sekwencję kodu, automatycznie zignoruje spację przed znakiem równości i wczyta łańcuch `price =` jako pojedynczy komunikat `price=` (wywołanie metody o nazwie `price=`, która została zdefiniowana). Jeśli chodzi o prawą stronę, nawiasy okrągłe są opcjonalne w przypadku argumentów metody, pod warunkiem że nie ma miejsca niejednoznaczność. Oznacza to, że możesz po prawej stronie wstawić wartość 63.00, która zostanie pobrana jako argument metody `price=`.

Celem włączenia takiej składni specjalnej jest umożliwienie tworzenia wywołań metody przypominających przypisania. Jeśli w programie napotkano właśnie wiersz `ticket.price = 63.00`, możesz przyjąć, że `ticket.price` to pewnego rodzaju wartość *l*, do której przypisywana jest wartość 63.00. Tak jednak nie jest. Cały ten wiersz stanowi wywołanie metody. Jest to odbiorca `ticket`, metoda `price=` i pojedynczy argument `63.00`.

Im więcej będziesz korzystać z tego stylu metody opartego na metodach ustawiających, tym bardziej będziesz przekonany, że dużo lepiej prezentuje się wersja z „lukrem" składniowym. Tego rodzaju zwracanie uwagi na wygląd jest typowe dla języka Ruby.

Pamiętaj też o tym, że metody ustawiające oferują większe możliwości niż zwykłe przypisanie zmiennej.

### 3.3.3. Pełnia możliwości metod ustawiających

Możliwość tworzenia własnych metod o nazwach zakończonych znakiem równości, a także to, że język Ruby zapewnia bazujący na wykorzystaniu „lukru" składniowego sposób wywoływania tych metod, stwarza kilka interesujących opcji.

Jedną z nich jest nadużywanie. Możliwe jest tworzenie metod ze znakiem równości na końcu ich nazwy. Metody takie sprawiają wrażenie, że mają zrealizować coś uwzględniającego przypisanie, ale tak nie jest:

```
class Silly
  def price=(x)
    puts "Bieżący czas: #{Time.now}"
  end
end
s = Silly.new
s.price = 111.22
```

W tym przykładzie usuwany jest odbierany argument (`111.22`), a następnie wyświetlany jest niepowiązany komunikat:

```
Bieżący czas: 2014-02-09 09:53:31 -0500
```

Przykład ten świadomie skarykaturyzowano. Wynikający z niego wniosek jest jednak ważny: interpreter języka Ruby sprawdza składnię, ale nie nadzoruje stosowanej semantyki. Masz możliwość tworzenia metod o nazwach zakończonych znakiem =, a ponadto zawsze dysponujesz opcją „lukru" składniowego dla metod przypominających przypisania. Od Ciebie zależy to, czy nazwa metody ma sens w odniesieniu do jej przeznaczenia.

Metody o nazwach zawierających znak równości mogą też pełnić rolę filtrów lub strażników. Załóżmy, że chcemy ustawić cenę biletu tylko wtedy, gdy ma ona sens jako kwota wyrażona w złotówkach i groszach. Aby zapewnić poprawność danych, możemy dodać „element inteligencji" do metody `price=`. W tym przypadku liczba zostanie pomnożona przez 100, w ramach operacji realizowanej przez metodę `to_i` (przekształca w liczbę całkowitą) zostaną obcięte wszelkie liczby dziesiętne pozostałe po przecinku, a ponadto wynik zostanie porównany z oryginalną liczbą pomnożoną przez 100. Powinno to ujawnić wszystkie dodatkowe cyfry dziesiętne znajdujące się poza kolumną setnych:

```
class Ticket
  def price=(amount)
    if (amount * 100).to_i == amount * 100
      @price = amount
    else
      puts "Cena wydaje się być zniekształcona"
    end
  end
  def price
    @price
  end
end
```

Tego rodzaju technika filtrowania może też zostać wykorzystana do *normalizowania* danych, czyli do zapewniania, że określone dane zawsze przyjmą konkretną postać. Dla przykładu załóżmy, że w witrynie internetowej agenta biura podróży użytkownik musi wpisać żądaną datę wyjazdu. Pożądane jest zezwolenie zarówno na datę w formacie mm/dd/rr, jak i w formacie mm/dd/rrrr.

Jeśli na przykład dysponujesz skryptem CGI Ruby, który przetwarza dane wejściowe, możesz dokonać normalizacji roku przez utworzenie następującej metody ustawiającej:

```
class TravelAgentSession
  def year=(y)
    @year = y.to_i
  if @year < 100  <------ Modyfikowanie jedno- lub dwucyfrowej liczby przez dodanie do niej wieku
      @year = @year + 2000
    end
  end
end
```

Przyjmując następnie, że istnieje zmienna o nazwie `date`, w której zapisano pole daty z formularza (za pomocą biblioteki CGI języka Ruby), możesz uzyskać następujące składniki daty:

```
month, day, year = date.split('/')
self.year = year
```

Pomysł polega na podzieleniu łańcucha daty na trzy łańcuchy za pomocą znaku `/` jako rozdzielacza i metody wbudowanej `split`, a następnie zapisaniu wartości roku w obiekcie `TravelAgentSession` przy użyciu jego metody `year=`.

**OSTRZEŻENIE** Metody ustawiające nie zwracają tego, czego możesz oczekiwać. Gdy korzystasz z opcji „lukru" składniowego pozwalającej tworzyć wywołania metod o nazwie zakończonej znakiem `=`, które przypominają przypisania,

interpreter języka Ruby na poważnie zajmuje się semantyką przypisań. Wartością określaną dla przypisań (np. `x = 1`) jest to, co znajduje się po ich prawej stronie. Metody zwracają zwykle wartość ostatniego wyrażenia obliczanego podczas wykonywania. Jednak wywołania metod o nazwie ze znakiem = na końcu zachowują się podobnie do przypisań: wartość wyrażenia `ticket.price = 63.00` to `63.00`, nawet jeśli metoda `ticket=` zwraca łańcuch `"Ha ha!"`. Zamiarem jest utrzymanie spójności semantyki. Choć tak naprawdę jest to wywołanie metody, wygląda jak przypisanie i zachowuje się jak ono w odniesieniu do swojej wartości w postaci wyrażenia.

Czasami będziesz tworzyć złożone metody pobierające i ustawiające, ale najczęściej stosowane są proste operacje pobierania i ustawiania realizowane w obrębie zmiennych instancji. Operacje te są tak powszechne, że w języku Ruby zapewniono kilka uproszczonych sposobów definiowania ich.

## 3.4. Atrybuty i rodzina metod attr_*

*Atrybut* jest właściwością obiektu, której wartość może być odczytywana i/lub zapisywana za pośrednictwem obiektu. W przypadku obiektów biletu założyliśmy, że każdy taki obiekt ma atrybut `price`, a także atrybuty `date` i `venue`. Metodę `price=` można opisać jako metodę *zapisującą atrybuty*. `date`, `venue` i `price` (bez znaku równości w nazwie) to metody *odczytujące atrybuty* (terminologia związana z zapisem i odczytem odpowiada wcześniej użytej terminologii dotyczącej ustawiania i pobierania, ale częściej pojawia się w dyskusjach odnoszących się do języka Ruby).

Atrybuty obiektów języka Ruby są implementowane jako metody odczytujące i/lub zapisujące, które opakowują zmienne instancji. Można też powiedzieć, że zmienne te są opakowane metodami odczytującymi i/lub zapisującymi. Na poziomie języka nie istnieje osobna konstrukcja nazywana „atrybutem". *Atrybut* to ogólny termin dotyczący konkretnej konfiguracji metod i zmiennych instancji. Jest to jednak przydatne pojęcie osadzone w języku Ruby w postaci skrótów, które ułatwiają tworzenie metod implementujących atrybuty.

### 3.4.1. Automatyzowanie tworzenia atrybutów

Przyjrzyj się listingowi 3.2 prezentującemu całość tego, co zostało przedstawione do tej pory w omawianym przykładzie (z wykorzystaniem metod odczytujących i/lub zapisujących atrybuty) klasy `Ticket` (kod nie zawiera niczego nowego, a jedynie jego fragmenty zebrano w jednym miejscu).

**Listing 3.2. Klasa Ticket ze zdefiniowanymi metodami odczytującymi i/lub zapisującymi atrybuty**

```
class Ticket
  def initialize(venue, date)
    @venue = venue
    @date = date
  end
  def price=(price)
```

```
    @price = price
  end
  def venue
    @venue
  end
  def date
    @date
  end
  def price
    @price
  end
end
```

Istnieje jeden atrybut odczytu/zapisu (`price`) oraz dwa atrybuty odczytu (`venue` i `date`). Kod działa, ale zawiera powtarzające się fragmenty. Trzy metody mają następującą postać:

```
def coś
  @coś
end
```

Ponadto na jednym powtórzeniu bazuje inne powtórzenie: nie tylko występują trzy powyższe metody, ale nazwa każdej z nich pojawia się ponownie w nazwie zmiennej instancji, która z nich korzysta.

Każdorazowo w przypadku stwierdzenia powtórzeń o takiej skali należy podjąć próbę ograniczenia jej, i to nie przez zmniejszenie funkcjonalności programu, ale przez znalezienie sposobu bardziej zwięzłego wyrażania tego samego. W pogoni za taką zwięzłością język Ruby znajduje się o krok dalej: zapewnia wbudowany skrót, który powoduje automatyczne tworzenie metody odczytującej i zwracającej wartość zmiennej instancji o takiej samej nazwie co nazwa metody (z pominięciem lub dodaniem znaku `@`). Odpowiedni kod ma następującą postać:

```
class Ticket
  attr_reader :venue, :date, :price
end
```

Elementy rozpoczynające się znakiem dwukropka (`:venue` itd.) to *symbole*. Symbole są odmianą rozwiązania służącego do nadawania nazw lub etykiet. Choć są one spokrewnione z łańcuchami, nie do końca są tym samym. W rozdziale 8. obszerniej zajmiemy się symbolami. Na razie możesz je traktować jako funkcjonalne odpowiedniki łańcuchów.

**Obiekt self w roli odbiorcy domyślnego**

Występuje większa liczba wywołań metod bez jawnego odbiorcy. Na przykład w powyższym wierszu komunikatu `attr_reader` nie ma obiektu obecnego po lewej stronie znaku kropki i samej kropki. Pod nieobecność jawnego odbiorcy komunikaty trafiają do obiektu domyślnego `self`. Na najwyższym poziomie treści definicji klasy obiektem `self` jest sam obiekt klasy. A zatem obiekt odbierający komunikat `attr_reader` jest w rzeczywistości obiektem klasy `Ticket`. W dalszej części rozdziału dokładniej zostaną omówione klasy jako obiekty, a tym samym jako odbiorcy komunikatów. W rozdziale 5. obszerniej zostanie zaprezentowany obiekt `self`.

Metoda `attr_reader` odczytująca atrybuty tworzy automatycznie odmianę metody, jaką właśnie omówiliśmy. Istnieje również metoda `attr_writer`:

```
class Ticket
  attr_writer :price
end
```

W postaci tego jednego wiersza utworzyliśmy (lub raczej zajął się tym interpreter języka Ruby) metodę ustawiającą `price=`. Jeden wiersz zastępuje trzy. W przypadku metod odczytujących jeden wiersz zajął miejsce dziewięciu!

Listing 3.3 zawiera cały program w obecnej postaci.

**Listing 3.3. Klasa Ticket z metodami pobierającymi i ustawiającymi zdefiniowanymi za pomocą wywołań attr_***

```
class Ticket
  attr_reader :venue, :date, :price
  attr_writer :price
  def initialize(venue, date)
    @venue = venue
    @date = date
  end
end
```

Nie tylko krótszy jest kod z listingu 3.3, ale też zapewnia więcej informacji, które nawet mogą posłużyć do automatycznego tworzenia dokumentacji. Od razu widać, że każdy obiekt biletu zawiera argumenty `venue`, `date` i `price`. Pierwsze dwa atrybuty mogą zostać odczytane, natomiast atrybut `price` może być odczytany lub zapisany.

Możliwe jest nawet tworzenie metod odczytujących lub zapisujących przy użyciu jednego polecenia.

### TWORZENIE ATRYBUTÓW METOD ODCZYTUJĄCYCH LUB ZAPISUJĄCYCH ZA POMOCĄ METODY ATTR_ACCESSOR

W świecie atrybutów obiektów częstą sytuacją jest łączenie ze sobą atrybutów metod odczytujących/zapisujących (np. `price`). Język Ruby zapewnia jedną metodę `attr_accessor` przeznaczoną do tworzenia dla atrybutu zarówno metody odczytującej, jak i metody zapisującej. Metoda `attr_accessor` jest odpowiednikiem kombinacji metod `attr_reader` i `attr_writer`. Taka połączona technika może zostać użyta dla atrybutu `price`, ponieważ wymagane są obie operacje realizowane przez te metody:

```
class Ticket
  attr_reader :venue, :date
  attr_accessor :price
  # ... itp.
end
```

Alternatywnie funkcjonalność metody `attr_accessor` możesz uzyskać przy użyciu zwykłej metody `attr` w następujący sposób:

```
attr :price, true
```

Wywołanie metody `attr` z wartością `true` jako drugim argumentem powoduje zainicjowanie tworzenia atrybutów `reader` i `writer` jak w przypadku metody `attr_accessor`.

Jednak metoda ta cechuje się większą przejrzystością swojego przeznaczenia (słowo `accessor` wskazuje na operację dostępu), a ponadto oferuje korzyść polegającą na tym, że jednocześnie możesz przekazać jej więcej niż jedną nazwę akcesora (z kolei metoda `attr` pobiera tylko jeden argument oraz opcjonalny argument `true`). Bez drugiego argumentu `true` metoda `attr` zapewnia jedynie atrybut `reader`.

### 3.4.2. Podsumowanie metod attr_*

W tabeli 3.1 podsumowano rodzinę metod `attr_*`.

**Tabela 3.1.** Podsumowanie rodziny metod attr_* złożonej z metod pobierających/ustawiających, które służą do tworzenia

| Nazwa metody | Efekt | Przykład | Równoznaczny kod |
|---|---|---|---|
| `attr_reader` | Tworzy metodę odczytującą. | `attr_reader :venue` | `def venue`<br>`  @venue`<br>`end` |
| `attr_writer` | Tworzy metodę zapisującą. | `attr_writer :price` | `def price=(price)`<br>`  @price = price`<br>`end` |
| `attr_accessor` | Tworzy metodę odczytującą i zapisującą. | `attr_accessor :price` | `def price=(price)`<br>`  @price = price`<br>`end`<br>`def price`<br>`  @price`<br>`end` |
| `attr` | Tworzy metodę odczytującą i opcjonalnie metodę zapisującą (jeśli drugim argumentem jest `true`). | 1. `attr :venue`<br>2. `attr :price, true` | 1. Sprawdź `attr_reader`<br>2. Sprawdź `attr_accessor` |

We wszystkich przypadkach techniki oparte na rodzinie metod `attr_` powodują automatyczne utworzenie co najmniej jednej metody pobierającej i/lub ustawiającej. Stanowią one zestaw skrótów programistycznych o dużych możliwościach.

Powróćmy do szerszego ujęcia klas, a dokładniej rzecz biorąc — do kwestii *dziedziczenia* klas.

## 3.5. Dziedziczenie i hierarchia klas języka Ruby

*Dziedziczenie* to swego rodzaju przebiegająca w dół relacja między dwoma klasami (superklasa i podklasa), zgodnie z którą jedna klasa „dziedziczy" z innej klasy, a instancje podklasy nabywają zachowania (metody) zdefiniowane w superklasie.

W poniższym przykładzie klasa `Magazine` dziedziczy z klasy `Publication`. Zwróć uwagę na definicję klasy `Magazine`:

```
class Publication
  attr_accessor :publisher
end
```

```
class Magazine < Publication
  attr_accessor :editor
end
```

Symbol < ustanawia klasę `Magazine` podklasą klasy `Publication`. Ponieważ każdy obiekt klasy `Publication` zawiera metody `publisher` i `publisher=` (dzięki kodowi `attr_accessor :publisher`), wszystkie obiekty klasy `Magazine` również mają te metody. Ponadto w obiektach tej klasy znajdują się metody `editor` i `editor=`:

```
mag = Magazine.new
mag.publisher = "Dawid A. Kowalski"
mag.editor = "Jan Nowak"
puts "Czasopismo jest publikowane przez wydawnictwo #{mag.publisher} oraz redagowane przez
↪redaktora #{mag.editor}."
```

Możemy kontynuować przechodzenie w dół kaskady:

```
class Ezine < Magazine
end
```

Instancje klasy `Ezine` zawierają atrybuty `publisher` i `editor`, które zostały zdefiniowane w superklasie i jej klasie nadrzędnej `Ezine`. Zauważ, że nie jest obowiązkowe dodawanie nowych metod do każdej podklasy. Możesz zdecydować się na utworzenie klasy `Ezine` tylko ze względu na możliwość wywołania metody `Ezine.new` zamiast metody `Magazine.new`, aby kod stał się bardziej wyrazisty.

Oczywiście nie są to wszystkie informacje dotyczące metod akcesora atrybutów. Dowolna metoda instancji definiowana w danej klasie może być wywoływana przez jej instancje, a także przez instancje dowolnej podklasy tej klasy:

```
class Person
  def species              ←——❶
    "Homo sapiens"
  end
end
class Rubyist < Person     ←——❷
end
david = Rubyist.new
puts david.species         ←—— Dane wyjściowe: Homo sapiens
```

W tym przykładzie klasa `Rubyist` jest podrzędna względem klasy `Person` ❷. Oznacza to, że dana instancja klasy `Rubyist` (np. `david`) może wywołać metodę `species` zdefiniowaną w klasie `Person` ❶. Jak zawsze w języku Ruby wszystko ma związek z obiektami: co dany obiekt może osiągnąć w wybranym miejscu programu, a czego nie. Obiekty uzyskują własne zachowania ze swoich klas, z poszczególnych metod lub metod pojedynczego obiektu, a także z przodków (superklasa, klasa nadrzędna superklasy itd.) swoich klas (oraz z jednego lub dwóch miejsc, które nie zostały jeszcze omówione). Podsumowując, obiekty języka Ruby cechują się interesującym i dynamicznym „życiem". Częścią tego wszystkiego jest dziedziczenie.

Dziedziczenie ma jednak istotne ograniczenie.

### 3.5.1. Pojedyncze dziedziczenie: po jednym dla klienta

W niektórych językach obiektowych możliwe jest, aby dana klasa dziedziczyła z więcej niż jednej klasy. Na przykład może istnieć klasa `Teacher`, która dziedziczy z klasy `Person`, a także z klasy `Employee` lub klasy `Car` dziedziczącej z klas `Machine`, `Powered` i `Driveable`. Język Ruby nie zezwala na wielokrotne dziedziczenie. Każda jego klasa może mieć tylko jedną superklasę, co jest zgodne z zasadą *pojedynczego dziedziczenia*.

Wbrew początkowemu wrażeniu pojedyncze dziedziczenie obecne w języku Ruby nie ogranicza użytkownika: język zapewnia *moduły*, które są pakietami funkcji programistycznych przypominających klasy (z wyjątkiem tego, że nie mają instancji). Moduły mogą zostać z łatwością dołączone do drzewa rodziny klas w celu udostępnienia obiektom wymaganej liczby metod (w rozdziale 4. zajmiemy się modułami). Nie ma ograniczenia stopnia zaawansowania modelowania obiektów, które może być po prostu przeprowadzane tylko z wykorzystaniem klas i dziedziczenia.

Zasada pojedynczego dziedziczenia oznacza, że nie możesz po prostu utworzyć wielkiego drzewa obiektów, a następnie dokonać jego translacji bezpośrednio do postaci hierarchii klas. Dziedziczenie funkcjonuje często bardziej jako wygodny sposób zastosowania dwóch lub większej liczby klas do współużytkowania definicji metod niż jako końcowe wyrażenie tego, jak rzeczywiste obiekty są ze sobą powiązane w ogólny i szczegółowy sposób. Na przykład każda klasa w języku Ruby ostatecznie wywodzi się (jako podklasa lub klasa podrzędna podklasy itd.) z klasy `Object`, która oczywiście jest bardziej ogólna niż klasa `String` lub `Ticket`. Jednak ograniczenie w postaci pojedynczego dziedziczenia oznacza, że nie możesz liczyć na zaprojektowanie hierarchii klas, które będą przechodzić kaskadowo w dół w stylu zwykłego grafu drzewa.

I tym razem moduły odgrywają kluczową rolę i okażą się przydatne w rozdziale 4. Na razie jednak będziemy kontynuować wątek dziedziczenia w górę, jak to można określić, i przyjrzymy się klasom pojawiającym się u samej góry drzewa dziedziczenia każdego obiektu języka Ruby. Mowa o klasach `Object` i `BasicObject`.

### 3.5.2. Przodkowie obiektów i nie do końca brakujące łącze: klasa Object

Wcześniej zaprezentowano standardową technikę tworzenia ogólnego obiektu:

```
obj = Object.new
```

Możesz teraz dokładniej zrozumieć to, co ma miejsce w tym fragmencie kodu.

Klasa `Object` znajduje się prawie u samej góry wykresu dziedziczenia. Każda klasa jest podklasą klasy `Object`, klasą podrzędną podklasy klasy `Object` lub, w pewnym uproszczeniu, bezpośrednim potomkiem klasy `Object`.

```
class C
end
class D < C
end
puts D.superclass
puts D.superclass.superclass
```

Oto dane wyjściowe:

```
C
Object
```

Takie dane wynikają z tego, że klasa `C` jest superklasą klasy `D` (to nasza zasługa), a klasa `Object` to superklasa klasy `C` (a to zasługa języka Ruby).

Jeśli podążysz w górę łańcucha wystarczająco daleko od dowolnej klasy, natrafisz na klasę `Object`. Dowolna metoda dostępna dla pustej instancji klasy `Object` jest dostępna dla każdego obiektu. Oznacza to, że jeśli możesz użyć następującego kodu:

```
obj = Object.new
obj.pewna_metoda
```

Możesz wywołać metodę `pewna_metoda` dla dowolnego obiektu.

W tym przypadku występuje jednak słowo „prawie". Jak się okazuje, u samej góry hierarchii ma miejsce inne generowanie.

### 3.5.3. Starszy brat El Viejo: BasicObject

Młodszy brat mojego ojca, obecnie 85-letni wspaniały dziadek, jest znany swoim potomkom jako El Viejo, czyli Stary Człowiek. Spowodowało to, że mój kuzyn miał następującą zagadkę do rozwiązania: jak wyjaśnić swojej córeczce, wnuczce El Viejo, kim dokładnie był mój ojciec, gdy spotkała go po raz pierwszy. Ostatecznie kuzyn „chwycił byka za rogi" i przedstawił mojego ojca wnuczce jego brata jako starszego brata El Viejo.

Podobnie do mojego zmarłego w swoim czasie ojca klasa `BasicObject` jest starsza niż stara: w drzewie rodziny klas języka Ruby klasa ta występuje przed klasą `Object`. Celem klasy `BasicObject` jest zapewnianie swego rodzaju obiektu „niezapisanej tablicy", który jest niemal pozbawiony metod (tak naprawdę poprzednikiem klasy `BasicObject` była utworzona przez Jima Weiricha biblioteka o nazwie `BlankSlate`). Klasa `BasicObject` zawiera tak niewiele metod, że wystąpi problem w przypadku utworzenia instancji tej klasy w narzędziu irb:

```
>> BasicObject.new
(Object doesn't support #inspect)
```

Obiekt jest tworzony, ale narzędzie irb nie może wyświetlić jego tradycyjnej reprezentacji łańcuchowej, ponieważ nie zawiera on żadnej metody `inspect`!

W nowo utworzonej instancji klasy `BasicObject` znajduje się tylko 8 metod instancji, natomiast nowa instancja klasy `Object` zawiera 55 metod (liczby te mogą się zmieniać nieznacznie w przypadku różnych wersji języka Ruby, ale są wystarczająco dokładne, aby zwrócić uwagę na to, że klasa `BasicObject` ma niewiele metod). Prawdopodobnie nie będzie konieczne regularne tworzenie instancji lub podklasy klasy `BasicObject` (jeśli w ogóle pojawi się taka potrzeba). Klasa jest przydatna głównie w sytuacjach, w których modelowane są obiekty blisko powiązane z konkretną dziedziną — prawie w stopniu powodującym utworzenie odmiany dialektu języka Ruby — gdy ponadto niepożądane są żadne wyniki fałszywie dodatnie podczas wysyłania komunikatów do tych obiektów. 55 metod może stanowić przeszkodę, gdy masz własne pomysły zwią-

zane z tym, czy obiekty powinny „udawać nieświadome” faktu wysyłania do nich takich komunikatów jak `display`, `extend` lub `clone` (więcej na ten temat będzie mowa w rozdziale 13. przy okazji kontynuowania omawiania klasy `BasicObject`).

Po uwzględnieniu dziedziczenia i przybliżeniu niektórych kluczowych składników rodowodu obiektów języka Ruby powróćmy do zagadnienia klas, a dokładniej rzecz biorąc — do jednego z najbardziej zauważalnych aspektów z nimi związanych, czyli do faktu, że są obiektami, co oznacza, że mogą pełnić rolę odbiorców komunikatów tak jak inne obiekty.

## 3.6. Klasy jako obiekty i odbiorcy komunikatów

Klasy to specjalne obiekty: jest to jedyny rodzaj obiektu, który ma możliwość tworzenia nowych obiektów (instancji). Niemniej jednak klasy są obiektami. Podczas tworzenia klasy, takiej jak `Ticket`, możesz do niej wysyłać komunikaty, dodawać metody, przekazywać ją innym obiektom jako argument metody oraz generalnie realizować dowolne działania możliwe w przypadku innego obiektu.

Podobnie jak inne obiekty, tak i klasy mogą być tak naprawdę tworzone na więcej niż jeden sposób.

### 3.6.1. Tworzenie obiektów klasy

Każda klasa (np. `Object`, `Person`, `Ticket`) to instancja klasy o nazwie `Class`. Jak już pokazano, obiekt klasy możesz utworzyć za pomocą następującego kodu ze słowem kluczowym `class`:

```
class Ticket
  # w tym miejscu znajduje się kod
end
```

Kod ten stanowi specjalny środek w języku Ruby, który pozwala uzyskać ładnie wyglądający i łatwo dostępny blok definicji klasy. Możliwe jest też jednak utworzenie klasy w ten sam sposób, w jaki tworzona jest większość obiektów — przez wysłanie komunikatu `new` do obiektu klasy `Class`:

```
my_class = Class.new
```

W tym przypadku zmiennej `my_class` przypisywany jest nowy obiekt klasy.

Wywołanie `Class.new` odpowiada dokładnie innym wywołaniom konstruktora, takim jak `Object.new` i `Ticket.new`. W momencie tworzenia instancji klasy `Class` tworzona jest klasa. Z kolei ta klasa pozwala utworzyć instancje samej siebie:

```
instance_of_my_class = my_class.new
```

W podrozdziale 3.1 wspomniano, że obiekty klasy są zwykle reprezentowane przez stałe (np. `Ticket` lub `Object`). W poprzednim przykładzie obiekt klasy powiązano ze zwykłą zmienną lokalną (`my_class`). Wywołanie metody `new` powoduje wysłanie komunikatu `new` do klasy za pośrednictwem tej zmiennej.

**Definiowanie metod instancji za pomocą wywołania Class.new**

Aby utworzyć anonimową klasę za pomocą wywołania `Class.new`, a ponadto dodać metody instancji w momencie tworzenia klasy, możesz dołączyć blok kodu po wywołaniu metody `new`. *Blok kodu* to fragment kodu zapewnianego jako część wywołania metody, który może być wykonywany z jej poziomu. Znacznie więcej informacji o blokach kodu zostanie zaprezentowanych podczas omawiania iteratorów w rozdziale 6. Na razie przedstawiamy niewielki przykład wywołania `Class.new` z blokiem:

```
c = Class.new do
  def say_hello
    puts "Witaj!"
  end
end
```

Jeśli utworzysz teraz instancję klasy (za pomocą `c.new`), możliwe będzie wywołanie dla tej instancji metody `say_hello`.

Tak, w tym miejscu pojawia się paradoks...

#### PARADOKS „JAJKO CZY KURA" W PRZYPADKU KLASY I OBIEKTU

Klasa `Class` jest instancją samej siebie, czyli jest obiektem `Class`. To nie wszystko. Czy pamiętasz klasę `Object`? Cóż, `Object` to klasa, ale klasy są obiektami. A zatem `Object` to obiekt, a `Class` to klasa. `Object` jest klasą, a `Class` obiektem.

Co było pierwsze? Jak można utworzyć klasę `Class`, jeśli nie istnieje jeszcze klasa `Object`? Jak jednak może istnieć klasa `Object` (lub dowolna inna klasa), gdy nie ma klasy `Class`, z której mogą być tworzone instancje?

Najlepszym sposobem poradzenia sobie z tym paradoksem, a przynajmniej na razie, jest zignorowanie go. Język Ruby musi zaradzić kwestiom typu „jajko czy kura", aby obsługiwać system obiektów i klas. W związku z tym „okrężność" i paradoksy nie mają znaczenia. Programując, musisz jedynie wiedzieć, że klasy są obiektami, czyli instancjami klasy o nazwie `Class` (jeśli chcesz dowiedzieć się w skrócie, jak to działa, wygląda to następująco: każdy obiekt zawiera wewnętrzny rekord z informacją o tym, jakiej klasy jest instancją, a wewnętrzny rekord w obrębie obiektu `Class` wskazuje na niego samego).

Klasy to obiekty, a obiekty odbierają komunikaty i wykonują metody. Jak dokładnie proces wywoływania metody przebiega w przypadku obiektów klasy?

### *3.6.2. Wywoływanie metod przez obiekty klasy*

Przy wysyłaniu komunikatu do obiektu klasy używany kod ma następującą postać:

```
Ticket.komunikat
```

W obrębie definicji klasy, która pełni rolę obiektu domyślnego `self`, kod ten ma następującą postać:

```
class Ticket
  komunikat    <------ Na przykład: attr_accessor!
```

W taki sposób obiekt klasy otrzymuje komunikaty. Skąd jednak pochodzą metody, którym odpowiadają komunikaty?

Aby zrozumieć, skąd klasy uzyskują swoje metody, zastanów się nad tym, w jakich ogólnie miejscach obiekty otrzymują swoje metody (oprócz modułów, które nie zostały jeszcze objaśnione):

- ze swojej klasy,
- z superklasy i wcześniejszych przodków swojej klasy,
- z własnego magazynu metod pojedynczego obiektu (`talk` w kodzie `def obj.talk`).

W przypadku klas sytuacja wygląda zasadniczo tak samo. Dla obiektów klasy występują specjalne przypadki lub dodatkowe elementy, ale jest ich bardzo niewiele. Takie obiekty zachowują się przeważnie jak inne obiekty.

Przyjrzyjmy się trzem scenariuszom wywoływania przedstawionej powyżej metody w przypadku obiektów klasy.

Instancje klasy `Class` mogą wywoływać metody zdefiniowane jako metody instancji w swojej klasie. Na przykład `Ticket` jest instancją klasy `Class`, która definiuje metodę instancji o nazwie `new`. Z tego powodu używamy następującego kodu:

```
Ticket.new
```

Kod realizuje scenariusz 1. Pora na scenariusz 2.

`Module` to superklasa klasy `Class`. Instancje klasy `Class` mają zatem dostęp do metod instancji zdefiniowanych w klasie `Module`. Wśród tych metod znajduje się rodzina metod `attr_accessor`. Z tego powodu używamy następującego kodu:

```
class Ticket
  attr_reader :venue, :date
  attr_accessor :price
```

Takie wywołania metod trafiają bezpośrednio do obiektu klasy `Ticket`, który pełni rolę obiektu domyślnego `self` w momencie tworzenia wywołań.

Pozostaje jeszcze scenariusz 3: wywołanie metody pojedynczego obiektu klasy.

### 3.6.3. Metoda pojedynczego obiektu

Pora na przykład. Załóżmy, że została utworzona klasa `Ticket`. Na tym etapie nie jest ona jedyną klasą, na bazie której mogą być tworzone obiekty (instancje biletu). Klasa `Ticket` sama również jest obiektem. Podobnie jak w przypadku innych obiektów do obiektu `Ticket` dodajmy metodę pojedynczego obiektu. Metoda pozwoli stwierdzić, z jakim biletem z listy obiektów biletu powiązana jest największa cena. Poniżej zamieszczono kod „czarnej skrzynki". Nie przejmuj się szczegółami. Zasadniczo chodzi o to, że metoda `max_by` znajdzie bilet z najwyższą ceną:

```
def Ticket.most_expensive(*tickets)
  tickets.max_by(&:price)
end
```

Możemy teraz użyć metody `Ticket.most_expensive` do określenia, który bilet spośród kilku biletów jest najdroższy (będziemy unikać występowania dwóch biletów z tą samą ceną, ponieważ stosowane rozwiązanie nie radzi sobie dobrze z taką sytuacją).

```
th = Ticket.new("Sala Miejska","11/12/13")
cc = Ticket.new("Centrum Kongresowe","12/13/14/")
fg = Ticket.new("Plac Targowy", "13/14/15/")
th.price = 12.55
cc.price = 10.00
fg.price = 18.00
highest = Ticket.most_expensive(th,cc,fg)
puts "Bilet z najwyższą ceną dotyczy wydarzenia odbywającego się w miejscu:
#{highest.venue}."
```

Oto dane wyjściowe:

```
Bilet z najwyższą ceną dotyczy wydarzenia odbywającego się w miejscu: Plac Targowy.
```

Metoda `most_expensive` jest definiowana bezpośrednio w obiekcie klasy `Ticket` w stylu metody pojedynczego obiektu. Metoda pojedynczego obiektu definiowana w obiekcie klasy jest często określana mianem *metody klasy*, w której jest definiowana. W przypadku metody klasy komunikat jest wysyłany do obiektu, który jest klasą, a nie do jednej z instancji klasy. Komunikat `most_expensive` trafia do klasy `Ticket`, a nie do konkretnego obiektu biletu.

**Pojęcie metody klasy: więcej problemów niż korzyści?**

W języku Ruby obiekty mogą mieć metody pojedynczego obiektu, a klasy są obiektami. A zatem gdy używasz kodu `def Ticket.most_expensive`, zasadniczo tworzysz metodę pojedynczego obiektu dla obiektu `Ticket`. Gdy po stronie wywołującej wystąpi metoda wywołana w obiekcie klasy (np. `Ticket.new`), samo przyjrzenie się jej nie wystarczy do stwierdzenia, czy ma się do czynienia z metodą pojedynczego obiektu zdefiniowaną bezpośrednio w tej klasie (`def Ticket.new`), czy z metodą instancji klasy `Class`.

Aby wszystko stało się jeszcze bardziej zabawne, klasa `Class` zawiera metodę `new` zarówno w wersji metody klasy, jak i w wersji metody instancji. Pierwsza wersja metody jest wywoływana podczas używania kodu `Class.new`, a druga wersja w przypadku zastosowania kodu `Ticket.new`. Będzie tak oczywiście, jeśli nie nadpiszesz metody `new` przez zdefiniowanie jej samemu dla obiektu `Ticket`...

Trzeba przyznać, że metoda `new` to wyjątkowo trudny przypadek. Ogólnie jednak pojęcie *metody klasy* niekoniecznie dobrze pasuje do języka Ruby. Pojęcie to występuje też w innych językach obiektowych. Jednak w ich przypadku metody klasy i metody instancji różnią się w większym stopniu. W języku Ruby w momencie wysyłania komunikatu do obiektu klasy nie możesz stwierdzić, gdzie, a także w jaki sposób odpowiednia metoda została zdefiniowana.

A zatem *metoda klasy* ma zarazem niewyraźne i konkretne znaczenie. W pierwszym wariancie dowolna metoda wywoływana bezpośrednio w obiekcie `Class` jest metodą klasy. W drugim wariancie metoda klasy jest definiowana, a nie tylko wywoływana, bezpośrednio w obiekcie `Class`. Będziesz się spotykać z obydwoma wariantami użycia metody klasy. Dopóki będziesz świadom bazowej inżynierii, a ponadto w razie potrzeby będziesz w stanie dokonywać wyraźnych rozróżnień, poradzisz sobie.

Dlaczego miałbyś zastosować metodę klasy? Czy nie spowoduje to zaburzenia fundamentalnego porządku, czyli tworzenia obiektów biletu i wysyłania do nich komunikatów?

### 3.6.4. Kiedy i dlaczego należy tworzyć metodę klasy?

Metody klasy mają swoje zastosowanie. Niektóre operacje dotyczące klasy nie mogą być wykonywane przez jej poszczególne instancje. Metoda `new` to znakomity tego przykład. Metoda `Ticket.new` jest wywoływana, ponieważ do momentu utworzenia pojedynczego obiektu biletu nie można wysłać do niego żadnych komunikatów! Poza tym zadanie polegające na tworzeniu nowego obiektu pod względem logicznym należy do klasy. Nie ma sensu, aby instancje klasy `Ticket` tworzyły się wzajemnie. Sensowne jest jednak scentralizowanie procesu tworzenia instancji jako działania klasy `Ticket`.

Innym podobnym przypadkiem jest wbudowana metoda języka Ruby o nazwie `File.open`, która — jak pokazano w rozdziale 1. — otwiera plik do odczytu i/lub zapisu. Operacja realizowana przez metodę `open` przypomina trochę działanie metody `new`. Metoda `open` inicjuje wejście i/lub wyjście plikowe oraz zwraca obiekt `File`. Sensowne jest, aby metoda `open` była metodą klasy `File`: utworzenie pojedynczego obiektu jest żądane od klasy. Klasa pełni rolę punktu wyjścia dla tworzonych przez nią obiektów.

Metoda `Ticket.most_expensive` to inny przypadek pod tym względem, że nie tworzy nowego obiektu, ale nadal jest metodą, która logicznie należy do klasy. Znajdowanie najdroższego biletu na liście biletów może być postrzegane jako nadrzędna operacja, czyli coś, co jest realizowane zbiorczo w odniesieniu do biletów, a nie przez poszczególne obiekty biletu. Tworzenie metody `most_expensive` jako metody klasy `Ticket` umożliwia zachowanie metody w — jak to można określić — rodzinie obiektów biletu, przypisując metodę do abstrakcyjnego, nadzorczego poziomu reprezentowanego przez klasę.

Zdarza się, że klasa jest tworzona tylko w celu zapewnienia jej metod klasy. We wcześniej zamieszczonych ćwiczeniach dotyczących konwersji stopni temperatury możliwe jest skorzystanie z takiego rozwiązania.

#### KONWERTOWANIE KONWERTERA

Skonwertujmy konwerter w klasę konwertera, dodając metody klasy obsługujące konwersję w obu kierunkach:

```
class Temperature
  def Temperature.c2f(celsius)
    celsius * 9.0 / 5 + 32
  end
  def Temperature.f2c(fahrenheit)
    (fahrenheit - 32) * 5 / 9.0
  end
end
```

Wypróbujmy powyższy kod:

```
puts Temperature.c2f(100)    ←——— Zwraca wartość 212.0
```

Chodzi o to, że dysponujemy metodami narzędziowymi powiązanymi z temperaturą, które odnoszą się do niej jako pojęcia, a nie do konkretnej wartości temperatury. Klasa `Temperature` to dobra propozycja obiektu, do którego mogą należeć te metody. Można pójść jeszcze dalej i zastosować instancje klasy `Temperature`, które będą w stanie stwierdzić, czy są powiązane ze stopniami Celsjusza, czy Fahrenheita, a ponadto samodzielnie

dokonać konwersji. Jednak praktycznie rzecz ujmując, użycie klasy `Temperature` z metodami klasy do przeprowadzania konwersji to właściwe i możliwe do zaakceptowania rozwiązanie (jeszcze lepsze, ponieważ nie ma w ogóle potrzeby stosowania instancji klasy `Temperature`, byłoby użycie modułu, czyli swego rodzaju klasy pozbawionej instancji, która szczegółowo zostanie omówiona w rozdziale 4.).

Metody klasy i metody instancji nie różnią się między sobą radykalnie. Wszystkie są metodami, a ich wykonywanie zawsze jest spowodowane wysłaniem komunikatu do obiektu, z tym że obiekt otrzymujący komunikat może być obiektem klasy. Niemniej jednak w dalszym ciągu występują różnice i ważne kwestie, o których trzeba pamiętać przy rozpoczynaniu tworzenia metod na różnych poziomach.

### 3.6.5. Porównanie metod klasy z metodami instancji

Definiując metodę `Ticket.most_expensive`, zdefiniowaliśmy metodę dostępną za pośrednictwem obiektu klasy `Ticket`, lecz nie jego instancji. Poszczególne obiekty biletu (instancje klasy `Ticket`) są pozbawione tej metody. Z łatwością można to sprawdzić. Spróbuj dodać następujący kod do kodu z punktu 3.6.3, gdzie zmienna `fg` odwoływała się do obiektu `Ticket` (dla wydarzenia na Placu Targowym):

```
puts "Testowanie odpowiedzi instancji klasy Ticket...."
wrong = fg.most_expensive
```

Pojawia się komunikat o błędzie, ponieważ obiekt `fg` nie zawiera metody o nazwie `most_expensive`. Klasa tego obiektu, czyli `Ticket`, oferuje taką metodę. Z kolei obiekt `fg`, który jest instancją klasy `Ticket`, już nie.

Pamiętaj o następujących rzeczach:

- Klasy są obiektami.
- Instancje klas również są obiektami.
- Obiekt klasy (np. `Ticket`) ma własne metody, stan oraz tożsamość. *Obiekt ten nie współużytkuje tych elementów z własnymi instancjami*. Wysłanie komunikatu do obiektu `Ticket` nie jest tym samym co wysłanie komunikatu do obiektu `fg` lub `cc` bądź do dowolnej innej instancji obiektu `Ticket`.

Jeśli kiedykolwiek nie będziesz pewien, co jest metodą klasy, a co metodą instancji, zwykle możesz rozwiać wątpliwości przez powrócenie do tych trzech zasad.

Omówienie klas zawsze wiąże się z użyciem mnóstwa stałych. Tak też będzie podczas prezentowania modułów w rozdziale 4. A zatem przyjrzymy się jeszcze dokładniej niż dotychczas przeznaczeniu i sposobowi działania stałych.

## 3.7. Szczegóły dotyczące stałych

Wiele klas składa się w zasadzie z metod instancji i/lub metod klasy. W przypadku wielu klas stałe są jednak ważnym i wspólnym trzecim składnikiem. Stałe zostały już zastosowane jako nazwy klas. Stałe mogą również być używane do ustawiania i zachowywania w klasach ważnych wartości danych.

**Uwaga dotycząca zapisu metod**

Przy omawianiu metod języka Ruby i odwoływania się do nich (oczywiście poza obrębem kodu) zwykle metody instancji są przywoływane przez podawanie nazwy klasy (lub modułu, jak też się zdarza), w której są one zdefiniowane. Po tej nazwie następuje znak # i nazwa metody. Z kolei w odniesieniu do metod klasy stosowana jest podobna konstrukcja, w której jednak zamiast znaku # występuje kropka. Czasami w miejsce kropki pojawi się podwójny dwukropek (`::`), co dotyczy metody klasy.

Oto przykłady takich zapisów wraz z informacją o tym, do czego się one odwołują:

- Zapis `Ticket#price` odnosi się do metody instancji `price` w klasie `Ticket`.
- Zapis `Ticket.most_expensive` odnosi się do metody klasy `most_expensive` w klasie `Ticket`.
- Zapis `Ticket::most_expensive` również odnosi się do metody klasy `most_expensive` w klasie `Ticket`.

Gdy teraz spotkasz się z takimi zapisami (w tej książce lub gdzie indziej), będziesz wiedzieć, co one oznaczają. Drugi przykładowy zapis, który zawiera odwołanie do metody klasy z kropką, wygląda tak samo jak wywołanie metody, ale na podstawie kontekstu stwierdzisz, czy jest to wywołanie metody, czy odwołanie do metody klasy.

Dalej zajmiemy się zasięgiem stałych i technikami zagnieżdżania ich wewnątrz wielopoziomowych klas i modułów. Na razie skoncentrujemy się na podstawach dotyczących sposobu ich wykorzystania, a także zastanowimy się, w jakim stopniu stałe faktycznie nimi są.

### 3.7.1. Podstawowe zastosowanie stałych

Nazwa każdej stałej zaczyna się od dużej litery. Przypisywanie do stałych odbywa się bardzo podobnie jako do zmiennych.

Załóżmy, że zdecydowaliśmy się na utworzenie listy predefiniowanych miejsc dla klasy `Ticket`. Do listy tej może odwołać się każdy obiekt biletu i dokonywać z niej wyboru. Lista przypisywana jest do stałej. Definicje stałych trafiają na początek definicji klasy lub w jego pobliże:

```
class Ticket
  VENUES = ["Centrum Kongresowe", "Plac Targowy", "Sala Miejska"]
```

Odwołanie do stałej zdefiniowanej w klasie może mieć miejsce z wnętrza metod instancji lub metod klasy. Przyjmijmy, że chcesz zapewnić, że każdy bilet jest przeznaczony na wydarzenie organizowane w ustalonym wcześniej miejscu. Metodę `initialize` możesz przebudować w następujący sposób:

```
def initialize(venue, date)
  if VENUES.include?(venue)      ←—— Czy jest to jedno ze znanych miejsc?
    @venue = venue
  else
    raise ArgumentError, "Nieznane miejsce #{venue}"  ←—— Zgłoszenie wyjątku (błąd
  end                                                     krytyczny — więcej
  @date = date                                            informacji w rozdziale 6.)
end
```

Możliwe jest też odwołanie do stałej całkowicie poza obrębem definicji klasy za pomocą specjalnego zapisu wyszukiwania stałej, który ma postać podwójnego dwukropka (`::`).

Oto przykład ustawiania wewnątrz klasy stałej, a następnie odwołania się do niej spoza klasy:

```
class Ticket
  VENUES = ["Centrum Kongresowe", "Plac Targowy", "Sala Miejska"]
end
puts "Zamknięto definicję klasy."
puts "Z tego powodu konieczne jest użycie zapisu ścieżki w celu uzyskania dostępu do stałej."
puts "Miejsca:"
puts Ticket::VENUES
```

**Przechowuje miejsca jako tablicę łańcuchów**

Zapis w postaci podwójnego dwukropka dokładnie określa w obrębie klasy stałą `VENUES` znaną pod nazwą stałej `Ticket`. W efekcie wyświetlana jest lista miejsc wydarzeń.

Język Ruby oferuje kilka predefiniowanych stałych, do których w ten sposób można uzyskać dostęp. Stałe te mogą okazać się przydatne.

### PREDEFINIOWANE STAŁE JĘZYKA RUBY

W narzędziu irb spróbuj wpisać następujący wiersz:

```
Math::PI
```

Choć `Math` to moduł (poświęcono mu rozdział 4.), obowiązuje taka sama zasada jak w przypadku stałej zdefiniowanej w obrębie klasy: konektor `::` służy do wyszukiwania stałej `PI` zdefiniowanej w module `Math`. W identyczny sposób możesz znaleźć stałą `E`.

Wiele predefiniowanych stałych możesz sprawdzić podczas uruchamiania interpretera języka Ruby (lub narzędzia irb). Są to nazwy klas wbudowanych: `String`, `Array`, `Symbol` itd. Część stałych ma cel informacyjny. Nawet bez ładowania pakietu `rbconfig` (omówiono go w rozdziale 1.) możesz uzyskać od interpretera sporą ilość informacji o jego ustawieniach. Oto kilka przykładów:

```
>> RUBY_VERSION
=> "2.1.0"
>> RUBY_PATCHLEVEL
=> 0
>> RUBY_RELEASE_DATE
=> "2013-12-25"
>> RUBY_REVISION
=> 44422
>> RUBY_COPYRIGHT
=> "ruby - Copyright (C) 1993-2013 Yukihiro Matsumoto"
```

Jak widać, informacje przechowywane w tych stałych odpowiadają informacjom uzyskiwanym za pomocą opcji `-v`:

```
$ ruby -v
ruby 2.1.0p0 (2013-12-25 revision 44422) [x86_64-darwin12.0]
```

Szczególną cechą stałych języka Ruby jest to, że nie są niezmienne. Można je zmieniać, przy czym w tym przypadku słowo *zmiana* ma dwa znaczenia. Z tego też powodu pora na lekcję pouczającą.

### 3.7.2. Porównanie ponownego przypisania z modyfikowaniem stałych

Istnieje możliwość wykonania przypisania dla stałej, do której zostało już coś przypisane, czyli przeprowadzenia operacji ponownego przypisania do stałej. Jeśli jednak takie działanie ma miejsce, pojawi się ostrzeżenie (nawet wtedy, gdy nie jest używana opcja wiersza poleceń `-w`). W narzędziu irb spróbuj zastosować następujące przypisania:

```
A = 1
A = 2
```

Zostanie wyświetlony następujący komunikat:

```
(irb):2: warning: already initialized constant A
(irb):1: warning: previous definition of A was here
```

To, że nazwy stałych mogą być wielokrotnie używane, co jednocześnie może być wykroczeniem podlegającym ostrzeżeniu, stanowi kompromis. Z jednej strony przydatnym rozwiązaniem jest istnienie w języku osobnej kategorii dla stałych jako sposobu przechowywania danych, które pozostają widoczne przez dłuższy czas działania programu niż w przypadku zwykłej zmiennej. Z drugiej strony Ruby to dynamiczny język w tym sensie, że podczas działania kodu wszystko może się zmieniać. Takie zdefiniowanie stałych, aby były tutaj wyjątkiem, teoretycznie byłoby możliwe, ale spowodowałoby wystąpienie w języku osobliwości.

Ponadto ze względu na to, że ponownie ładujesz już załadowane pliki programów, które mogą zawierać przypisania do stałych, zabronienie ponownego przypisania do stałych oznaczałoby, że wiele operacji ponownego ładowania plików zakończyłoby się niepowodzeniem z wygenerowanym błędem krytycznym.

A zatem możliwe jest ponowne przypisania do stałej, z tym że takie działanie nie jest uznawane za dobrą praktykę. Jeśli wymagany jest identyfikator do wielokrotnego wykorzystania, należy zastosować zmienną.

Inne znaczenie możliwej *zmiany* stałej to wprowadzanie zmian w obiekcie, do którego stała się odwołuje. Na przykład dodanie miejsca do listy miejsc klasy `Ticket` jest prostą operacją:

```
venues = Ticket::VENUES
venues << "Sala gimnastyczna liceum"
```

**Stosuje operator << w celu dodania nowego elementu do istniejącej tablicy**

Nie pojawia się ostrzeżenie, ponieważ nie występuje ponowna definicja stałej. Zamiast tego modyfikowana jest tablica, która nie ma żadnych konkretnych informacji o tym, czy została przypisana do stałej. Tablica realizuje tylko to, czego od niej zażądano.

Istotna jest różnica między ponownym przypisaniem nazwy stałej i modyfikowaniem obiektu, do którego odwołuje się stała. Różnica ta stanowi przydatną lekcję związaną z dwoma rodzajami zmiany w języku Ruby: zmiany odwzorowania identyfikatorów na obiekty (przypisanie) i zmiany stanu lub stałych obiektu. W przypadku nazw zwykłych zmiennych nie pojawia się ostrzeżenie przy wykonywaniu ponownego przypisania. Jednak ponowne przypisanie nadal różni się od wprowadzania zmian w obiekcie, co dotyczy dowolnej kategorii identyfikatora.

Jeśli powiążesz zagadnienia przedstawione w tym rozdziale z niektórymi wcześniej zaprezentowanymi przykładami, zaczniesz uzyskiwać ogólny i właściwy obraz tego, jak są konstruowane obiekty języka Ruby: swoją funkcjonalność nabywają z metod instancji zdefiniowanych w ich klasach oraz przodkach tych klas, ale jednocześnie mają też możliwość „uczenia się" specyficznych i zindywidualizowanych zachowań w postaci metod pojedynczego obiektu. Właśnie to sprawia, że język Ruby jest tak fascynujący. „Życie" obiektu języka Ruby jest, przynajmniej potencjalnie, mieszanką okoliczności jego „narodzin" i cech nabywanych w czasie swojego istnienia. Niniejszy rozdział podsumujemy dalszym eksplorowaniem tych ważnych kwestii.

## 3.8. „Natura" i „wychowanie" w przypadku obiektów języka Ruby

Relacja między klasami i ich instancjami to w zasadzie relacja między ogółem i szczegółem, czyli wzorzec znany w skali całego świata. Postrzegamy królestwo zwierząt, posługując się ogólnymi i specyficznymi terminami. Podobnie jest w odniesieniu do wszystkiego innego, począwszy od instrumentów muzycznych, poprzez wydziały uczelni i systemy magazynowania bibliotek, a skończywszy na panteonach bogów.

W zakresie, w jakim język programowania pomaga modelować rzeczywisty świat (lub odwrotnie rzecz ujmując, rzeczywisty świat zapewnia metody organizowania programów), możliwy jest gorszy wariant niż silne bazowanie na relacji między ogółem i szczegółem. Jak można zauważyć, dziedziczenie (relacja między superklasą i podklasą) w dużym stopniu powiela proporcję ogółu do szczegółu. Co więcej, jeśli masz styczność z kręgami powiązanymi z programowaniem obiektowym, poznasz skróconą formę określania takiej relacji. Takim skrótem jest słowo *to*. Jeśli na przykład obiekt `Ezine` dziedziczy z obiektu `Magazine`, mówimy, że „czasopismo online to magazyn". Podobnie obiekt `Magazine` to obiekt `Publication`, gdy pierwszy obiekt dziedziczy z drugiego.

Język Ruby umożliwia modelowanie w ten sposób. Postrzeganie używanej domeny jako kaskadowej mapy obiektów bazującej na dziedziczeniu pozwala uzyskać wiele korzyści. Język ten udostępnia nawet metodę `is_a?`, która informuje o tym, czy obiekt ma daną klasę jako własną klasę, czy jako jedną z klas nadrzędnych swojej klasy:

```
>> mag = Magazine.new
=> #<Magazine:0x36289c>
>> mag.is_a?(Magazine)
=> true
>> mag.is_a?(Publication)
=> true
```

Organizowanie klas w postaci drzew rodzin powiązanych elementów, gdzie każda generacja jest trochę bardziej szczegółowa niż poprzednia, może zapewnić miłe odczucie uporządkowania i determinizmu w „krajobrazie" programu.

Obiekty języka Ruby (w przeciwieństwie do obiektów w kilku innych językach obiektowych) mogą jednak być indywidualnie modyfikowane. Instancja danej klasy nie jest ograniczona wyłącznie do zachowań i właściwości, które instancji nadała jej klasa.

Jak pokazano w wielu przykładach, zawsze możesz dodawać metody dla poszczególnych obiektów. Ponadto klasy mogą się zmieniać. Możliwe jest nabywanie przez obiekt możliwości (metod) w okresie swojego istnienia, jeśli jego klasa lub klasa nadrzędna uzyska nowe metody instancji.

W językach, w których nie możesz dodawać metod do poszczególnych obiektów lub klas już utworzonych, klasa obiektu (oraz superklasa tej klasy itd.) informuje o wszystkim, czego musisz się dowiedzieć o obiekcie. Jeśli obiekt jest instancją klasy `Magazine`, a ponadto jesteś zaznajomiony z metodami udostępnianymi przez klasę `Magazine` na potrzeby użycia jej instancji, wiesz dokładnie, jak obiekt się zachowuje.

W języku Ruby zachowanie lub możliwości obiektu mogą jednak różnić się od zapewnianych przez jego klasę. Możemy sprawić, że magazyn „uniesie się na skrzydłach":

```
mag = Magazine.new
def mag.wings
  puts "Spójrz! Potrafię latać!"
end
mag.wings  <------ Dane wyjściowe: Spójrz! Potrafię latać!
```

Pokazuje to, że możliwości, z jakimi „narodził się" obiekt, niekoniecznie stanowią całość obrazu.

Oznacza to, że drzewo dziedziczenia (skierowana do góry kaskada klasy przechodzącej do superklasy i jej klasy nadrzędnej) nie jest jedynym wyznacznikiem zachowania obiektu. Aby dowiedzieć się, do czego służy obiekt zupełnie nowego magazynu, sprawdź metody w klasie `Magazine` i jej przodkach. Jeśli chcesz wiedzieć, co obiekt magazynu może zrealizować później, musisz stwierdzić, co stało się z obiektem od momentu jego utworzenia (przydatna może okazać się metoda `respond_to?`, która umożliwia określenie z góry, czy obiekt potrafi obsługiwać konkretną metodę).

Obiekty języka Ruby cechują się wyjątkową elastycznością i dynamicznością. Elastyczność ta przekłada się na możliwości programisty: możesz sprawić, że magazyny będą „latać", krowy powiedzą, kto je opublikował, a ponadto wykonać całą resztę rzeczy. Jak wyraźnie pokazują te zabawne przykłady, możliwości oznaczają odpowiedzialność. Gdy wprowadzasz zmiany w pojedynczym obiekcie (dodajesz metody do takiego obiektu, a także jego samego), musisz to robić z uzasadnionego powodu.

Większość programistów używających języka Ruby wykazuje się konserwatywnością w tym względzie. Wbrew oczekiwaniom zauważysz, że metody będą w mniejszym stopniu dodawane do poszczególnych obiektów. Najczęstszym przypadkiem dodawania metod bezpośrednio do obiektów jest dodawanie metod klasy do obiektów klasy. Zdecydowana większość definicji metod pojedynczego obiektu, z jakimi się spotkasz (`def obiekt.metoda`), będzie definicjami metod klasy. Dodawanie metod do innych obiektów (magazyny, bilety, krowy itp.) również jest możliwe, ale musisz to realizować ostrożnie i w sposób selektywny, a także uwzględniając projekt programu.

W większości przypadków indywidualizacja obiektów (nawiasem mówiąc, zagadnieniu temu poświęcono w całości rozdział 13.) ma związek z dynamicznie określanymi warunkami w czasie działania kodu. Na przykład możesz dodawać do obiektów metody akcesora w celu dopasowania nazw kolumn bazy danych, których nie znasz do

momentu uruchomienia programu i skierowania zapytania do bazy danych. Możesz też używać biblioteki specjalnych metod utworzonej dla obiektów łańcuchowych, do których to metod mają mieć dostęp tylko określone łańcuchy. Język Ruby sam zajmuje się realizacją takich zadań, ponieważ klasa obiektu stanowi jedynie część całości, czyli „naturę", jak to można określić, a nie „wychowanie".

Istnieje jeszcze jeden element układanki, czyli moduły. Jest to konstrukcja języka Ruby, o której wspomniano już wcześniej kilkakrotnie. Moduły zostaną dokładniej omówione w następnym rozdziale.

## 3.9. Podsumowanie

W rozdziale zaprezentowano następujące podstawowe informacje dotyczące klas języka Ruby:

- Sposób, w jaki utworzenie klasy, a następnie jej instancji, pozwala współużytkować zachowania między wieloma obiektami.
- Sposób użycia metod ustawiających i pobierających, które napisano ręcznie lub utworzono automatycznie za pomocą rodziny metod `attr_*`, do tworzenia atrybutów obiektów przechowujących stan obiektu w zmiennych instancji.
- Do klas będących obiektami mogą być dodawane metody dla poszczególnych obiektów. Takie metody są powszechnie znane jako metody klasy. Zapewniają one ogólną funkcjonalność narzędziową połączoną z klasą.
- Stałe języka Ruby to specjalnego rodzaju identyfikator, który zwykle znajduje się w obrębie definicji klas (lub modułów).
- Dziedziczenie to relacja między superklasą i co najmniej jedną podklasą. Wszystkie obiekty języka Ruby mają wspólnego przodka w klasach `Object` i `BasicObject`.
- Struktura złożona z superklasy i podklas może nadawać się do modelowania elementów w ściśle hierarchiczny i taksonomiczny sposób, ale dynamiczne właściwości obiektów języka Ruby (w tym obiektów klasy!) mogą zaoferować mniej ściśle określane sposoby postrzegania obiektów, a także tego, jak ich zachowania mogą rozwijać się w czasie istnienia obiektów.

Zamieszczone w rozdziale omówienie klas zapewnia solidne podstawy pozwalające zrozumieć, jak powstają obiekty, a także jak są ze sobą powiązane w języku Ruby. W dalszej kolejności poszerzymy tę podstawową wiedzę, prezentując moduły, czyli kolejny istotny blok tworzący system obiektowy.

# 4 Moduły i organizacja programu

**W tym rozdziale:**

- Hermetyzacja zachowania w modułach.
- Rozszerzenie modularne klas.
- Ścieżka wyszukiwania metod obiektów.
- Obsługa błędu wyszukiwania metod.
- Definiowanie przestrzeni nazw za pomocą modułów oraz zagnieżdżanie.

W rozdziale zostanie zaprezentowana konstrukcja języka Ruby blisko powiązana z klasami. Mowa o modułach. Jak wskazuje ich nazwa, moduły sprzyjają projektowi modularnemu, w przypadku którego duże komponenty programu są dzielone na mniejsze, a ponadto możliwe jest łączenie zachowań obiektów.

Podobnie do klas moduły stanowią pakiety metod i stałych. W przeciwieństwie do klas moduły nie mają instancji. Zamiast tego określasz, że chcesz dodać funkcjonalność konkretnego modułu do funkcjonalności klasy lub określonego obiektu.

Nie jest przypadkiem to, że pod wieloma względami moduły przypominają klasy: klasa `Class` jest podklasą klasy `Module`, dlatego każdy obiekt klasy jest też obiektem modułu. Klasy zostały omówione jako pierwsze, gdyż język Ruby to język obiektowy, a obiekty są instancjami klas. Możesz jednak stwierdzić, że moduły są bardziej podstawową strukturą, a klasy są jedynie specjalizacją. Końcowy wniosek jest taki, że oba elementy są częścią języka Ruby, a ponadto są dostępne dla programisty podczas projektowania programów i modelowania danych.

Analizowanie modułów stanowi kontynuację podążania ścieżkami, które po części zostały obrane w poprzednim rozdziale:

- Pokazano, że wszystkie obiekty wywodzą się z klasy `Object`. W tym rozdziale dowiesz się o module `Kernel`, który zawiera większość metod wspólnych dla wszystkich obiektów.
- Dowiedziałeś się, że obiekty szukają swoich metod zarówno w klasie, jak i w superklasie, pokonując całą drogę prowadzącą w górę drzewa dziedziczenia. W rozdziale dość dokładnie przyjrzymy się temu, jak przebiega taki proces wyszukiwania metod, gdy zaangażowane są klasy i moduły.

## 4.1. Podstawowe informacje dotyczące tworzenia i używania modułów

Tworzenie modułu przypomina tworzenie klasy, z tą różnicą, że zamiast od słowa kluczowego `class` definicja modułu rozpoczyna się od słowa kluczowego `module`:

```
module MyFirstModule
  def say_hello
    puts "Witaj"
  end
end
```

Po utworzeniu klasy tworzone są jej instancje. Mogą one wykonywać metody instancji klasy. Z kolei moduły są pozbawione instancji. Moduły są *dodawane* do klas za pomocą metody `include` lub `prepend`.

> **UWAGA** Metoda `prepend` jest nowością w języku Ruby 2.0, natomiast metoda `include` jest jego częścią od samego początku.

Moduł dodawany w ten sposób czasami jest określany mianem „dodatku". Wynikiem dodania modułu jest to, że instancje klasy uzyskują dostęp do metod instancji zdefiniowanych w module.

Używając na przykład niewielkiego modułu z poprzedniego przykładu, możesz uzyskać następujący kod:

```
class ModuleTester
  include MyFirstModule
end
mt = ModuleTester.new
mt.say_hello
```

Obiekt `ModuleTester` wywołuje odpowiednią metodę (`say_hello`) i zwraca łańcuch `Witaj`. Zauważ, że metoda ta nie jest zdefiniowana w klasie, której obiekt jest instancją. Metodę zdefiniowano w module dodawanym przez klasę.

Operacja dodawania w tym przykładzie jest realizowana za pomocą wywołania metody `include`. Dodawanie metody w dużym stopniu przypomina dziedziczenie z superklasy. Jeśli na przykład klasa `B` dziedziczy z klasy `A`, instancje klasy `B` mogą wywoływać metody instancji klasy `A`. Jeśli natomiast klasa `C` dodaje moduł `M`, instancje klasy `C` mogą wywołać metody instancji modułu `M`.

W obu przypadkach instancje klasy na dole listy zyskują: mogą wywoływać nie tylko metody instancji własnej klasy, ale też takie metody superklasy (w pierwszym przypadku) lub dodanego modułu (w drugim przypadku).

Główna różnica między dziedziczeniem z klasy i dodawaniem modułu jest taka, że możliwe jest dodanie więcej niż jednego modułu. Żadna klasa nie może dziedziczyć z więcej niż jednej klasy. W sytuacjach, w których na potrzeby instancji klasy wymaganych jest wiele dodatkowych zachowań, a ponadto nie chcesz umieszczać ich wszystkich w superklasie i jej klasach nadrzędnych, możesz użyć modułów w celu zorganizowania kodu w bardziej szczegółowy sposób. Każdy moduł może dodać coś innego do metod dostępnych za pośrednictwem klasy (w punkcie 4.4.1 obszerniej zostanie porównane dodawanie z dziedziczeniem).

Moduły stwarzają mnóstwo możliwości, a zwłaszcza w przypadku współużytkowania kodu przez więcej niż jedną klasę, ponieważ dowolna liczba klas może dodawać ten sam moduł. Przyjrzymy się kilku dodatkowym przykładom, które pozwolą zorientować się w tych możliwościach.

### 4.1.1. Moduł hermetyzujący „podobieństwo do stosu"

Moduły zapewniają możliwość gromadzenia i hermetyzowania zachowań. Typowy moduł zawiera metody połączone z konkretnym podzbiorem tego, co ostatecznie będzie pełnymi możliwościami obiektu.

Aby uzasadnić powyższe stwierdzenie, stworzymy moduł hermetyzujący właściwość pozwalającą na upodabnianie do stosu. W dalszej kolejności użyjemy tego modułu w celu nadania obiektom zachowań przypominających stos za pośrednictwem procesu dodawania modułu podobnego do stosu do jednej lub większej liczby klas.

Jak możesz wiedzieć z wcześniejszych omówień, *stos* to struktura danych funkcjonująca zgodnie z zasadą LIFO (*Last In, First Out*), czyli ostatni element wejściowy jest pierwszym elementem wyjściowym. Klasycznym przykładem jest fizyczny stos talerzy. Pierwszy talerz, który zostanie użyty, jest ostatnim umieszczonym na stosie. Stosy są zwykle omawiane w połączeniu z kolejkami, które działają zgodnie z zasadą FIFO (*First In, First Out*), czyli pierwszy element wejściowy jest pierwszym elementem wyjściowym. Pomyśl o stołówce: talerze są na stosie, a klienci stoją w kolejce.

W sposób przypominający stos zgodnie z zasadą LIFO działa wiele elementów. Ostatni arkusz papieru do drukarki umieszczony w podajniku jest pierwszym drukowanym. Samochody zaparkowane w sąsiadujących rzędach muszą wyjeżdżać w kolejności odwrotnej do kolejności parkowania. Właściwość upodabniająca do stosu może objawić się sama w przypadku wielu różnych kolekcji i agregacji elementów.

W odniesieniu do tego pojawiają się moduły. Gdy projektujesz program i zidentyfikujesz zachowanie lub zestaw zachowań, które mogą być przejawiane przez więcej niż jeden rodzaj elementu lub obiektu, taki przypadek będzie dobrym kandydatem do zastosowania modułu. Podobieństwo do stosu pasuje idealnie do następującego wariantu: więcej niż jeden element, stąd też prawdopodobnie więcej niż jedna klasa przejawiają zachowanie podobne do stosu. Tworząc moduł definiujący metody, które zawierają wszystkie obiekty przypominające stos, zapewniasz sobie sposób wymuszania podobieństwa do stosu w przypadku wszystkich bez wyjątku klas, które tego wymagają.

Listing 4.1 prezentuje prostą implementację podobieństwa do stosu w postaci modułu języka Ruby. Zapisz kod z tego listingu w pliku o nazwie *stacklike.rb*. Plik zostanie załadowany w zamieszczonych dalej przykładach.

**Listing 4.1. Moduł Stacklike hermetyzujący strukturę i zachowanie przypominające stos**

```
module Stacklike
  def stack
    @stack ||= []          ←❶
  end
  def add_to_stack(obj)    ←❷
    stack.push(obj)
  end
  def take_from_stack      ←❸
    stack.pop
  end
end
```

Moduł `Stacklike` w powyższym listingu używa *tablicy* (uporządkowana kolekcja obiektów) do reprezentowania stosu. Tablica jest utrzymywana w zmiennej instancji `@stack` i udostępniana za pośrednictwem metody `stack` ❶. Metoda ta korzysta z typowej techniki służącej do warunkowego ustawiania zmiennej, czyli z operatora `||=` (lub-równa się). Efektem zastosowania tego operatora jest ustawienie zmiennej na określoną wartość, którą w tym przypadku jest nowa, pusta tablica, wtedy i tylko wtedy, gdy zmienna nie jest już ustawiona na coś innego niż wartość `nil` lub `false`. Z praktycznego punktu widzenia oznacza to, że przy pierwszym wywołaniu metody `stack` ustawi ona zmienną `@stack` na pustą tablicę. Z kolei w przypadku kolejnych wywołań zmienna `@stack` będzie mieć już wartość i po prostu ją zwróci (tablica).

**Operatory skrótów w języku Ruby**

Oprócz operatora `||=` język Ruby oferuje rodzinę operatorów skrótów z wyglądu podobnych do operatora `||=`, lecz skonstruowanych trochę inaczej. Operatory te są rozwijane do postaci wywołań bazowej metody. Typowym operatorem jest operator `+=`. Wyrażenie `+= 1` odpowiada `a = a + 1`. Wśród innych członków tej rodziny są operatory `-=`, `*=`, `/=`, `**=` (podnoszenie do potęgi), `&=` (operacja bitowa `AND`), `|=` (operacja bitowa `OR`), `^=` (operacja bitowa `EXCLUSIVE OR`), `%=` (reszta z dzielenia) oraz rzadko używany operator (`&&=`), który działa podobnie do operatora `||=`. A zatem `a -= 1` oznacza `a = a - 1`, `*= 10` odpowiada `a = a * 10` itd.

Każdy z tych operatorów opakowujących metody działa z dowolnym obiektem, który zawiera odpowiednią metodę bazową, w tym z instancjami utworzonych przez siebie klas. Jeśli na przykład zdefiniujesz metodę `+`, możesz użyć składni `x += y` w instancji klasy (`x`). W efekcie wyrażenie zostanie automatycznie rozwinięte do postaci `x = x + y`. To z kolei jest po prostu „lukrem" składniowym w przypadku `x = x.+(y)`, czyli wywołania metody `+`.

„Oficjalnie" z tymi technikami skrótów zaznajomisz się w rozdziale 7. Na razie nie wahaj się wypróbować ich w narzędziu irb.

W momencie dodania obiektu do stosu ❷ operacja jest obsługiwana przez przekazanie obiektu do tablicy `@stack`, czyli dodanie go na końcu (dostęp do tej tablicy jest uzyskiwany za pomocą wywołania metody `stack`, która zapewnia, że będzie mieć

miejsce inicjalizacja w postaci pustej tablicy, gdy obiekt będzie dodawany po raz pierwszy). Usuwanie obiektu ze stosu ❸ uwzględnia zdejmowanie elementu z tablicy, czyli usuwanie go z jej końca (`push` i `pop` są metodami instancji klasy `Array`; pojawią się one ponownie przy omawianiu w rozdziale 10. obiektów kontenera, w tym tablic).

A zatem moduł `Stacklike` implementuje podobieństwo do stosu przez selektywne wdrażanie zachowań, które istnieją już w przypadku obiektów `Array`: dodaj element na końcu tablicy i usuń element z końca. Tablice są bardziej wszechstronne niż stosy. Stos nie pozwala na wszystko, co oferuje tablica. Na przykład możesz usuwać elementy z tablicy w dowolnej kolejności. Z kolei z założenia jedynym elementem, który możesz usunąć ze stosu, jest element dodany jako ostatni. Tablica umożliwia jednak wszystko to co stos. Dopóki nie zażądamy od tablicy, aby wykonała coś, co nie przypomina działania stosu, sensownym jest użycie jej jako czegoś w rodzaju agenta lub proxy na potrzeby działań dodawania/usuwania wyraźnie przypominających działanie stosu.

Dysponujemy teraz modułem, który implementuje działanie przypominające działanie stosu: utrzymywanie listy elementów, dzięki czemu na jej końcu mogą być dodawane nowe elementy, a elementy dodane jako ostatnie mogą być usuwane. Oto następne pytanie: do czego można użyć takiego modułu?

### 4.1.2. Dodawanie modułu do klasy

Jak pokazano, moduły nie zawierają instancji, dlatego nie jest możliwe następujące działanie:

```
s = Stacklike.new ←——— Niepoprawnie! Nie istnieje taka metoda.
```

Aby utworzyć instancje (obiekty), potrzebujesz klasy. Ponadto aby takie obiekty przypominały stos, niezbędne będzie dodanie modułu `Stacklike` do takiej klasy. Ale do jakiej klasy? Prawdopodobnie najbardziej oczywistą rzeczą podobną do stosu będzie klasa `Stack`. Zapisz kod z listingu 4.2 w pliku *stack.rb* w tym samym katalogu co plik *stacklike.rb*.

**Listing 4.2. Dodawanie modułu Stacklike do klasy Stack**

```
require_relative "stacklike"
class Stack
  include Stacklike  ←——❶
end
```

Efektem końcowym klasy `Stack` w powyższym listingu jest instrukcja `include` ❶, która powoduje dodanie modułu `Stacklike`. Zapewnia to, że instancje klasy `Stack` przejawiają zachowania zdefiniowane w module `Stacklike`.

Zauważ, że nazwa klasy jest rzeczownikiem, natomiast nazwa modułu to przymiotnik. Choć żaden z tych wariantów nie jest obowiązkowy, są one powszechnie stosowane. Ostatecznie uzyskujemy coś w rodzaju predykatu dotyczącego klasy opisanego w języku mówionym: *obiekty stosu przypominają stos*. Odpowiada temu następujący kod:

```
class Stack
  include Stacklike
end
```

**Porównanie składni metod require/load i instrukcji include**

Być może zauważyłeś, że podczas korzystania z metody `require` lub `load` nazwa wymaganego lub ładowanego elementu jest umieszczana w znakach cudzysłowu. Jednak w przypadku instrukcji `include` (i `prepend`) tak nie jest. Wynika to stąd, że metody `require` i `load` jako swoje argumenty pobierają łańcuchy, natomiast instrukcja `include` pobiera nazwę modułu w postaci stałej. Ujmując to w bardziej fundamentalny sposób: jest to spowodowane tym, że obie metody odnajdują i ładują pliki dyskowe, natomiast instrukcje `include` i `prepend` realizują w pamięci obszaru programu operację, która nie ma nic wspólnego z plikami. Częstym działaniem jest żądanie opcji za pomocą metody `require`, a następnie dołączenie przy użyciu instrukcji `include` modułu definiowanego przez opcję. A zatem te dwie operacje są często ze sobą powiązane, choć całkowicie różnią się od siebie.

Aby przekonać się, jak to wszystko wygląda w praktyce, utwórzmy obiekt `Stack` i przetestujmy go. Kod w listingu 4.3 tworzy ten obiekt i wykonuje na nim operacje. Kod możesz umieścić na końcu pliku *stack.rb*.

**Listing 4.3. Tworzenie i używanie instancji klasy Stack**

```
s = Stack.new                                  ❶
s.add_to_stack("element jeden")
s.add_to_stack("element dwa")                  ❷
s.add_to_stack("element trzy")
puts "Obiekty znajdujące się obecnie na stosie:"
puts s.stack
taken = s.take_from_stack                      ❸
puts "Usunięto następujący obiekt:"
puts taken
puts "Pozostały na stosie:"
puts s.stack
```

**Wywołanie metody `puts` dla tablicy powoduje z kolei wywołanie każdego jej elementu**

Powyższy listing rozpoczyna się od niewinnie wyglądającej operacji tworzenia instancji ❶ nowego obiektu `Stack`, który jest przypisywany do zmiennej `s`. Po utworzeniu obiekt ten zawiera informacje o działaniach, jakie może zrealizować, gdy zażądamy od niego wykonania działań związanych ze stosem. Jest to możliwe dzięki temu, że klasa obiektu dodała moduł `Stacklike`. Reszta kodu obejmuje żądanie od obiektu pokonania pewnych przeszkód podobnych jak w przypadku stosu, czyli wykonania następujących operacji: dodanie do samego siebie elementów (łańcuchów) ❷ i usunięcie we własnym zakresie ostatniego elementu ❸. Od obiektu wymagane jest, aby w trakcie realizowania działań zgłaszał swój stan.

Pora uruchomić program. Oto przebieg wywołania pliku `stack.rb` wraz z uzyskanymi danymi wyjściowymi:

```
$ ruby stack.rb
Obiekty znajdujące się obecnie na stosie:
element jeden
element dwa
element trzy
Usunięto następujący obiekt:
element trzy
```

```
Pozostały na stosie:
element jeden
element dwa
```

Z wystarczającą pewnością obiekt `Stack` „wie”, jak postąpić. Jak już wspomniano, obiekt ten przypomina stos.

Klasa `Stack` jest nie najgorsza, ale może przywołać na myśl następujące pytanie: „dlaczego należy trudzić się tworzeniem modułu?”. Ostatecznie byłoby możliwe umieszczenie całej funkcjonalności modułu `Stacklike` bezpośrednio w klasie `Stack` bez tworzenia modułu. Listing 4.4 prezentuje możliwy wygląd takiej klasy.

**Listing 4.4. Przebudowanie klasy Stack bez użycia modułu**

```
class Stack
  attr_reader :stack
  def initialize
    @stack = []
  end
  def add_to_stack(obj)
    @stack.push(obj)
  end
  def take_from_stack
    @stack.pop
  end
end
```

**Metoda konstruktora: zmienna instancji `@stack` mogła nie zostać jeszcze zainicjowana, dlatego ustawiana jest za pomocą operatora `=`, a nie operatora `||=`**

Jak się okaże, dodanie kodu z listingu 4.3 do kodu z listingu 4.4 i wykonanie go przez interpreter języka Ruby spowoduje uzyskanie takich samych wyników jak w przypadku implementacji, która bazuje na module.

Zanim stwierdzisz, że moduły są niepotrzebne, pamiętaj o tym, co zapewnia modularyzacja: umożliwia ona zastosowanie ogólnego pojęcia, takiego jak podobieństwo do stosu, w kilku przypadkach, a nie tylko w jednym.

A zatem co jeszcze przypomina stos?

### 4.1.3. Bardziej zaawansowane użycie modułów

Wcześniej podano kilka następujących przykładów: talerze, papier do drukarki itd. Skorzystajmy z nowego przykładu zapożyczonego ze świata legendy miejskiej.

Wiele osób jest przekonanych, że jeśli ktoś jest pierwszym pasażerem dokonującym odprawy przed lotem, to jego bagaż zostanie wyładowany z samolotu jako ostatni. Praktyczne doświadczenie podpowiada, że wcale tak to nie wygląda. Aby jeszcze poćwiczyć działania ze stosem, dowiedzmy się, jak wyglądałby model w języku Ruby dotyczący właściwego ładunku lotniczego, z którym związana jest ta legenda miejska.

W celu przeprowadzenia dość dokładnego modelowania zdefiniujemy następujące elementy:

- Podstawowa klasa `Suitcase`: element zastępczy (lub fragment kodu), który umożliwia tworzenie obiektów walizek umieszczanych w luku bagażowym.
- Klasa `CargoHold` z dwoma metodami: `load_and_report` i `unload`.

- Metoda `load_and_report` wyświetla komunikat informujący o dodawaniu walizki do luku bagażowego, a ponadto przekazuje numer identyfikacyjny obiektu walizki, który ułatwi śledzenie tego, co się dzieje z każdą walizką.
- Metoda `unload` wywołuje metodę `take_from_stack`, która mogłaby zostać wywołana bezpośrednio, ale nazwa metody `unload` (*wyładuj*) bardziej nadaje się do opisania procesu usuwania walizki z luku bagażowego.

Kod z listingu 4.5 umieść w pliku *cargohold.rb* i uruchom go.

**Listing 4.5. Użycie modułu Stacklike po raz drugi dla innej klasy**

```
require_relative "stacklike"
class Suitcase
end
class CargoHold
  include Stacklike          ❶
  def load_and_report(obj)   ❷
    print "Ładowanie obiektu"
    puts obj.object_id
    add_to_stack(obj)        ❸
  end
  def unload
    take_from_stack          ❹
  end
end
ch = CargoHold.new           ❺
sc1 = Suitcase.new
sc2 = Suitcase.new
sc3 = Suitcase.new
ch.load_and_report(sc1)
ch.load_and_report(sc2)
ch.load_and_report(sc3)
first_unloaded = ch.unload
print "Pierwsza walizka wyładowana z samolotu to...."
puts first_unloaded.object_id
```

Zasadniczo kod programu z powyższego listingu nie różni się zbytnio od kodu z listingów 4.2 i 4.3 (zostały one kolejno zapisane w pliku *stack.rb*). Kod jest zgodny w dużym stopniu z następującą procedurą: dodawanie modułu `Stacklike` do klasy ❶, tworzenie jej instancji ❺ oraz dodawanie do tej instancji elementów ❸ i usuwanie ich z niej ❹ (w tym przypadku luk bagażowy pod względem działania przypomina stos). Tak jak w przypadku innego kodu programu kod zapewnia również raportowanie dotyczące bieżącego stanu stosu ❷.

Dane wyjściowe programu obsługującego luk bagażowy mają następującą postać (pamiętaj, że walizki są przywoływane za pomocą identyfikatorów liczbowych ich obiektów, które mogą różnić się w używanym systemie):

```
Ładowanie obiektu 1001880
Ładowanie obiektu 1001860
Ładowanie obiektu 1001850
Pierwsza walizka wyładowana z samolotu to....1001850
```

Przykład z lukiem bagażowym pokazuje, w jaki sposób możesz użyć dla nowej klasy istniejącego modułu. Choć czasami warto dla nowej domeny opakować metody za pomocą nowych metod z lepszymi nazwami (np. użycie metody `unload` zamiast metody `take_from_stack`), ale uznasz, że ma miejsce zbyt wiele zmian, może to być oznaką tego, że moduł nie jest odpowiednio dopasowany.

W następnym podrozdziale połączymy ze sobą kilka elementów, które zostały wcześniej osobno omówione w większym lub mniejszym stopniu: wywołania metod (wysyłanie komunikatów), obiekty i ich status jako instancji klas oraz dodawanie modułów do klas. Wszystkie te zagadnienia łączą się w procesie, w ramach którego obiekt w momencie wysyłania komunikatu szuka metody (może ją znaleźć albo nie) do wykonania, o nazwie zgodnej z nazwą komunikatu.

## 4.2. Moduły, klasy i wyszukiwanie metody

Wiesz już, że po odebraniu komunikatu przez obiekt zamierzonym (i standardowym) efektem jest wykonanie metody o takiej samej nazwie jak nazwa komunikatu w klasie obiektu lub jej superklasie (i dalej aż do klasy `Object`, a nawet `BasicObject`) bądź w module, który został dodany do dowolnej z tych klas. Jak to jednak się odbywa? Co ma miejsce w niejednoznacznych przypadkach, na przykład w sytuacji, gdy klasa i dodany moduł definiują metodę o danej nazwie? Który z tych elementów wybiera obiekt do wykonania?

Warto udzielić dokładnej odpowiedzi na te pytania. Nieprecyzyjne objaśnienia tego, co ma miejsce, są łatwe do uzyskania. Czasami są one nawet wystarczające: stwierdzając na przykład: „Ten obiekt zawiera metodę `push`", możesz z powodzeniem przekazać to, o czym próbujesz poinformować, nawet pomimo tego, że obiekty nie zawierają metod, lecz raczej znajdują je przez przeszukiwanie klas i modułów.

Nieprecyzyjne objaśnienie nie będzie jednak mogło zostać wykorzystane w szerszym zakresie. Nie ułatwi ono zrozumienia tego, co się dzieje w bardziej złożonych przypadkach, a ponadto nie będzie wsparciem podczas projektowania własnego kodu. Najlepszym rozwiązaniem jest stwierdzanie, co faktycznie się dzieje w momencie wysyłania komunikatów do obiektów.

Na szczęście przebiega to w prosty sposób.

### 4.2.1. Demonstracja podstaw wyszukiwania metod

Aby dobrze zrozumieć, jak obiekty znajdują metody, skorzystajmy ponownie z odniesień do świata rzeczywistego i utwórzmy klasy i moduły o prostych nazwach, takich jak `C` i `M`. Ułatwi to skoncentrowanie się na logice i mechanizmie wyszukiwania metod bez konieczności jednoczesnego myślenia o modelowaniu domeny ze świata rzeczywistego. Zostaną też utworzone metody, które wyświetlają jedynie komunikat informujący o tym, że zostały wywołane. Ułatwi to śledzenie kolejności wyszukiwania metod.

Przyjrzyj się programowi z listingu 4.6.

**Listing 4.6. Demonstracja włączania modułu i dziedziczenia**

```
module M
  def report
    puts "Metoda report w module M"
  end
end
class C
  include M
end
class D < C
end
obj = D.new
obj.report
```

Metodę instancji `report` zdefiniowano w module `M`. Moduł dodano do klasy `C`. Klasa `D` jest podklasą klasy `C`, a obiekt `obj` to instancja klasy `D`. Za pośrednictwem takiej kaskady obiekt `obj` uzyskuje dostęp do metody `report`.

Określenie „*uzyskuje dostęp do* metody", tak jak „*zawiera* metodę", to jednak mało precyzyjny sposób opisywania operacji. Spróbujmy bardziej doprecyzować proces, rozważając go z punktu widzenia obiektu.

### WYSZUKIWANIE METOD Z PUNKTU WIDZENIA OBIEKTU

Wyobraź sobie, że jesteś obiektem, a ktoś wysyła Ci komunikat. Musisz określić, w jaki sposób na niego odpowiedzieć, albo czy w ogóle *możesz* na niego odpowiedzieć. Oto porcja „toku rozumowania" obiektu:

> Jestem obiektem języka Ruby i wysłano mi komunikat `report`. Muszę spróbować znaleźć metodę o nazwie `report` w mojej ścieżce wyszukiwania metod. Jeśli metoda `report` istnieje, znajduje się w klasie lub module.
>
> Jestem instancją klasy o nazwie `D`. Czy klasa `D` definiuje metodę instancji `report`?
>
> *Nie.*
>
> Czy klasa `D` dodaje jakiekolwiek moduły?
>
> *Nie*.
>
> Czy superklasa `C` klasy `D` definiuje metodę instancji `report`?
>
> *Nie.*
>
> Czy klasa `C` dodaje jakiekolwiek moduły?
>
> *Tak, moduł `M`.*
>
> Czy moduł `M` definiuje metodę `report`?
>
> *Tak.*
>
> Dobrze! Wykonam tę metodę.

Wyszukiwanie kończy się w momencie znalezienia szukanej metody lub zwrócenia warunku błędu, gdy metoda nie zostanie znaleziona. Warunek błędu jest wyzwalany przez specjalną metodę o nazwie `method_missing`, która jest wywoływana w ostateczności w przypadku komunikatów niedopasowanych w inny sposób. Metodę tę możesz

przesłonić (czyli zdefiniować ją na nowo w jednej z własnych klas lub modułów), aby dla takich komunikatów zdefiniować niestandardowe zachowanie (więcej związanych z tym szczegółów zamieszczono w podrozdziale 4.3).

Przejdźmy teraz z „toku rozumowania" obiektu do konkretów scenariusza wyszukiwania metod, a w szczególności do tego, jaki jest zasięg tego procesu.

### JAKI JEST ZASIĘG PROCESU WYSZUKIWANIA METOD?

Ostatecznie każdy obiekt w języku Ruby jest instancją klasy pochodzącej od wielkiej klasy z „niebios", czyli `BasicObject`. Choć po drodze może wystąpić wiele klas i modułów, maksymalnym zasięgiem wyszukiwania metody zawsze będzie klasa `BasicObject`. Przypomnij sobie jednak, że w jej przypadku chodzi o to, że oferuje ona kilka metod instancji. Poznanie klasy `BasicObject` nie zapewni zbyt wielu informacji o metodach współużytkowanych przez wszystkie obiekty języka Ruby.

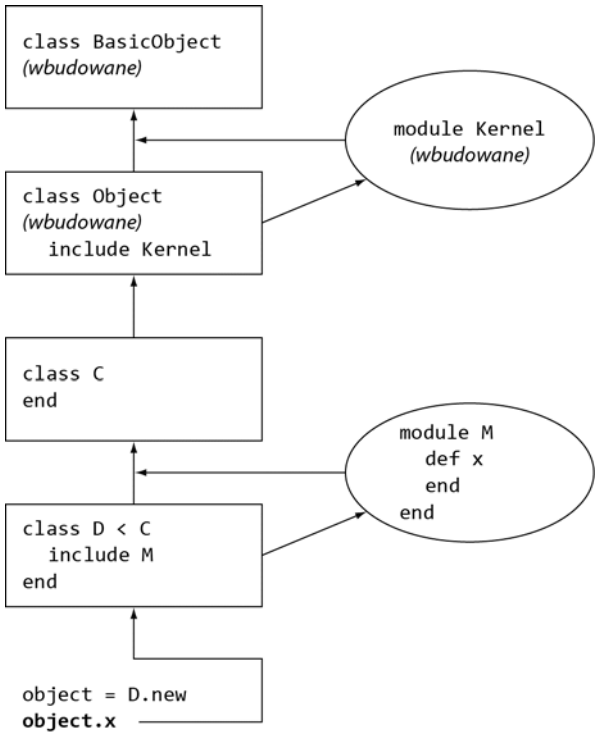
Aby zrozumieć typowe zachowanie i funkcjonalność wszystkich obiektów języka Ruby, musisz „zejść z chmur" i przyjrzeć się klasie `Object`, a nie klasie `BasicObject`. Dokładniej rzecz biorąc, niezbędne jest sprawdzenie modułu `Kernel` dodawanego przez klasę `Object`. Właśnie w module `Kernel` (jądro) zdefiniowano większość podstawowych obiektów metod języka Ruby. Ponieważ klasa `Object` dodaje ten moduł, wszystkie instancje tej klasy oraz każdy jej potomek mają dostęp do metod instancji w module `Kernel`.

Załóżmy, że jesteś obiektem i próbujesz znaleźć metodę do wykonania na podstawie odebranego komunikatu. Jeśli szukałeś w module `Kernel` i klasie `BasicObject`, ale metoda nie została tam znaleziona, nie uda się to (możliwe jest dodanie modułów do klasy `BasicObject`, a tym samym zapewnienie wszystkim obiektom dodatkowego potencjalnego źródła metod; trudno jednak przytoczyć sytuację, w której zostałoby to zrealizowane).

Na rysunku 4.1 zilustrowano ścieżkę wyszukiwania metod z wcześniejszego przykładu (obiekt klasy `D`), analizując ją aż do samej góry. W przykładzie wyszukiwanie metody kończy się powodzeniem w module `M`. Na rysunku pokazano, jaki byłby zasięg wyszukiwania przez obiekt, jeśli nie znalazłby metody w tym miejscu. Po wysłaniu komunikatu `x` do obiektu rozpoczyna się wyszukiwanie metody, w ramach którego zgodnie z widocznymi strzałkami napotykane są różne klasy i dodane moduły.

Wewnętrzne definicje klas `BasicObject` i `Object` oraz modułu `Kernel` utworzono w języku C. Aby zorientować się, jak wygląda interakcja między nimi, przyjrzyj się makiecie ich relacji w języku Ruby:

```
class BasicObject
  # w tym miejscu znajduje się zaledwie siedem definicji metod
end
module Kernel
  # w tym miejscu znajduje się ponad 100 definicji metod!
end
class Object < BasicObject
  # w tym miejscu znajduje się jedna lub dwie metody prywatne,
  # ale podstawową kwestią jest dodanie modułu Kernel
  include Kernel
end
```




**Rysunek 4.1.** Instancja klasy D szuka metody x w swojej ścieżce wyszukiwania metody

`Object` to podklasa klasy `BasicObject`. Każda klasa pozbawiona jawnie określonej superklasy jest podklasą klasy `Object`. Oto dowód tego uzyskany w sesji narzędzia irb:

```
>> class C
>> end
=> nil
>> C.superclass
=> Object
```

Wśród przodków każdej klasy znajduje się klasa `Object`, a tym samym moduł `Kernel` i klasa `BasicObject`. Oczywiście pojawia się paradoks taki, że klasa `BasicObject` jest klasą `Object`, klasa `Object` jest klasą `Class`, a ta klasa — klasą `Object`. Jak jednak pokazano wcześniej, odrobina kolistości w modelu klas ma na celu zainicjowanie hierarchii, która po „wprawieniu w ruch" funkcjonuje w przejrzysty i logiczny sposób.

### *4.2.2. Definiowanie tej samej metody więcej niż raz*

W rozdziale 3. wspomniano, że w przypadku zdefiniowania metody po raz drugi w obrębie tej samej klasy druga definicja ma pierwszeństwo przed pierwszą. To samo dotyczy modułów. Reguła sprowadza się do tego, że w dowolnym momencie w klasie lub module może występować tylko jedna metoda o danej nazwie. Jeśli w klasie `Bank`
`↪Account` znajduje się metoda o nazwie `calculate_interest`, a ponadto utworzysz drugą metodę o identycznej nazwie w tej samej klasie, zignoruje ona wszystkie informacje o pierwszej wersji metody.

W taki właśnie sposób klasy i moduły dbają o porządek w swoim obrębie. Gdy jednak spojrzy się na to z punktu widzenia obiektu, kwestia uzyskiwania dostępu do dwóch lub większej liczby metod o takiej samej nazwie stanie się bardziej złożona.

Metody obiektu mogą pochodzić z dowolnej liczby klas i modułów. Prawdą jest, że dowolna klasa lub moduł mogą zawierać tylko jedną metodę `calculate_interest` (takiej nazwy można użyć dla przykładu). Obiekt może mieć jednak wiele metod `calculate_interest` w swojej ścieżce wyszukiwania metod, ponieważ uwzględnia ona wiele klas i modułów.

W dalszym ciągu reguła dotycząca obiektów jest analogiczna do reguły obowiązującej dla klas i modułów: w dowolnym momencie obiekt może rozpoznać tylko jedną wersję metody o danej nazwie. Jeśli ścieżka wyszukiwania metod obiektu obejmuje dwie lub większą liczbę metod o tej samej nazwie, „zwycięzcą" zostaje pierwsza napotkana, która jest wykonywana.

Listing 4.7 prezentuje sytuację, w której w ścieżce wyszukiwania metod obiektu znajdują się dwie wersje metody: pierwsza wersja występuje w klasie obiektu, a druga w module dodanym przez tę klasę.

**Listing 4.7. Dwie metody o takiej samej nazwie w jednej ścieżce wyszukiwania**

```
module InterestBearing
  def calculate_interest
    puts "Element zastępczy! Bieżący moduł to InterestBearing."
  end
end
class BankAccount
  include InterestBearing
def calculate_interest
    puts "Element zastępczy! Bieżąca klasa to BankAccount."
    puts "Przesłaniana jest metoda calculate_interest..."
    puts "zdefiniowana w module InterestBearing."
  end
end
account = BankAccount.new
account.calculate_interest
```

Po uruchomieniu kodu z tego listingu uzyskujemy następujące dane wyjściowe:

```
Element zastępczy! Bieżąca klasa to BankAccount.
Przesłaniana jest metoda calculate_interest...
zdefiniowana w module InterestBearing.
```

Dwie metody `calculate_interest` znajdują się w ścieżce wyszukiwania metod obiektu `c`. Proces wyszukiwania natrafia jednak na klasę `BankAccount` (klasa obiektu `account`) przed napotkaniem modułu `InterestBearing` (dodany przez klasę `BankAccount`). A zatem wykonywana metoda podana w raporcie jest metodą zdefiniowaną w klasie `BankAccount`.

W następującej sytuacji obiekt może mieć dwie metody o takiej samej nazwie w swojej ścieżce wyszukiwania metod: gdy klasa dodaje dwie lub większą liczbę modułów, więcej niż jeden z nich implementuje wyszukiwaną metodę. W takim przypadku moduły są przeszukiwane w kolejności odwrotnej do kolejności ich dołączenia. Oznacza to, że jako pierwszy jest przeszukiwany moduł dodany jako ostatni. Jeśli okaże

się, że taki moduł zawiera metodę o takiej samej nazwie co nazwa metody we wcześniej dodanym module, pierwszeństwo będzie mieć wersja metody w nowo dodanym module, ponieważ nowszy moduł znajduje się bliżej początku ścieżki wyszukiwania metod obiektu.

Na przykład rozważ sytuację, w której w dwóch modułach `M` i `N` (dla przykładu zachowamy dość schematyczną postać) zdefiniowano metodę `report`. Ponadto oba moduły dodano do klasy `C`, co prezentuje listing 4.8.

**Listing 4.8. Dodawanie dwóch modułów zawierających zdefiniowaną metodę o takiej samej nazwie**

```
module M
  def report
    puts "Metoda report w module M"
  end
end
module N
  def report
    puts "Metoda report w module N"
  end
end
class C
  include M
  include N
end
```

Jakie działanie wykonuje instancja tej klasy w momencie wysłania jej komunikatu `report`, gdy następuje przejście ścieżki wyszukiwania w poszukiwaniu pasującej metody? Pytanie zadajmy za pomocą następującego kodu:

```
c = C.new
c.report
```

Odpowiedź brzmi: `"Metoda report w module N"`. Pierwsza metoda `report` napotkana w ścieżce wyszukiwania metod obiektu `c` jest metodą zawartą w *module dodanym jako ostatni*. W tym przypadku oznacza to moduł `N`. Z tego powodu metoda `report` tego modułu „zwycięża" w rywalizacji z metodą modułu `M` o takiej samej nazwie.

Należy tutaj zauważyć, że dołączenie modułu więcej niż raz nie spowoduje żadnego efektu.

### DOŁĄCZENIE MODUŁU WIĘCEJ NIŻ RAZ

Przyjrzyj się następującemu przykładowi bazującemu na poprzednim przykładzie, lecz tym razem po module `N` drugi raz dołączono moduł `M`:

```
class C
  include M
  include N
  include M
end
```

Możesz oczekiwać, że po uruchomieniu metody `report` uzyskasz wersję modułu `M`, ponieważ został on dołączony jako ostatni. Jednak ponowne dołączenie modułu niczego

nie zmienia. Ze względu na to, że moduł `M` znajduje się już w ścieżce wyszukiwania, druga instrukcja `include M` nie da żadnego efektu. Moduł `N` nadal jest traktowany jak moduł dołączony jako ostatni:

```
c = C.new
c.report  ←——— Dane wyjściowe: metoda report w module N
```

Podsumowując: możesz modyfikować ścieżki wyszukiwania metod obiektów, ale tylko do pewnego stopnia.

We wszystkich dotychczasowych przykładach do dodawania modułów używano instrukcji `include`. Pora przywrócić do omawiania instrukcję `prepend`.

### 4.2.3. Sposób działania instrukcji prepend

Każdorazowo podczas dołączania modułu do klasy za pomocą instrukcji `include` wpływasz na to, co się dzieje, gdy instancje tej klasy muszą określić nazwy metod dla komunikatów. To samo dotyczy instrukcji `include`. Różnica polega na tym, że w przypadku dołączania modułu do klasy przy użyciu instrukcji `prepend` obiekt sprawdza najpierw ten moduł, zanim sprawdzi klasę.

Oto przykład:

```
module MeFirst
  def report
    puts "Powitanie od modułu!"
  end
end
class Person
  prepend MeFirst
  def report
    puts "Powitanie od klasy!"
  end
end
p = Person.new
p.report
```

Dane wyjściowe to `"Powitanie od modułu!"`. Dlaczego? Wynika to z tego, że moduł `MeFirst` został dołączony do klasy za pomocą instrukcji `prepend`. Oznacza to, że instancja klasy sprawdzi najpierw moduł przy próbie znalezienia metody o nazwie `report`. Jeśli zostałaby użyta instrukcja `include`, klasa zostałaby przeszukana przed modułem, wersja metody `report` klasy zostałaby „zwycięzcą”.

Różnicę między instrukcjami `include` i `prepend` zaprezentowano zarówno na rysunku 4.2, jak i na liście kolejno uporządkowanych przodków klasy, czyli wszystkich klas i modułów, w których instancja klasy będzie szukać metod. Oto przodkowie klasy `Person` z ostatniego przykładu wyświetleni w narzędziu irb:

```
> Person.ancestors
=> [MeFirst, Person, Object, Readline, Kernel, BasicObject]
```

Modyfikuj teraz przykład w celu zastosowania instrukcji `include` zamiast instrukcji `prepend`. Mają miejsce dwie rzeczy. Po pierwsze, zmieniają się dane wyjściowe:

```
Powitanie od klasy!
```

Po drugie, zmienia się kolejność przodków:

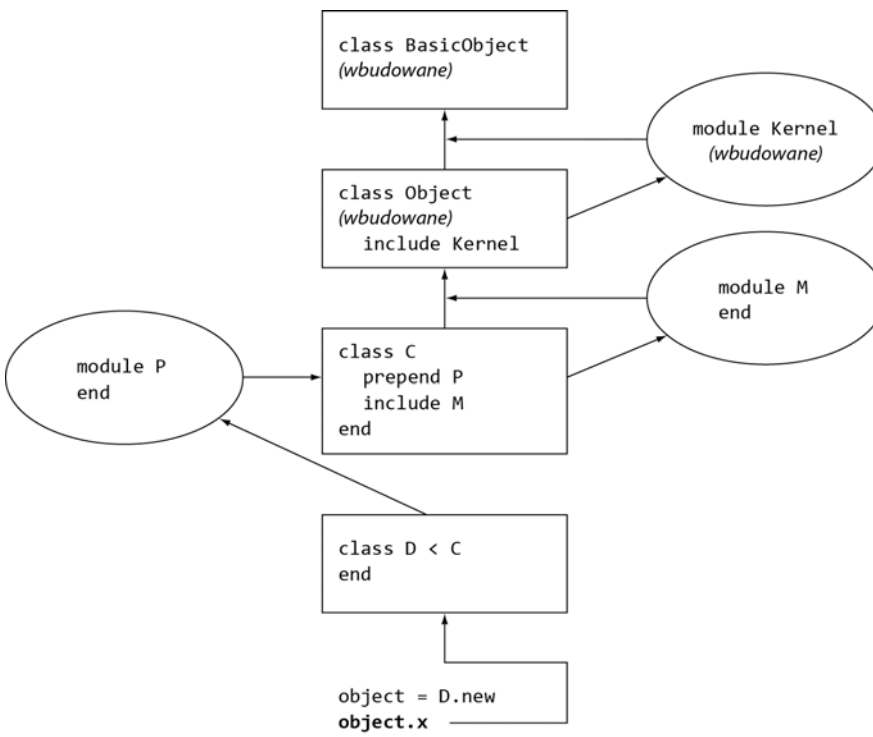
```
> Person.ancestors
=> [Person, MeFirst, Object, Readline, Kernel, BasicObject]
```

Oczywiście nazwa `MeFirst` przestaje mieć tutaj sens, ale ogólnie wiadomo, o co chodzi.

Instrukcji `prepend` możesz użyć, aby wersja modułu jednej lub większej liczby metod miała pierwszeństwo przed wersjami zdefiniowanymi w danej klasie. Jak wcześniej wspomniano, instrukcja ta jest nowością w języku Ruby 2.0. Nie będzie ona często spotykana, przynajmniej na razie. Warto jednak wiedzieć o jej istnieniu, aby w razie potrzeby skorzystać z niej, a także aby znać jej przeznaczenie w przypadku napotkania tej instrukcji w kodzie napisanym przez kogoś innego.

### 4.2.4. Podsumowanie reguł wyszukiwania metod

Na rysunku 4.2 zilustrowano podstawowe reguły zarządzające wyszukiwaniem metod i uporządkowaniem w ścieżce wyszukiwania metod w języku Ruby 2.




**Rysunek 4.2.** Instancja klasy D szuka metody x w swojej ścieżce wyszukiwania metod, uwzględniając zarówno moduły dołączone za pomocą instrukcji include, jak i instrukcji prepend

Aby dla komunikatu określić metodę, obiekt szuka metody w następujących miejscach:

1. Moduły dołączone do klasy za pomocą instrukcji `prepend` (wyszukiwanie odbywa się w kolejności odwrotnej do kolejności dołączania).
2. Własna klasa obiektu.

3. Moduły dołączone do własnej klasy obiektu za pomocą instrukcji `include` (wyszukiwanie odbywa się w kolejności odwrotnej do kolejności dołączania).
4. Moduły dołączone do superklasy własnej klasy obiektu za pomocą instrukcji `prepend`.
5. Superklasa własnej klasy obiektu.
6. Moduły dołączone do superklasy własnej klasy obiektu za pomocą instrukcji `include`.
7. Powyższe kroki dotyczą też pozostałych klas aż do klas `Object` (oraz dodanego do niej modułu `Kernel`) i `BasicObject`.

Szczególną uwagę zwróć na punkt, w którym moduły są przeszukiwane pod kątem metod w kolejności odwrotnej do kolejności dołączania za pomocą instrukcji `prepend` lub `include`. Zapewnia to przewidywalne zachowanie w sytuacji, gdy klasa dodaje dwa moduły definiujące tę samą metodę.

**A co z metodami pojedynczego obiektu?**

W rozdziale 3. zaprezentowano już *metodę pojedynczego obiektu*, czyli metodę zdefiniowaną bezpośrednio w obiekcie (`def obj.talk`). Możesz się zastanawiać, gdzie takie metody znajdują się w ścieżce wyszukiwania metod. Otóż występują one w specjalnej klasie utworzonej wyłącznie w celu ich przechowywania. Mowa o klasie pojedynczych obiektów obiektu. W dalszej części książki przyjrzymy się dokładniej tego rodzaju klasom. Zostaną one umiejscowione w modelu wyszukiwania metod.

Dostępna jest w pewnym stopniu specjalistyczna, ale przydatna i często stosowana technika, która służy do jawnej nawigacji w obrębie ścieżki wyszukiwania. Jest to słowo kluczowe `super`.

### 4.2.5. Nawigacja w obrębie ścieżki wyszukiwania metod za pomocą słowa kluczowego super

W definicji metody możesz użyć słowa kluczowego `super` do przeskoczenia do następnej, wyżej położonej definicji w ścieżce wyszukiwania metod dla aktualnie wykonywanej metody.

Listing 4.9 prezentuje prosty przykład (po nim zajmiemy się kwestią: Dlaczego należałoby tak postąpić?).

**Listing 4.9. Użycie słowa kluczowego super do przejścia jeden poziom wyżej w ścieżce wyszukiwania**

```
module M
  def report                    ←❶
    puts "Metoda report w module M"
  end
end
class C
  include M
  def report                    ←❷
    puts "Metoda report w klasie C"
    puts "Zostanie wywołana następna, wyżej położona metoda report..."
```

```
    super          ←——❸
    puts "Powrót z wywołania słowa kluczowego super."
  end
end
c = C.new
c.report           ←——❹
```

Dane wyjściowe uzyskane po uruchomieniu kodu z listingu 4.9 są następujące:

```
Metoda report w klasie C
Zostanie wywołana następna, wyżej położona metoda report...
Metoda report w module M
Powrót z wywołania słowa kluczowego super.
```

Instancja klasy `C` (mianowicie obiekt `c`) odbiera komunikat `report` ❹. Proces wyszukiwania metody rozpoczyna się od klasy `C` obiektu `c`. Z całą pewnością istnieje w niej metoda `report` ❷, która jest wykonywana.

Wewnątrz metody znajduje się wywołanie słowa kluczowego `super` ❸. Oznacza to, że nawet pomimo tego, że obiekt znalazł metodę odpowiadającą komunikatowi (`report`), musi kontynuować szukanie i znaleźć następne dopasowanie. W tym przypadku kolejne dopasowanie metody `report` to jej wersja zdefiniowana w module `M` ❶.

Zauważ, że jeśli nie istniałoby dopasowanie `C#report`, dopasowanie `M#report` byłoby pierwszym zidentyfikowanym podczas wyszukiwania metody `report`. Słowo kluczowe `super` zapewnia sposób wywołania tego, co byłoby właściwą wersją metody w sytuacjach, w których została ona później przesłonięta w ścieżce wyszukiwania. Dlaczego należałoby tak postąpić?

Czasami, a zwłaszcza podczas tworzenia podklasy, metoda w istniejącej klasie realizuje niemal to, czego się oczekuje, ale nie do końca. W przypadku słowa kluczowego `super` możesz zapewnić sobie najlepsze cechy obu „światów" przez opakowanie oryginalnej metody. Ilustruje to listing 4.10.

**Listing 4.10. Użycie słowa kluczowego super do opakowania metody w podklasie**

```
class Bicycle
  attr_reader :gears, :wheels, :seats
  def initialize(gears = 1)        ←——❶
    @wheels = 2
    @seats = 1
    @gears = gears
  end
end
class Tandem < Bicycle
  def initialize(gears)
    super
    @seats = 2       ←——❷
  end
end
```

Słowo kluczowe `super` zapewnia przejrzysty sposób pozwalający sprawić, że tandem będzie prawie jak zwykły rower. Zmieniane jest tylko to, co wymaga modyfikacji (liczba siodełek ❷), a to słowo kluczowe wyzwala wcześniej wspomnianą metodę `initialize` ❶,

która ustawia wartości domyślne podobne jak w przypadku zwykłego roweru dla innych właściwości tandemu.

W momencie wywoływania słowa kluczowego `super` nie jest jawnie zapewniany argument `gears`, który jest przekazywany metodzie `initialize`. Niemniej jednak gdy w klasie `Bicycle` wywoływana jest oryginalna metoda `initialize`, widoczne są wszystkie argumenty udostępnione wersji w klasie `Tandem`. Jest to specjalne działanie słowa kluczowego `super`. Obsługuje ono argumenty w następujący sposób:

- Słowo kluczowe `super` wywołane bez listy argumentów (pusta lub w innej niewłaściwej postaci) powoduje automatyczne skierowanie przekazanych argumentów do metody, z której zostało ono wywołane.
- Słowo kluczowe `super` wywołane z pustą listą argumentów (`super()`) nie wysyła żadnych argumentów do wyżej położonej metody nawet wtedy, gdy bieżącej metodzie przekazano argumenty.
- Słowo kluczowe `super` wywołane z konkretnymi argumentami (`super(a,b,c)`) powoduje wysłanie tylko tych argumentów.

Takie nietypowe traktowanie argumentów wynika z tego, że najczęściej występuje pierwszy wariant, w przypadku którego wymagane jest przejście do następnej, wyżej położonej metody z tymi samymi argumentami, jakie zostały odebrane przez metodę, z której wywoływane jest słowo kluczowe `super`. Z tym wariantem powiązana jest najprostsza składnia, czyli wystarczy wpisać słowo `super` (a ponieważ `super` to słowo kluczowe, a nie metoda, może zostać wykorzystane do zapewnienia takiego szczególnego zachowania).

Gdy już wiesz, jak przebiega wyszukiwanie metod, rozważmy, co się dzieje w przypadku niepowodzenia tego procesu.

## 4.3. Metoda method_missing

Moduł `Kernel` zapewnia metodę instancji o nazwie `method_missing`. Metoda jest wykonywana każdorazowo, gdy komunikat odbiera obiekt pozbawiony informacji o tym, jak na niego odpowiedzieć. Jest to komunikat, który nie pasuje do metody w żadnym miejscu ścieżki wyszukiwania metod obiektu:

```
>> o = Object.new
=> #<Object:0x0000010141bbb0>
>> o.blah
NoMethodError: undefined method `blah' for #<Object:0x0000010141bbb0>
```

Z łatwością można przechwycić wywołania brakujących metod. Metoda `method_missing` jest przesłaniana dla poszczególnych pojedynczych obiektów w przypadku obiektu, którego metoda jest wywoływana, bądź w klasie obiektu lub w jednym z przodków tej klasy:

```
>> def o.method_missing(m, *args)                                  ←❶
>>   puts "Nie możesz wywołać #{m} dla tego obiektu. Spróbuj ponownie."
>> end
=> nil
>> o.blah
Nie możesz wywołać blah dla tego obiektu. Spróbuj ponownie.
```

Przy przesłanianiu metody `method_missing` niezbędne jest imitowanie sygnatury metody oryginału ❶. Pierwszym argumentem jest nazwa brakującej metody (komunikat wysłany obiektowi, który nie został przez niego rozpoznany). Parametr `*args` pobiera wszystkie pozostałe argumenty (możesz również dodać specjalny argument w celu powiązania bloku kodu, ale nie przejmujmy się tym do momentu dokładniejszego przyjrzenia się blokom kodu). Pierwszy argument przekazywany jest jako obiekt w postaci symbolu. Aby sprawdzić ten argument lub poddać go analizie, musisz przekształcić go w łańcuch.

Jeśli nawet przesłaniasz metodę `method_missing`, poprzednia definicja jest nadal dostępna za pośrednictwem słowa kluczowego `super`.

### 4.3.1. Łączenie metody method_missing ze słowem kluczowym super

Częstą sytuacją jest wymóg przechwycenia nierozpoznanego komunikatu i natychmiastowego zdecydowania, czy ma być obsługiwany, czy przekazany do oryginalnej metody `method_missing` (lub ewentualnie do pośredniej wersji, jeśli taka została zdefiniowana). Z łatwością możesz to osiągnąć za pomocą słowa kluczowego `super`. Oto przykład typowego wzorca:

```
class Student
  def method_missing(m, *args)
    if m.to_s.start_with?("grade_for_")
      # zwracanie odpowiedniego stopnia na podstawie analizy nazwy metody
    else
      super
    end
  end
end
```

**Przekształcanie symbolu w łańcuch za pomocą metody `to_s` przed przeprowadzeniem testu**

W przypadku powyższego kodu wywołanie na przykład metody `grade_for_english` dla instancji klasy `Student` prowadzi do gałęzi `true` testu instrukcji `if`. Jeśli nazwa brakującej metody nie rozpoczyna się od łańcucha `grade_for`, wybierana jest gałąź `false`, co powoduje wywołanie słowa kluczowego `super`. Wywołanie to skieruje do następnej dowolnej implementacji metody `method_missing` znajdującej się w ścieżce wyszukiwania metod obiektu. Jeśli metoda `method_missing` nie została przesłonięta nigdzie indziej po drodze, słowo kluczowe `super` znajdzie metodę `method_missing` modułu `Kernel` i wykona ją.

Przyjrzyjmy się bardziej złożonemu przykładowi wykorzystania tych technik. Zostanie utworzona klasa `Person`. Zacznijmy od samej góry — od kodu, który ilustruje żądany sposób użycia klasy. Później zaimplementujemy klasę w taki sposób, aby kod działał.

Listing 4.11 prezentuje kod wykorzystujący klasę `Person`.

**Listing 4.11. Przykładowe użycie klasy Person**

```
j = Person.new("Jan")
p = Person.new("Piotr")
g = Person.new("Grzegorz")
r = Person.new("Ryszard")
j.has_friend(p)
```

```
j.has_friend(g)
g.has_friend(p)
r.has_hobby("rings")
Person.all_with_friends(p).each do |person|
  puts "Osoba, z którą przyjaźni się #{person.name}, to #{p.name}"
end
Person.all_with_hobbies("rings").each do |person|
  puts "#{person.name} interesuje się dzwonkami"
end
```

Oczekujemy, że dane wyjściowe tego kodu będą mieć następującą postać:

```
Osoba, z którą przyjaźni się Jan, to Paweł
Osoba, z którą przyjaźni się Grzegorz, to Paweł
Ryszard interesuje się dzwonkami
```

Ogólnie chodzi o to, że dana osoba może mieć przyjaciół i/lub zainteresowania. Co więcej, klasa `Person` umożliwia sprawdzenie wszystkich osób, które mają danego przyjaciela, lub każdej osoby z danym zainteresowaniem. Wyszukiwanie jest realizowane za pomocą metod klasy `all_with_friends` i `all_with_hobbies`.

Wzorzec nazwy metody `all_with_*` wygląda na dobrego kandydata, który może być obsługiwany za pośrednictwem metody `method_missing`. Choć używamy tylko dwóch wariantów tego wzorca (przyjaciele i zainteresowania), jest to rodzaj wzorca, który może być rozwijany do dowolnej liczby nazw metod. Przechwyćmy metodę `method_missing` w klasie `Person`.

W tym przypadku rozpatrywana metoda `method_missing` jest metodą klasy: niezbędne jest przechwycenie brakujących metod wywoływanych w klasie `Person`. A zatem gdzieś po drodze wymagana jest następująca definicja:

```
class Person
  def self.method_missing(m, *args)
  # w tym miejscu znajduje się kod
  end
end
```

**Definiowanie metody bezpośrednio w obiekcie `self`, który jest obiektem klasy `Person`**

Nazwa metody `m` może zaczynać się od podłańcucha `all_with_`. Jeśli tak jest, pozostajemy przy tym. W przeciwnym razie zdajemy się na słowo kluczowe `super` i zezwalamy na obsługę metody przez kod `Kernel #method_missing` (zapamiętaj: klasy są obiektami, dlatego obiekt klasy `Person` ma dostęp do wszystkich metod instancji modułu `Kernel`, w tym do metody `method_missing`).

Oto trochę bardziej zaawansowany (lecz nadal schematyczny) wariant użycia metody `method_missing`:

```
class Person
  def self.method_missing(m, *args)
    method = m.to_s                          ❶
    if method.start_with?("all_with_"        ❷
      # W tym miejscu znajduje się kod obsługi żądania
    else
      super                                  ❸
    end
  end
end
```

Powodem wywołania metody `to_s` ❶ jest to, że nazwa metody (komunikatu) jest przekazywana metodzie `method_missing` w postaci symbolu. Symbole nie mają metody `start_with?`, dlatego konieczne jest przekształcenie symbolu w łańcuch przed sprawdzeniem jego zawartości.

Logika warunkowa ❷ rozgałęzia się zależnie od tego, czy obsługiwany jest komunikat `all_with_*`. W przypadku takiego komunikatu obsługa ma miejsce. W przeciwnym razie następuje przejście do słowa kluczowego `super` ❸.

Dysponując przynajmniej planem odnośnie do metody `method_missing`, opracujmy resztę klasy `Person`. Na podstawie zamieszczonego wcześniej kodu wywołującego na wyższym poziomie możliwe jest wyraźne określenie kilku następujących wymagań:

- Obiekty `Person` śledzą powiązanych z nimi przyjaciół i zainteresowania.
- Klasa `Person` śledzi wszystkie istniejące osoby.
- Każda osoba ma imię.

Drugi z powyższych punktów wynika z faktu, że skierowano już do klasy `Person` pytanie dotyczące listy osób, które mają określone zainteresowania i/lub konkretnych przyjaciół.

Listing 4.12 zawiera implementację części klasy `Person`, które odnoszą się do tych wymagań.

**Listing 4.12. Implementacja głównej logiki klasy Person**

```
class Person
  PEOPLE = []                                  ❶
  attr_reader :name, :hobbies, :friends        ❷
  def initialize(name)
    @name = name
    @hobbies = []                              ❸
    @friends = []
    PEOPLE << self                             ❹
  end
  def has_hobby(hobby)
    @hobbies << hobby
  end                                          ❺
  def has_friend(friend)
    @friends << friend
  end
```

Wszystkie istniejące osoby są umieszczane w tablicy utrzymywanej w stałej `PEOPLE` ❶. W momencie tworzenia instancji obiektu nowej osoby jest ona dodawana do tablicy osób, co jest możliwe dzięki metodzie dołączania do tablicy `<<` ❹. Tymczasem niezbędne są atrybuty odczytujące: `name`, `hobbies` i `friends` ❷. Zapewnienie tych atrybutów pozwala na udostępnianie na zewnątrz ważnych aspektów obiektów `Person`. Atrybuty `hobbies` i `friends` okażą się również przydatne w pełnej implementacji metody `method_missing`.

Metoda `initialize` pobiera nazwę jako swój jedyny argument i zapisuje ją w zmiennej `@name`. Metoda inicjuje również tablice `hobbies` i `friends` ❸. Tablice są ponownie wykorzystywane w metodach `has_hobby` i `has_friend` ❺, które tak naprawdę są jedynie przyjaznymi dla użytkownika elementami opakowującymi te tablice.

Dysponujemy już ilością kodu wystarczającą do zakończenia implementacji metody `Person.method_missing`. Kod zaprezentowano w listingu 4.13 (z uwzględnieniem końcowego separatora `end` dla całej klasy). Używamy wygodnej, wbudowanej metody zapytania `public_method_defined?`, która informuje, czy obiekt `Person` (reprezentowany w metodzie przez słowo kluczowe `self`) zawiera metodę o takiej samej nazwie jak nazwa znajdująca się na końcu łańcucha `all_with_`.

**Listing 4.13. Pełna implementacja metody Person.method_missing**

```
def self.method_missing(m, *args)
    method = m.to_s
    if method.start_with?("all_with_")                    ❶
      attr = method[9..-1]                                ❷
      if self.public_method_defined?(attr)                ❸
        PEOPLE.find_all do |person|
          person.send(attr).include?(args[0])             ❹
        end
      else
        raise ArgumentError, "Nie można znaleźć #{attr}"  ❺
      end
    else
      super                                               ❻
    end
  end
```

Jeśli istnieje komunikat o nazwie zawierającej łańcuch `all_with_` ❶, pożądane jest zignorowanie tej części nazwy i przechwycenie reszty łańcucha. W tym celu można pobrać podłańcuch położony od dziewiątego do ostatniego znaku. Właśnie to jest osiągane przez indeksowanie łańcucha za pomocą kodu `9..-1` ❷ (oznacza to rozpoczęcie od dziesiątego znaku, ponieważ indeksowanie łańcucha zaczyna się od zera). Trzeba teraz stwierdzić, czy wynikowy podłańcuch odpowiada jednej z metod instancji obiektu `Person`, a dokładniej rzecz biorąc, metodzie `hobbies` lub `friends`. Zamiast umieszczać w kodzie na stałe te dwie nazwy, zachowujemy elastyczność i skalowalność, sprawdzając, czy klasa `Person` definiuje metodę z tym podłańcuchem jako jej nazwą ❸.

To, co następuje dalej, zależy od tego, czy powiedzie się wyszukiwanie symbolu. Aby zacząć najpierw od drugiej gałęzi, gdy żądany atrybut nie istnieje, zgłaszamy błąd za pomocą odpowiedniego komunikatu ❺. Jeśli wyszukiwanie się powiedzie (ma to miejsce, gdy komunikat to `friends` lub `hobbies` albo dowolny inny później dodany atrybut), uzyskujemy to, do czego dążymy.

Oprócz nazwy metody `all_with_*` wywołanie metody obejmuje argument zawierający szukany element (na przykład imię przyjaciela lub zainteresowania). Argument ten znajduje się w `args[0]`, czyli w pierwszym elemencie tablicy pobierającej argumenty, który na liście argumentów identyfikowany jest przez `*args`. Ostatecznym efektem całej metody `method_missing` jest znalezienie wszystkich osób, dla których `attr` zawiera `args[0]` ❹. Taki kod pozwala na przykład określić wszystkie osoby, których zainteresowania obejmują muzykę, lub każdą osobę mającą wśród swoich przyjaciół konkretną osobę.

Zauważ, że ta wersja metody `method_missing` uwzględnia dwie struktury warunkowe. Wynika to stąd, że następujące dwa warunki mogą nie zostać spełnione: po pierwsze, możemy obsługiwać komunikat niezgodny z wzorcem `all_with_*` (na przykład `"blah"`); po drugie, może istnieć żądanie `all_with_*`, w którego nazwie znak * nie odpowiada niczemu, co jest znane klasie `Person` (na przykład `all_with_children`). Drugi warunek traktujemy jako warunek krytyczny i zgłaszamy błąd ❺. Jeśli pierwszy warunek nie zostanie spełniony, oznacza to, że konkretny komunikat nie jest tym, czego szuka określona metoda `method_missing`. Sterowanie jest przekazywane w górę do następnej, wyżej położonej definicji metody `method_missing` przez wywołanie słowa kluczowego `super` ❻.

To słowo kluczowe wywołane bez żadnych argumentów automatycznie pobiera wszystkie argumenty przekazane bieżącej metodzie. Oznacza to, że w tym przypadku takie wywołanie słowa kluczowego `super` odpowiada wywołaniu `super(m, *args)` (jest jednak krótsze i wygodniejsze).

> **UWAGA** W rozdziale 15. ponownie przyjrzymy się metodzie `method_missing` przy okazji szerszego omówienia mechanizmów i wywołań zwrotnych środowiska wykonawczego języka Ruby, wśród których metoda `method_missing` jest tylko jedną z nich (istnieje też metoda o nazwie `respond_to_missing?`, która — jak wskazuje jej nazwa — stanowi odmianę hybrydy; metoda ta zostanie zaprezentowana w rozdziale 15.). Warto jednak było zaprezentować w tym miejscu metodę `method_missing`, ponieważ prawdopodobnie jest ona najczęściej używanym członkiem rodziny wywołań zwrotnych, a także metodą, o której usłyszysz raczej wcześniej niż później w trakcie dalszych eksploracji języka Ruby.

Orientujesz się teraz dobrze zarówno w klasach i modułach, jak i w tym, w jaki sposób poszczególne obiekty po odebraniu komunikatów szukają pasującej metody, przechodząc swoje drzewo rodziny modułów/klas, a także jak obsługują wyszukiwanie zakończone niepowodzeniem. W dalszej kolejności przyjrzymy się temu, co jest możliwe do zrealizowania z wykorzystaniem takiego systemu, a dokładniej rzecz biorąc, zajmiemy się rodzajami decyzji, jakie można i należy podjąć odnośnie do projektu oraz nadawania nazw klasom i modułom. Dzięki temu możliwe będzie tworzenie przejrzystych i zrozumiałych programów.

## 4.4. Projekt i nadawanie nazw w przypadku klas i modułów

Fakt, że w języku Ruby występują klasy i moduły, a także to, że z punktu widzenia obiektu istotne jest tylko to, czy dana metoda istnieje, a nie to, w jakiej klasie lub module znajduje się definicja metody, oznacza, że masz wiele możliwości wyboru odnośnie do projektu i struktury tworzonego programu. Takie bogactwo opcji projektowych przywołuje na myśl kilka kwestii, których należy być świadomym.

Przyjrzeliśmy się już jednemu przypadkowi (klasa `Stack`), w którym możliwe było umieszczenie w jednej klasie wszystkich niezbędnych definicji metod. Korzystne było jednak wyodrębnienie części z nich i umiejscowienie ich w module (`Stacklike`),

a następnie połączenie modułu z klasą. Nie istnieje reguła określająca, kiedy należy wybrać poszczególne warianty działania. Zależy to od obecnych i przyszłych (w zakresie możliwym do przewidzenia) potrzeb. Czasami kuszące jest rozdzielenie wszystkiego w osobnych modułach, ponieważ moduły utworzone dla jednego programu mogą okazać się przydatne w innym („Po prostu wiem, że pewnego dnia ponownie będzie mi potrzebny ten moduł `ThreePronged`!" — podpowiada głos przewodnika w Twojej głowie). Istnieje jednak coś takiego jak przesadna modularyzacja. Zależy to od sytuacji. Masz do dyspozycji parę bogatych w możliwości narzędzi (dodawanie modułów do klas i dziedziczenie), dlatego w każdym przypadku musisz się zastanowić, jak je zrównoważyć.

### 4.4.1. Dodawanie modułów do klas i/lub dziedziczenie

Dodawanie modułów do klas jest blisko powiązane z dziedziczeniem klas. W obu przypadkach jeden element (klasa lub moduł) ustanawia bliskie powiązanie z drugim elementem. Elementy stają się sąsiadami w ścieżce wyszukiwania metod. W niektórych sytuacjach możesz stwierdzić możliwość zaprojektowania części programu z wykorzystaniem modułów lub dziedziczenia.

Klasa `CargoHold` jest przykładem. Zaimplementowano ją przez dodanie modułu `Stacklike`. Wybrano jednak wariant tworzenia klasy `Stack` zamiast modułu `Stacklike`, ale w dalszym ciągu możliwe było zastosowanie klasy `CargoHold`. Byłaby ona podklasą klasy `Stack`, co zilustrowano w listingu 4.14.

**Listing 4.14. Klasa CargoHold dziedzicząca z klasy Stack zamiast dodawania modułu Stacklike**

```
class Stack
  attr_reader :stack
  def initialize
    @stack = []
  end
  def add_to_stack(obj)
    @stack.push(obj)
  end
  def take_from_stack
    @stack.pop
  end
end
class Suitcase
end
class CargoHold < Stack
  def load_and_report(obj)
    print "Ładowanie obiektu "
    puts obj.object_id
    add_to_stack(obj)
  end
  def unload
    take_from_stack
  end
end
```

Z punktu widzenia pojedynczego obiektu `CargoHold` działanie kodu z powyższego listingu przebiega dokładnie tak jak w przypadku wcześniejszej implementacji, w której do klasy `CargoHold` dodano moduł `Stacklike`. Obiekt zajmuje się znajdowaniem i wykonywaniem metod odpowiadających odbieranym przez niego komunikatom. Obiekt znajduje takie metody w swojej ścieżce wyszukiwania metod albo ich nie znajduje. Obiektu nie interesuje to, czy metody zostały zdefiniowane w module, czy w klasie. Można to porównać do szukania śrubokrętu w domu: nie interesuje Cię to, w jakim pokoju go znajdziesz, a ponadto nie ma to żadnego znaczenia dla tego, co się stanie, gdy później użyjesz śrubokrętu do wykonania zadania.

Nie ma nic złego w wykorzystaniu takiego rozwiązania opartego na dziedziczeniu do implementowania klasy `CargoHold`, z tym wyjątkiem, że spowoduje to utratę możliwości dziedziczenia, jaką ta klasa oferuje. Jeśli inna klasa może być bardziej odpowiednia niż klasa `Stack` do pełnienia roli superklasy klasy `CargoHold` (hipotetycznie może to być klasa `StorageSpace` lub `AirplaneSection`), może okazać się niezbędna elastyczność, jaką by uzyskano przez przekształcenie w moduł przynajmniej jednej z tych klas.

Żadna pojedyncza reguła ani formuła nigdy nie zapewni właściwego projektu. Warto jednak pamiętać o następujących dwóch kwestiach podczas podejmowania decyzji odnośnie do użycia klas i modułów:

- *Moduły nie mają instancji*. Jest to zgodne z tym, że elementy są generalnie najlepiej modelowane w klasach, natomiast charakterystyki lub właściwości elementów są najlepiej hermetyzowane w modułach. Jak zaznaczono w punkcie 4.1.1, odpowiednio nazwy klasy będą zwykle rzeczownikami, natomiast nazwy modułów często są przymiotnikami (klasa `Stack` i moduł `Stacklike`).
- *Klasa może mieć tylko jedną superklasę, ale może dodać dowolną żądaną liczbę modułów*. Jeśli korzystasz z dziedziczenia, wyższy priorytet nadaj stworzeniu sensownej relacji superklasy i podklasy. Nie wykorzystuj w pełni relacji wyłącznie między klasą i superklasą, aby zapewnić klasie to, co może okazać się tylko jednym z kilku zestawów właściwości.

Podsumowano te reguły w jednym przykładzie, więc przyjrzyj się temu, czego *nie* należy robić:

```
module Vehicle
...
class SelfPropelling
...
class Truck < SelfPropelling
  include Vehicle
...
```

Zamiast tego kodu należy użyć następującego:

```
module SelfPropelling
...
class Vehicle
  include SelfPropelling
...
class Truck < Vehicle
...
```

W drugiej wersji kodu obiekty i właściwości są modelowane w znacznie bardziej staranny sposób. Klasa `Truck` pochodzi od klasy `Vehicle` (co ma sens), natomiast `Self ➥Propelling` to parametry pojazdów (przynajmniej wszystkich tych, które są uwzględnione w tym modelu „świata"). Parametry te są przekazywane ciężarówkom ze względu na bycie potomkiem przez klasę `Truck` lub specjalizowaną formą klasy `Vehicle`.

Inną ważną kwestią związaną z projektem klas/modułów jest wzajemne zagnieżdżanie modułów i/lub klas.

### 4.4.2. Zagnieżdżanie modułów i klas

Definicję klasy możesz zagnieździć w definicji modułu w następujący sposób:

```
module Tools
  class Hammer
  end
end
```

Aby utworzyć instancję klasy `Hammer` zdefiniowanej wewnątrz modułu `Tools`, należy użyć tokenu wyszukiwania stałych w postaci podwójnego dwukropka (`::`) w celu wskazania ścieżki do nazwy klasy:

```
h = Tools::Hammer.new
```

Zagnieżdżone łańcuchy modułów/klas, takie jak `Tools::Hammer`, są czasem wykorzystywane do tworzenia osobnych przestrzeni nazw dla klas, modułów i metod. Technika ta może być pomocna, gdy dwie klasy mają podobną nazwę, ale nie są identyczną klasą. Jeśli na przykład istnieje klasa `Tools::Hammer`, możesz też użyć klasy `Piano::Hammer`. Między tymi dwoma klasami `Hammer` nie będzie występować konflikt, ponieważ każda z nich jest zagnieżdżona we własnej przestrzeni nazw (w pierwszym przypadku jest to przestrzeń `Tools`, a w drugim przestrzeń `Piano`).

Alternatywnym sposobem osiągnięcia takiego rozdzielenia byłoby zastosowanie klas `ToolsHammer` i `PianoHammer` bez martwienia się o zagnieżdżanie ich w modułach. Jednak łączenie ze sobą łańcuchów nazw w taki sposób może szybko doprowadzić do wizualnego nieporządku, zwłaszcza wtedy, gdy elementy są zagnieżdżane głębiej niż na dwóch poziomach.

**Klasa czy moduł?**

Gdy napotkasz konstrukcję taką jak `Tools::Hammer`, nie możesz wyłącznie na jej podstawie stwierdzić, co jest klasą, a co modułem, ani też czy `Hammer` to zwykła, poczciwa stała (`Tools` musi być klasą lub modułem, ponieważ zawiera zagnieżdżony element `Hammer`). W wielu sytuacjach nie ma znaczenia to, że w tego rodzaju kontekście nie możesz odróżnić klas od modułów. Istotne jest zagnieżdżanie lub tworzenie łańcucha nazw w sensowny sposób. Brak możliwości określenia, co jest czym, bez przyjrzenia się kodowi źródłowemu lub sprawdzenia dokumentacji nie jest taki zły. Jest to konsekwencją tego, że klasy są modułami (klasa `Class` jest podklasą klasy `Module`), dlatego pod wieloma względami ich działanie (z najbardziej widocznym wyjątkiem, że mogą być tworzone instancje klas) jest podobne. Oczywiście zwykle będziesz wiedzieć, co reprezentuje `Tools::Hammer`, ponieważ sam utworzyłeś kod lub zaznajomiłeś się z dokumentacją. Warto jednak mieć świadomość tego, że sam zapis nie przekazuje wszystkich informacji.

W następnym rozdziale przyjrzymy się dokładniej zagnieżdżonym klasom, modułom i innym stałym przy okazji szczegółowego omawiania zagadnienia zasięgu. Na razie zwróć uwagę na to, że możliwość wzajemnego zagnieżdżania modułów i klas (o dowolnej głębokości i w dowolnej kolejności) zapewnia jeszcze jedną „oś", wzdłuż której możesz planować projekt i strukturę programu.

## 4.5. Podsumowanie

Rozdział 4. stanowił zarówno uzupełnienie, jak i kontynuację poprzedniego rozdziału dotyczącego klas. W rozdziale zaprezentowano następujące rzeczy:

- Dokładna prezentacja modułów.
- Podobieństwa i różnice między modułami i klasami (choć oba elementy mogą przechowywać metody i stałe, moduły nie umożliwiają tworzenia instancji).
- Przykłady możliwego wykorzystania modułów do opisania projektu programu.
- Postrzeganie przez obiekt procesu znajdowania i wykonywania metody w odpowiedzi na komunikat lub obsługi niepowodzenia za pomocą metody `method_missing` w sytuacjach, gdy dla komunikatu nie ma pasującej metody.
- Wzajemne zagnieżdżanie klas i modułów zapewniające korzyść w postaci utrzymywania osobno przestrzeni nazw w przejrzysty sposób.

Szczególnie ważne jest uwzględnienie sposobu określania przez obiekty metod dla komunikatów: obiekty przeszukują serię klas i modułów. Same obiekty nie mają metod, nawet pomimo tego, że czasami wygodnym skrótem myślowym jest stwierdzenie, że obiekt zawiera metodę. Metody znajdują się w klasach i modułach. Obiekty mają możliwość przechodzenia hierarchii klas i modułów w poszukiwaniu metod.

Gdy już wzajemnie zagnieżdżamy elementy, pojęcie zasięgu jest następnym tematem, jaki należy omówić szczegółowo. Co się dzieje z danymi i zmiennymi, gdy program przełącza się między kontekstami kodu? Przyjrzymy się zasięgowi, prezentując jednocześnie zagadnienie często powiązane z zasięgiem, czyli obiekt domyślny `self`.

# 5 Obiekt domyślny (self), zasięg i widoczność

**W tym rozdziale:**

- Rola obiektu bieżącego lub domyślnego `self`.
- Reguły określania zasięgu zmiennych lokalnych, zmiennych globalnych i zmiennych klasy.
- Wyszukiwanie stałych i widoczność.
- Reguły uzyskiwania dostępu do metod.

Przy opisywaniu i omawianiu programów komputerowych często używamy metafor przestrzennych, a czasem odnoszących się do ludzi. Mówimy o byciu „w" definicji klasy lub zwracaniu „z" wywołania metody. Zwracamy się do obiektów w drugiej osobie, tak jak w przypadku wywołania `obj.respond_to?("x")` (co oznacza: „Hej, `obj`, czy odpowiadasz na `x`?"). W trakcie działania programu to, jakie obiekty są adresowane, a także gdzie znajdują się one w wyimaginowanej przestrzeni programu, cały czas się zmienia.

Zmiany te nie są wyłącznie metaforyczne. Zmienia się również znaczenie identyfikatorów. Kilka elementów ma wszędzie to samo znaczenie. Na przykład liczby całkowite oznaczają to, co oznaczają wszędzie tam, gdzie występują. To samo dotyczy słów kluczowych: nie możesz użyć słów kluczowych takich jak `def` i `class` jako nazw zmiennych. Z tego powodu po napotkaniu ich z łatwością możesz ustalić ich przeznaczenie. Znaczenie większości elementów zależy jednak od ich kontekstu.

Większość słów i tokenów (większość identyfikatorów) może oznaczać co innego w różnych miejscach i momentach.

Niniejszy rozdział ma na celu zapewnienie umiejętności orientacji w kodzie Ruby. Oznacza to możliwość stwierdzenia, jak zostaną określone używane identyfikatory, śledzenie zmian w kontekście oraz ustalanie sensu używania i ponownego wykorzystywania identyfikatorów i terminów. Jeśli rozumiesz, co może się zmienić między dwoma kontekstami, a także co powoduje zmianę w kontekście (na przykład aktywowanie bloku definicji metody), zawsze będziesz w stanie zorientować się w programie Ruby. I nie jest to jedynie kwestia umiejętności biernego czytania kodu Ruby: musisz też wiedzieć o istnieniu kontekstów, jak również orientować się, jaki mają one wpływ na znaczenie tego, co robisz, pisząc kod Ruby.

Na początku rozdziału skoncentrowano się na dwóch zagadnieniach: *obiekt* `self` i *zasięg*. `Self` to obiekt „bieżący" lub „domyślny". Taka rola jest zwykle przypisywana wielu kolejnym (choć tylko jednemu na raz) obiektom w czasie działania programu. Obiekt `self` w języku Ruby można porównać z pierwszą osobą lub „ja" programu. Jak w przypadku książki z wieloma narratorami w pierwszej osobie rola „ja" może być przekazywana. Zawsze występuje jedna rola „ja", ale zmienia się obiekt, który ją przyjmuje. Reguły zasięgu nadzorują widoczność zmiennych (oraz innych elementów, ale głównie zmiennych). Ważne jest, aby wiedzieć, jaki zasięg jest bieżącym. W ten sposób można stwierdzić, jakie zmienne odwołują się do niego, a ponadto nie mylić ich ze zmiennymi z innych zasięgów o takiej samej nazwie lub z metodami o podobnej nazwie.

Między obiektem `self` i zasięgiem znajdują się klucze główne służące do orientacji w programie Ruby. Jeśli wiesz, jaki zasięg jest bieżącym, a ponadto jaki obiekt jest obiektem `self`, będziesz w stanie stwierdzić, co ma miejsce, jak również szybko analizować błędy.

Trzecie podstawowe zagadnienie omawiane w rozdziale to *dostęp do metody*. Język Ruby zapewnia mechanizmy rozróżniania poziomów dostępu do metod. Zasadniczo oznacza to reguły ograniczające wywoływanie metod zależnie od tego, czym jest obiekt `self`. Dostęp do metody jest zatem metapojęciem bazującym na omówieniu pojęcia obiektu `self` i zasięgu.

Na końcu zajmiemy się też tematem, który łączy w sobie niektóre z wcześniej wymienionych zagadnień. Mowa o *metodach najwyższego poziomu*, które są tworzone poza dowolną definicją klasy lub modułu.

Zacznijmy od obiektu `self`.

## 5.1. Obiekt self, czyli obiekt bieżący/domyślny

Jednym z kamieni milowych programowania w języku Ruby (pod pewnymi względami jest to szkielet) jest obiekt domyślny lub obiekt bieżący dostępny w programie za pośrednictwem słowa kluczowego `self`. W każdym momencie działania programu występuje jeden i tylko jeden obiekt `self`. Jak się okaże, bycie takim obiektem daje określone przywileje. W tym podrozdziale przyjrzymy się temu, jak interpreter języka Ruby określa, jaki obiekt w danym momencie jest obiektem `self`, a także jakie przywileje są nadawane takiemu obiektowi.

### 5.1.1. Co zostaje obiektem self i w jakim miejscu?

Zawsze istnieje jeden (i tylko jeden) obiekt bieżący lub obiekt `self`. Aby stwierdzić, który to jest obiekt, należy postępować zgodnie z niewielkim zestawem reguł podsumowanych w tabeli 5.1. Zawartość tabeli będzie sukcesywnie objaśniana i ilustrowana.

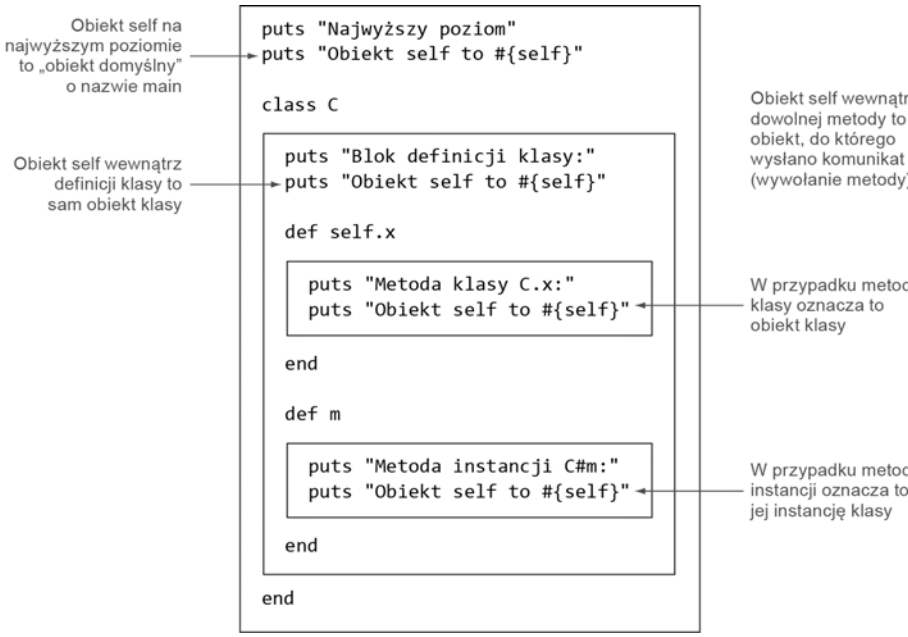
**Tabela 5.1.** Sposób określania obiektu bieżącego (self)

| Kontekst | Przykład | Jaki obiekt jest obiektem `self`? |
|---|---|---|
| Najwyższy poziom programu | Dowolny kod poza obrębem innych bloków. | `main` (wbudowany obiekt domyślny najwyższego poziomu) |
| Definicja klasy | class C<br>self | Obiekt klasy `C` |
| Definicja modułu | module M<br>self | Obiekt modułu `M` |
| Definicje metody | 1. Najwyższy poziom (poza obrębem bloku definicji):<br>`def nazwa_metody`<br>`self` | W momencie wywołania metody dowolny obiekt jest obiektem `self`. Metody najwyższego poziomu są dostępne jako metody prywatne dla wszystkich obiektów. |
| | 2. Definicja metody instancji w klasie:<br>`class C`<br>`def nazwa_metody`<br>`self` | Instancja klasy `C`, która odpowiada metodzie `nazwa_metody`. |
| | 3. Definicja metody instancji w module:<br>`module M`<br>`def nazwa_metody`<br>`self` | – Pojedynczy obiekt rozwijany przez moduł `M`.<br>– Instancja klasy dodająca moduł `M`. |
| | 4. Metoda pojedynczego obiektu w konkretnym obiekcie:<br>`def obj.nazwa_metody`<br>`self` | `Obj` |

Aby dowiedzieć się, jaki obiekt jest obiektem `self`, musisz zidentyfikować kontekst bieżący. W praktyce pod uwagę należy brać wiele kontekstów. Dostępny jest najwyższy poziom (przed uaktywnieniem dowolnego innego kontekstu, takiego jak definicja klasy). Istnieją bloki definicji klasy, bloki definicji modułu oraz bloki definicji metody. W zasadzie to tyle, pomijając kilka subtelności związanych ze sposobem interakcji tych kontekstów. Jak pokazano w tabeli 5.1, obiekt `self` jest ustalany na podstawie tego, jaki z tych kontekstów jest aktualnie używany (definicje klasy i modułu są podobne oraz blisko powiązane).

Na rysunku 5.1 zaprezentowano diagram z większością przypadków z tabeli 5.1. Rysunek i tabela pokazują, że jakiś obiekt zawsze jest obiektem `self`. To, który jest to obiekt, zależy od bieżącego położenia w programie.

Najwyższy poziom to najbardziej podstawowy kontekst programu, który pod pewnymi względami jest unikatowy. Jest to kontekst programu istniejący przed otwarciem jakiejkolwiek definicji klasy lub modułu lub po zamknięciu ich wszystkich. Przyjrzymy się teraz związanym z najwyższym poziomem pomysłom odnoszącym się do obiektu `self`.




**Rysunek 5.1.** Określanie obiektu self w różnych kontekstach

### *5.1.2. Obiekt self najwyższego poziomu*

Termin *najwyższy poziom* odnosi się do kodu programu utworzonego poza obrębem dowolnego bloku definicji klasy lub modułu. Jeśli otworzysz zupełnie nowy plik tekstowy i wpiszesz:

```
x = 1
```

spowoduje to utworzenie zmiennej lokalnej `x` najwyższego poziomu. Jeśli wpiszesz:

```
def m
end
```

nastąpi utworzenie metody najwyższego poziomu (znacznie bardziej szczegółowo przyjrzymy się takim metodom w podrozdziale 5.4; w tym miejscu są one odpowiednie jedynie jako wskaźniki istnienia obiektu `self` najwyższego poziomu). W kilku przykładach, zwłaszcza w tych zamieszczonych we wcześniejszych rozdziałach (na przykład znajdują się one w rozdziale 2., gdzie zademonstrowano semantykę argumentów), uwzględniono kod najwyższego poziomu. Po rozpoczęciu tworzenia definicji klas i modułów w ich obrębie zaczęło się pojawiać więcej przykładowego kodu. Obiekt `self` zmienia się w definicjach klas, modułów i metod w jednolity sposób: słowo kluczowe (`class`, `module` lub `def`) oznacza przełączenie na nowy obiekt `self`. Jaki jednak będzie obiekt `self`, gdy nie aktywowano jeszcze żadnego bloku definicji?

Odpowiedź na to pytanie jest taka, że w języku Ruby zapewniany jest początkowy obiekt `self` na najwyższym poziomie. Jeśli zażądasz od interpretera języka zidentyfikowania się za pomocą następującego polecenia:

```
ruby -e 'puts self'
```

zostanie podane, że obiekt ten ma nazwę `main`.

`main` to specjalny termin używany przez obiekt domyślny `self` do odwołania się do samego siebie. Nie możesz odwołać się do tego obiektu jako `main`. Takie użycie `main` zostanie zinterpretowane przez język Ruby jako zastosowanie nazwy zwykłej zmiennej lub metody. Aby z jakiegoś powodu uzyskać dostęp do `main`, konieczne jest przypisanie tej nazwy do zmiennej na najwyższym poziomie:

```
m = self
```

Choć mało prawdopodobne jest to, że niezbędne będzie wykonanie takiego działania, właśnie w taki sposób jest ono realizowane. Częściej odczujesz potrzebę dość dokładnego określenia znaczenia tego, co jest obiektem `self` w używanych definicjach klas, modułów i metod, w których realizowana jest większość działań programistycznych.

### 5.1.3. Obiekt self w definicjach klas, modułów i metod

Warto zwrócić szczególną uwagę na obiekt `self` podczas tworzenia klas, modułów i metod. Nie ma wielu reguł do opanowania, które są konsekwentnie stosowane. Są one jednak warte poznania na początku, aby dobrze zrozumieć, dlaczego taką, a nie inną rolę odgrywają różne używane techniki, które zależą od wartości obiektu `self`.

Wszystko sprowadza się do zmiany obiektu `self` między kolejnymi obiektami, co ma miejsce w momencie aktywowania definicji klasy lub modułu, definicji metody instancji lub definicji metody pojedynczego obiektu (w tym metody klasy).

#### OBIEKT SELF W DEFINICJACH KLAS I MODUŁÓW

W definicji klasy lub modułu obiekt `self` jest obiektem klasy lub modułu. Ta niepozornie brzmiąca reguła jest ważna. Jeśli ją opanujesz, uchronisz się przed kilkoma błędami często popełnianymi przez osoby, które uczą się języka Ruby.

Za pomocą jawnie użytej metody `puts` możliwe jest stwierdzenie, czym jest obiekt `self` na różnych poziomach definicji klasy i/lub modułu. Prezentuje to listing 5.1.

**Listing 5.1. Sprawdzanie obiektu self za pośrednictwem wywołań metody puts w definicjach class i module**

```
class C
  puts "Właśnie aktywowano klasę C:"
  puts self                    ←——— Dane wyjściowe: C
  module M
    puts "Zagnieżdżony moduł C::M:"
    puts self                  ←——— Dane wyjściowe: C::M
  end
  puts "Powrót do zewnętrznego poziomu klasy C:"
  puts self                                ←——— Dane wyjściowe: C
end
```

Od razu po napotkaniu słowa kluczowego `class` lub `module` obiektem `self` staje się klasa lub moduł, których blok definicji (obiekt `Class` lub `Module`) aktywowano. Listing 5.1 prezentuje następujące dwa przypadki: aktywowanie klasy `C`, a następnie modułu `C::M`. Po opuszczeniu tego modułu, lecz jeszcze nie klasy `C`, obiektem `self` ponownie staje się ta klasa.

Oczywiście bloki definicji klasy i modułu realizują więcej działań niż tylko rozpoczynanie i kończenie. Zawierają one również definicje metod, które zarówno w przypadku metod instancji, jak i metod klasy mają własne reguły określania obiektu `self`.

### OBIEKT SELF W DEFINICJACH METOD INSTANCJI

W przypadku definicji metod instancji pojęcie obiektu `self` jest subtelne. Wynika to z następującego powodu: gdy interpreter napotka blok `def/end`, natychmiast definiuje metodę. Jednak kod wewnątrz definicji metody nie jest wykonywany do momentu, gdy obiekt mający możliwość wyzwolenia jej wykonania odbierze odpowiedni komunikat.

Przyglądając się definicji metody na papierze lub ekranie, w zasadzie możesz stwierdzić tylko tyle, że po wywołaniu metody obiektem `self` będzie obiekt, który ją wywołał (odbiorca komunikatu). W momencie tworzenia definicji metody możesz jedynie stwierdzić, że obiekt `self` wewnątrz tej metody będzie jakimś przyszłym obiektem, który wywoła metodę.

Możesz tak zmodyfikować metodę, aby podczas działania poinformowała o obiekcie `self`:

```
class C
  def x
    puts "Klasa C, metoda x:"
    puts self
  end
end
c = C.new
c.x
puts "Wystąpiło wywołanie metody x przez: #{c}"
```

Powyższy fragment kodu zwraca następujące dane wyjściowe:

```
Klasa C, metoda x:
#<C:0x00000101b381a0>
Wystąpiło wywołanie metody x przez: #<C:0x00000101b381a0>
```

Dziwnie wyglądający element w danych wyjściowych (`#<C:0x00000101b381a0>`) to sposób przekazania przez interpreter języka Ruby informacji: „instancja klasy `C`" (liczba szesnastkowa po dwukropku to odwołanie do lokalizacji w pamięci; po uruchomieniu kodu w systemie prawdopodobnie uzyskasz inną liczbę). Jak widać, odbiorca komunikatu `x`, a mianowicie `c`, przyjmuje rolę obiektu `self` podczas wykonywania metody `x`.

### OBIEKT SELF W DEFINICJACH METOD POJEDYNCZEGO OBIEKTU I METOD KLASY

Jak możesz oczekiwać, podczas wykonywania metody pojedynczego obiektu obiektem `self` jest obiekt będący właścicielem metody. Obiekt chętnie o tym poinformuje:

```
obj = Object.new
def obj.show_me
  puts "Wewnątrz metody pojedynczego obiektu show_me obiektu #{self}"
end
obj.show_me
puts "Powrót z wywołania metody show_me przez #{obj}"
```

Dane wyjściowe dla tego przykładowego kodu są następujące:

```
Wewnątrz metody pojedynczego obiektu show_me obiektu #<Object:0x00000101b19840>
Powrót z wywołania metody show_me przez #<Object:0x00000101b19840>
```

Sensowne jest to, że jeśli metodę utworzono tak, aby została wywołana przez tylko jeden obiekt, staje się on obiektem `self`. Co więcej, jest to dobry moment, aby utrwalić metody klasy, które w zasadzie są metodami pojedynczego obiektu powiązanymi z obiektami klasy. W następującym przykładowym kodzie zwracane są informacje dotyczące obiektu `self` z wnętrza metody klasy `C`:

```
class C
  def C.x
    puts "Metoda klasy C"
    puts "Obiekt self: #{self}"
  end
end
C.x
```

Oto zwracane informacje:

```
Metoda klasy C
Obiekt self: C
```

Z całą pewnością obiektem `self` wewnątrz metody pojedynczego obiektu (w tym przypadku metody klasy) jest obiekt tej metody.

**Użycie obiektu self zamiast nazw klas umieszczanych na stałe w kodzie**

W ramach drobnej porady programistycznej przedstawiamy wariant związany z ostatnim przykładem:

```
class C
  def self.x          ←— ❶
    puts "Metoda klasy C"
    puts "Obiekt self: #{self}"
  end
end
```

Zwróć uwagę na użycie wywołania `self.x` ❶ zamiast `C.x`. W takim sposobie zapisania metody klasy wykorzystano to, że w definicji klasy obiektem `self` jest `C`. A zatem `def self.x` jest tożsame z `def C.x`.

Wersja `self.x` oferuje drobną korzyść: jeśli kiedykolwiek zdecydujesz się na zmianę nazwy klasy, `self.x` automatycznie dopasuje się do nowej nazwy. W przypadku umieszczenia `C.x` na stałe w kodzie konieczna będzie zmiana nazwy klasy `C` na jej nową nazwę. Trzeba jednak zachować ostrożność. Pamiętaj o tym, że obiekt `self` wewnątrz metody zawsze jest obiektem, w którym metoda została wywołana. Może wystąpić sytuacja, w której może się wydawać, że obiektem `self` powinien być jeden obiekt klasy, lecz w rzeczywistości jest nim inny obiekt:

```
class D < C
end
D.x
```

Klasa `D` wywołuje metodę `x`, ponieważ podklasy wywołują metody klasy ich superklas. Jak się okaże, w przypadku uruchomienia kodu metoda `C.x` poprawnie informuje o byciu obiektem `self` przez obiekt `D`, ponieważ w tym obiekcie wywoływana jest metoda.

Bycie obiektem `self` w danym miejscu programu wiąże się z pewnymi przywilejami. Głównym przywilejem takiego obiektu jest, jak się okaże dalej, pełnienie roli domyślnego odbiorcy komunikatów.

### 5.1.4. Obiekt self jako domyślny odbiorca komunikatów

Wywoływanie metod (czyli wysyłanie komunikatów do obiektów) wiąże się zwykle z użyciem zapisu z kropką:

```
obj.talk
ticket.venue
"abc".capitalize
```

Jest to standardowa, pełna postać składni wywoływania metod w języku Ruby. Specjalna reguła zarządza jednak wywołaniami metod: jeśli odbiorca komunikatu jest obiektem `self`, możesz pominąć takiego odbiorcę i znak kropki. Interpreter języka Ruby użyje obiektu `self` jako odbiorcy domyślnego. Oznacza to, że wysyłany komunikat zostanie skierowany do obiektu `self`. Prezentują to następujące odpowiedniki:

```
talk       <------ To samo co self.talk
venue      <------ To samo co self.venue
capitalize <------ To samo co self.capitalize
```

> **OSTRZEŻENIE** Choć możesz nadać metodzie i zmiennej lokalnej taką samą nazwę, rzadko (jeśli w ogóle) jest to dobry pomysł. Jeżeli istnieją metoda i zmienna o danej nazwie, a ponadto używasz samego identyfikatora (np. `talk`), pierwszeństwo będzie mieć zmienna. Aby wymusić rozpoznawanie identyfikatora przez interpreter języka Ruby jako nazwę metody, konieczne byłoby zastosowanie kodu `self.talk` lub wywołanie metody z pustą listą argumentów: `talk()`. Ponieważ zmienne nie pobierają argumentów, nawiasy oznaczają, że masz na myśli metodę, a nie zmienną. I tym razem, jeśli to możliwe, najlepiej unikać tego rodzaju kolizji nazw.

Sprawdźmy w praktyce to pojęcie, tworząc sytuację, w której wiemy, co jest obiektem `self`, a następnie przetestujmy formę wywoływania metody bez używania znaku kropki. Na najwyższym poziomie bloku definicji klasy obiekt `self` jest obiektem klasy. Wiemy, w jaki sposób dodawać metody bezpośrednio do obiektów klasy. A zatem dysponujemy składnikami pozwalającymi na przeprowadzenie demonstracji odbiorcy domyślnego:

```
class C
  def C.no_dot
    puts "Dopóki obiektem self jest obiekt C, możesz wywołać tę metodę bez znaku kropki"
  end
  no_dot     <------ ❶
end
C.no_dot     <------ ❷
```

Pierwsze wywołanie metody `no_dot` ❶ nie zawiera jawnego odbiorcy. Wywołanie ma uproszczoną postać. Gdy interpreter języka Ruby napotka taki kod (i określi, że jest to wywołanie metody, a nie zmienna lub słowo kluczowe), rozpozna go jako skrót następującego wywołania:

```
self.no_dot
```

W efekcie zostanie wyświetlony komunikat. W przypadku omawianego przykładu `self.no_dot` byłoby tożsame z `C.no_dot`, ponieważ uaktywniono blok definicji klasy `C`, a zatem obiektem `self` jest obiekt `C`. W rezultacie wywoływana jest metoda `C.no_dot` i zwracane są dane wyjściowe.

Przy drugim wywołaniu metody ❷ ponownie przetwarzanie odbywa się poza blokiem definicji klasy, dlatego `C` nie jest już obiektem `self`. Oznacza to, że w celu wywołania metody `no_dot` niezbędne jest określenie odbiorcy: `C`.

Wynikiem jest drugie wywołanie metody `no_dot` (lecz z użyciem kropki) i ponowne wyświetlenie jej danych wyjściowych.

Najczęstszy wariant użycia wywołania metody bez znaku kropki ma miejsce w przypadku wywoływania jednej metody instancji z poziomu innej. Oto przykład:

```
class C
  def x
    puts "To jest metoda x"
  end
  def y
    puts "To jest metoda y, która wywoła metodę x bez znaku kropki."
    x
  end
end
c = C.new
c.y
```

Oto dane wyjściowe:

```
To jest metoda y, która wywoła metodę x bez znaku kropki.
To jest metoda x.
```

W momencie wywołania metody `c.y` wykonywana jest metoda `y` z obiektem `self` ustawionym na obiekt `c` (jest on instancją klasy `C`). Wewnątrz metody `y` uproszczone odwołanie do metody `x` jest interpretowane jako komunikat, który ma zostać wysłany do obiektu `self`. Z kolei to oznacza, że wykonywana jest metoda `x`.

Istnieje sytuacja, w której nie możesz pominąć w wywołaniu metody części złożonej z nazwy obiektu i kropki, a mianowicie, gdy nazwa metody zakończona jest znakiem równości. Inaczej mówiąc, jest to metoda *ustawiająca*. Aby wywołać metodę `venue=` w obiekcie `self`, zamiast `venue = "Sala Miejska"` musisz użyć `self.venue = "Sala Miejska"`. Wynika to stąd, że w języku Ruby sekwencja *identyfikator* = *wartość* zawsze jest interpretowana jako przypisanie do zmiennej lokalnej. W celu wywołania metody `venue=` w bieżącym obiekcie niezbędne jest jawne dołączenie obiektu `self`. W przeciwnym razie uzyska się zmienną o nazwie `venue`, a ponadto nie wystąpi wywołanie metody ustawiającej.

Przypisanie obiektowi `self` roli odbiorcy domyślnego w przypadku wywołań metod bez znaku kropki umożliwia ulepszenie kodu w sytuacjach, w których jedna metoda

korzysta z innej metody. Typowym przypadkiem jest tworzenie całej nazwy za pomocą jej składników: pierwszego, opcjonalnego środkowego i ostatniego. Listing 5.2 prezentuje technikę, która to umożliwia z wykorzystaniem atrybutów odpowiadających trzem wartościom nazwy, oraz logiki warunkowej dołączającej drugie imię i spację końcową (wyłącznie wtedy, gdy występuje drugie imię).

**Listing 5.2. Tworzenie całej nazwy na podstawie wartości z wykorzystaniem wywołań metody w niejawnym obiekcie self**

```
class Person
  attr_accessor :first_name, :middle_name, :last_name
  def whole_name
    n = first_name + " "
    n << "#{middle_name} " if middle_name
    n << last_name                          ←❶
  end
end
david = Person.new
david.first_name = "Dawid"
david.last_name = "Nowak"
puts "Personalia Dawida: #{david.whole_name}"
david.middle_name = "Adam"
puts "Nowe personalia Dawida: #{david.whole_name}"
```

Dane wyjściowe zwrócone przez kod wywołujący z powyższego listingu mają następującą postać:

```
Personalia Dawida: Dawid Nowak
Nowe personalia Dawida: Dawid Adam Nowak
```

Definicja metody `whole_name` zależy od uproszczonych wywołań metod `first_name`, `middle_name` i `last_name`, które są wysyłane do obiektu `self`, czyli instancji klasy `Person` (w przykładzie `david`). Zmienna `n` pełni rolę akumulatora łańcuchów, do którego kolejno po jednym dodawane są składniki personaliów. Wartością zwracaną całej metody jest zmienna `n`, ponieważ wyrażenie `n << last_name` ❶ powoduje dołączanie metody `last_name` do `n` i zwracanie wyniku tej operacji.

Oprócz automatycznego pełnienia roli odbiorcy w przypadku komunikatów w uproszczonej postaci obiekt `self` korzysta z tego, że jest właścicielem zmiennych instancji.

### *5.1.5. Określanie zmiennych instancji za pośrednictwem obiektu self*

Następująca prosta reguła zarządza zmiennymi instancji i ich określaniem: każda zmienna instancji, z jaką kiedykolwiek spotkasz się w programie Ruby, należy do dowolnego obiektu, który jest bieżącym obiektem (`self`) w danym miejscu programu.

Poniżej przedstawiono klasyczny przypadek, w którym taka informacja okazuje się przydatna. Przekonaj się, czy możesz stwierdzić, co wyświetli następujący kod, przed jego uruchomieniem:

```
class C
  def show_var
    @v = "Jestem zmienną instancji inicjalizowaną w postaci łańcucha."  ←❶
    puts @v
  end
```

```
  @v = "Zmienne instancji mogą pojawić się wszędzie...." ←——❷
end
C.new.show_var
```

Kod wyświetla następujący komunikat:

```
Jestem zmienną instancji inicjalizowaną w postaci łańcucha.
```

Występuje tutaj pułapka polegająca na tym, że możesz pomyśleć, iż zostanie wyświetlony komunikat `Zmienne instancji mogą pojawić się wszędzie.....` Kod nie wyświetla tego komunikatu, ponieważ zmienna `@v` w definicji metody ❶ oraz zmienna `@v` poza jej obrębem ❷ są całkowicie niepowiązane ze sobą. Obie zmienne są zmiennymi instancji o nazwie `@v`, ale nie są tą samą zmienną. Zmienne te należą do różnych obiektów.

Do czego należą te zmienne?

Pierwsza zmienna `@v` ❶ znajduje się wewnątrz bloku definicji metody instancji klasy `C`. Ma to konsekwencje dotyczące nie pojedynczego obiektu, lecz ogólnie instancji klasy `C`: każda taka instancja, która wywołuje tę metodę, będzie mieć własną zmienną instancji `@v`.

Druga zmienna `@v` ❷ należy do obiektu klasy `c`. Jest to jedna z wielu sytuacji, w których warto pamiętać o tym, że klasy są obiektami. Dowolny obiekt może mieć własne zmienne instancji, czyli prywatny schowek z informacjami i stanem obiektu. Obiekty klasy korzystają z tego przywileju w takim stopniu jak każdy inny obiekt.

I tym razem logika niezbędna do stwierdzenia, jaki obiekt jest w posiadaniu danej zmiennej instancji, jest prosta i spójna: każda zmienna instancji należy do dowolnego obiektu pełniącego rolę obiektu `self` w momencie wykonywania kodu, który zawiera zmienną instancji.

Dokonajmy szybkiej przebudowy przykładu, zapewniając tym razem trochę więcej informacji o tym, co się dzieje. Listing 5.3 prezentuje zmodyfikowany kod.

**Listing 5.3. Demonstracja relacji między zmiennymi instancji i obiektem self**

```
class C
  puts "Właśnie aktywowano blok definicji klasy. Oto obiekt self:"
  p self
  @v = "Jestem zmienną instancji na najwyższym poziomie zawartości klasy."
  puts "Oto zmienna instancji @v, która należy do obiektu #{self}:"
  p @v
  def show_var
    puts "Właśnie aktywowano blok definicji metody instancji. Oto obiekt self:"
    p self
    puts "Oto zmienna instancji @v, która należy do obiektu #{self}:"
    p @v
  end
end
c = C.new
c.show_var
```

Dane wyjściowe z tej wersji kodu są następujące:

```
Właśnie aktywowano blok definicji klasy. Oto obiekt self:
C
Oto zmienna instancji @v, która należy do obiektu C:
"Jestem zmienną instancji na najwyższym poziomie zawartości klasy."
```

```
Właśnie aktywowano blok definicji metody instancji. Oto obiekt self:
#<C:0x00000101a77338>
Oto zmienna instancji @v, która należy do obiektu #<C:0x00000101a77338>:
Nil
```

Z całą pewnością każdy z tych dwóch różnych obiektów (obiekt klasy `C` oraz instancja klasy `C` o nazwie `c`) ma własną zmienną instancji `@v`. To, że zmienna `@v` instancji ma wartość `nil`, pokazuje, że przypisanie do zmiennej `@v` klasy nie miało nic wspólnego ze zmienną `@v` instancji.

Zrozumienie obiektu `self`, zarówno podstawowego faktu, że jego rola jest pełniona przez jakiś obiekt w każdym miejscu programu, jak i sposobu identyfikowania obiektu, który jest obiektem `self`, to jeden z najważniejszych aspektów procesu poznawania języka Ruby. Innym równie istotnym aspektem jest poznanie zasięgu, którym zajmiemy się w dalszej kolejności.

## 5.2. Określanie zasięgu

*Zasięg* odnosi się do widoczności identyfikatorów, a dokładniej rzecz biorąc — zmiennych i stałych. Odmienne reguły określania zasięgu powiązane są z różnymi typami identyfikatorów. Na przykład użycie identyfikatora `x` dla zmiennej lokalnej w każdej z dwóch definicji metody powoduje inny efekt niż zastosowanie zmiennej globalnej `$x` w tych samych dwóch miejscach. Wynika to stąd, że zmienne lokalne i globalne różnią się zasięgiem. W tym podrozdziale omówimy trzy typy zmiennych: zmienne lokalne, zmienne globalne i zmienne klasy (jak już pokazano, zmienne instancji są powiązane bardziej z obiektem `self` niż z zasięgiem). Przyjrzymy się również regułom określania stałych.

Obiekt `self` i zasięg są podobne w tym, że zmieniają się w trakcie działania programu. Ponadto w ich przypadku możesz określić, co się z nimi dzieje, analizując program oraz uruchamiając go. Obiekt `self` i zasięg nie są jednak tym samym. Możliwe jest zainicjowanie nowego zasięgu lokalnego bez zmieniania obiektu `self`, ale czasami obiekt `self` i zasięg zmieniają się razem. W odniesieniu do nich wspólne jest to, że są wymagane do nadania sensu temu, co kod zamierza wykonać. Podobnie jak stwierdzenie tego, jaki obiekt jest obiektem `self`, tak i określenie bieżącego zasięgu pozwala ustalić znaczenie kodu.

Zacznijmy od zmiennych globalnych, które nie są najczęściej używaną konstrukcją, ale są niezbędne do opanowania.

### 5.2.1. Zasięg globalny i zmienne globalne

*Zasięg globalny* to zasięg obejmujący cały program. Korzystają z niego zmienne globalne, które mogą być rozpoznane dzięki znakowi `$` umieszczonemu na początku nazwy. Tego rodzaju zmienne są dostępne wszędzie. „Pokonują one mury": jeśli nawet rozpoczniesz tworzenie nowej definicji klasy lub metody albo zmieni się tożsamość obiektu `self`, zainicjowane zmienne globalne nadal będą dostępne.

Inaczej mówiąc, zmienne globalne nigdy nie są poza zasięgiem (wyjątkiem są zmienne globalne wątku lokalnego, które zostaną omówione w rozdziale 14.). W oma-

wianym przykładzie metoda zdefiniowana wewnątrz definicji klasy (dwa zasięgi usunięto z zasięgu zewnętrznego lub zasięgu najwyższego poziomu programu) ma dostęp do zmiennej globalnej zainicjowanej na początku kodu:

```
$gvar = "Jestem globalna!"
class C
  def examine_global
    puts $gvar
  end
end
c = C.new
c.examine_global  ←——— Dane wyjściowe: "Jestem globalna!"
```

Zmienna `$gvar` przekazała jasno i wyraźnie komunikat: `Jestem globalna!`. Jeśli wszystkie wystąpienia zmiennej `$gvar` zmienisz na zmienną inną niż globalna (np. `local_var`), stwierdzisz, że zmienna najwyższego poziomu `local_var` nie znajduje się w zasięgu wewnątrz bloku definicji metody.

### WBUDOWANE ZMIENNE GLOBALNE

Interpreter języka Ruby rozpoczyna pracę od dość dużej liczby już zainicjowanych zmiennych globalnych. W zmiennych tych są przechowywane informacje, które potencjalnie mogą zostać użyte w dowolnym miejscu programu. Na przykład zmienna globalna `$0` zawiera nazwę pliku startowego aktualnie działającego programu. Zmienna globalna `$:` zawiera katalogi tworzące ścieżkę, którą interpreter języka Ruby przeszukuje podczas ładowania pliku zewnętrznego. Zmienna `$$` zawiera identyfikator procesu Ruby. Istnieje więcej zmiennych globalnych.

> **WSKAZÓWKA** Dobrym miejscem, w którym możesz znaleźć opisy wszystkich wbudowanych zmiennych globalnych, jakie prawdopodobnie będą potrzebne, jest plik *English.rb* wchodzący w skład instalacji języka Ruby. Plik ten zapewnia mniej tajemnicze nazwy dla znanego zestawu zagadkowych zmiennych globalnych (nie wiń języka Ruby za nazwy — większość z nich wywodzi się z języków powłoki i/lub języków Perl i awk). Aby w programach użyć trochę bardziej przyjaznych nazw, możesz zastosować instrukcję `require "English"`, po której możesz odwołać się do zmiennej `$IGNORECASE`, a nie do zmiennej `$=`, do zmiennej `$PID` zamiast do zmiennej `$$` itd. W przypadku kilku zmiennych globalnych wcześniej ładowane są nazwy wyrażone w języku angielskim. Zamiast `$:` możesz użyć nazwy zmiennej `$LOAD_PATH` nawet bez ładowania pliku *English.rb*.

Tworzenie własnych zmiennych globalnych może być kuszące, zwłaszcza dla początkujących programistów oraz osób uczących się nowego języka (nie dotyczy to tylko języka Ruby). Jednak rzadko jest to dobra lub właściwa decyzja.

### ZALETY I WADY ZMIENNYCH GLOBALNYCH

Zmienne globalne wydają się rozwiązywać wiele problemów projektowych: nie musisz przejmować się zasięgiem, a ponadto wiele klas może współużytkować informacje przez przechowywanie ich w zmiennych globalnych, a nie tworzenie obiektów, które

muszą być odpytywane za pomocą wywołań metod. Bez wątpienia zmienne globalne mają swoisty urok.

Są one jednak w bardzo niewielkim stopniu używane przez doświadczonych programistów. Powody, dla których unika się zmiennych globalnych, są podobne do tych, dla których można się skusić na ich użycie. Użycie zmiennych globalnych kończy się zwykle uzyskaniem czegoś, co zamiast wnosić coś do solidnego i elastycznego projektu programu, stanowi jego namiastkę. Jednym z podstawowych elementów programowania obiektowego jest to, że dane i działania są hermetyzowane w obiektach. *Powinno* być konieczne kierowanie zapytań do obiektów w celu uzyskania informacji i żądanie od nich realizowania działań.

Z kolei obiekty mają mieć określoną prywatność. Gdy żądasz od obiektu wykonania czegoś, nie powinieneś przejmować się tym, jakie obiekt wykonuje wewnętrzne działania w celu zrealizowania zadania. Jeśli nawet sam utworzyłeś kod metod obiektu, to w momencie wysyłania komunikatu obiektowi traktujesz go jak „czarną skrzynkę", która działa w tle i udostępnia odpowiedź.

Zmienne globalne „zniekształcają krajobraz", zapewniając warstwę informacji współużytkowanych przez każdy obiekt w każdym kontekście. W efekcie obiekty przestają komunikować się ze sobą, dzieląc się zamiast tego informacjami za pomocą ustawianych zmiennych globalnych.

Pora na niewielki przykład, czyli przebudowaną klasę `Person`, którą zaprezentowano wcześniej (zawiera imię, opcjonalne drugie imię i nazwisko). Tym razem zamiast atrybutów obiektu na podstawie zmiennych globalnych zostaną wygenerowane personalia:

```
class Person
  def whole_name
    n = $first_name + " "
    n << "#{$middle_name} " if $middle_name
    n << $last_name
  end
end
```

Aby użyć tej klasy i z jej instancji uzyskać personalia, konieczne będzie zastosowanie następującego kodu:

```
david = Person.new
$first_name = "Dawid"
$middle_name = "Adam"
$last_name = "Nowak"
puts david.whole_name  ←——— Dane wyjściowe: Dawid Adam Nowak
```

W tej wersji kodu personalia w dalszym ciągu są uzyskiwane z zewnątrz przez odpytanie obiektu. Jednak składniki personaliów są przekazywane „ponad głowami" obiektów, jak to można określić, w osobnej sieci zmiennych globalnych. Choć jest to zwięzłe i proste, jednocześnie znacznie ograniczone. Co by się stało, gdyby istniało wiele obiektów `Person` albo pożądane byłoby zapisanie w bazie danych obiektu `Person`, włącznie z jego różnymi nazwami? Delikatnie mówiąc, kod szybko stałby się zagmatwany.

Zasadniczo dane o zasięgu globalnym są sprzeczne z filozofią obiektową, zgodnie z którą obiekty są wyposażane w możliwości, a następnie działania są realizowane przez wysyłanie żądań do tych obiektów. Niektórzy programiści używający języka Ruby od wielu lat nigdy nie skorzystali z ani jednej zmiennej globalnej (być może z wyjątkiem kilku wbudowanych). Choć możesz nie mieć takich doświadczeń, nie jest to zły cel do obrania.

Po zakończeniu części „spróbuję tego nie robić" przejdźmy do szczegółowego omówienia zasięgu lokalnego.

### 5.2.2. Zasięg lokalny

*Zasięg lokalny* to podstawowa warstwa „materiału" każdego programu Ruby. W dowolnym danym momencie program używa konkretnego zasięgu lokalnego. Podstawowa rzecz, jaka zmienia się między kolejnymi zasięgami lokalnymi, to zapewniane zmienne lokalne. Po opuszczeniu zasięgu lokalnego (w ramach powrotu z wywołania metody lub przez wykonanie jakiegoś działania, które spowoduje utworzenie nowego zasięgu lokalnego) uzyskujesz nowy zestaw zmiennych. Jeśli nawet dokonano przypisania do zmiennej lokalnej `x` w jednym zasięgu, możesz zdefiniować przypisanie do nowej zmiennej `x` w nowym zasięgu. Te dwie zmienne `x` nie będą ze sobą kolidować.

Analizując program Ruby, możesz stwierdzić, gdzie rozpoczynają się i kończą zasięgi lokalne na podstawie kilku następujących reguł:

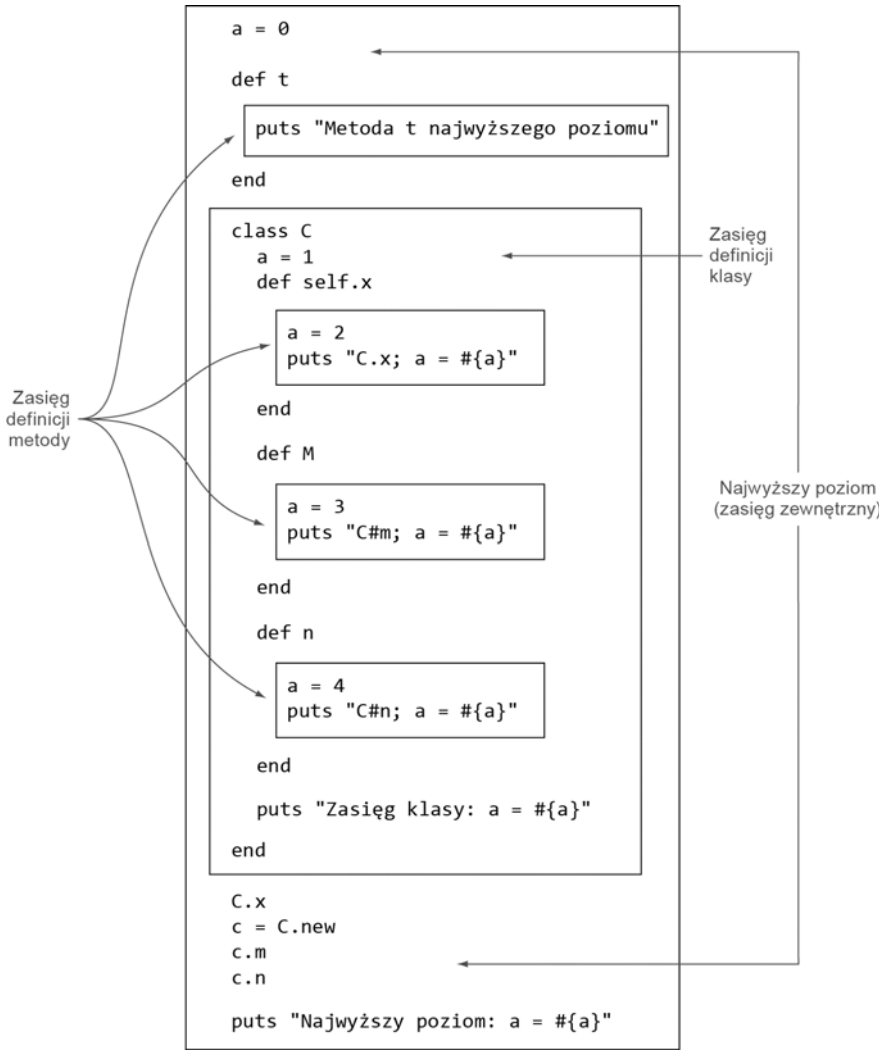
- Z najwyższym poziomem (poza obrębem wszystkich bloków definicji) powiązany jest jego własny zasięg lokalny.
- Każdy blok definicji klasy lub modułu (słowa kluczowe `class` i `module`) ma własny zasięg lokalny. Dotyczy to nawet tego rodzaju bloków, które są zagnieżdżone.
- Każda definicja metody (słowo kluczowe `def`) ma własny zasięg lokalny. Dokładniej mówiąc, każde wywołanie metody generuje nowy zasięg lokalny z wszystkimi zmiennymi lokalnymi zresetowanymi do stanu niezdefiniowanego.

Choć istnieją wyjątki od tych reguł oraz elementy rozszerzające je, jest ich naprawdę niewiele, dlatego nie musimy się nimi teraz zajmować.

Na rysunku 5.2 zilustrowano proces tworzenia kilku zasięgów lokalnych.

Każdorazowo po przekroczeniu granicy bloku definicji klasy, modułu lub metody (czyli po napotkaniu słowa kluczowego `class`, `module` lub `def`) inicjowany jest nowy zasięg lokalny. Zmienne lokalne znajdujące się fizycznie bardzo blisko siebie tak naprawdę mogą nie mieć ze sobą nic wspólnego. Potwierdza to następujący przykład:

```
class C
  a = 1        ←— ❶
  def local_a
    a = 2      ←— ❷
    puts a
  end
  puts a       ←— ❸
end
c = C.new
c.local_a      ←— ❹
```




**Rysunek 5.2.** Schematyczny widok zasięgów lokalnych na najwyższym poziomie, na poziomie definicji klasy oraz na poziomie definicji metody

Kod generuje następujące dane wyjściowe:

```
1
2
```

Zmienna `a` inicjowana w zasięgu lokalnym definicji klasy ❶ znajduje się w innym zasięgu niż zmienna `a` położona wewnątrz definicji metody ❷. Po dotarciu do instrukcji `puts` umieszczonej za definicją metody ❸ ponownie aktywowany jest zasięg lokalny definicji klasy. Wyświetlana zmienna `a` jest tą, która została zainicjowana na początku kodu ❶, a nie zmienną `a` zlokalizowaną w zasięgu definicji metody ❷. Z kolei druga zmienna `a` nie jest wyświetlana do momentu utworzenia instancji `c` i wysłania do niej komunikatu `local_a` ❹.

W przypadku zagnieżdżania klas i modułów każde aktywowanie nowego bloku definicji powoduje utworzenie nowego zasięgu lokalnego. Listing 5.4 prezentuje wielopoziomowe zagnieżdżanie klas i modułów z kilkoma zmiennymi `a`, które są inicjowane i wyświetlane w trakcie wykonywania kodu.

**Listing 5.4. Ponowne wykorzystanie nazwy zmiennej w zagnieżdżonych zasięgach lokalnych**

```
class C
  a = 5
  module M
    a = 4
    module N
      a = 3
      class D
        a = 2
        def show_a
          a = 1
          puts a
        end
        puts a   <------ Dane wyjściowe: 2
      end
      puts a   <------ Dane wyjściowe: 3
    end
    puts a   <------ Dane wyjściowe: 4
  end
  puts a   <------ Dane wyjściowe: 5
end
d = C::M::N::D.new
d.show_a   <------ Dane wyjściowe: 1
```

Każdy blok definicji (niezależnie od tego, czy klasy, modułu, czy metody) inicjuje nowy zasięg lokalny („pamięć tymczasowa" nowej zmiennej lokalnej) i uzyskuje własną zmienną `a`. Przykład ilustruje również to, że cały kod w blokach definicji klasy i modułu jest wykonywany przy pierwszym napotkaniu go, natomiast metody nie są wykonywane do momentu wysłania obiektowi odpowiedniego komunikatu. Z tego właśnie powodu wartość zmiennej `a` ustawionej wewnątrz metody `show_a` jest wyświetlana jako ostatnia wśród pięciu wartości prezentowanych przez program. Pozostałe cztery wartości zmiennych są przetwarzane po napotkaniu ich w definicjach klasy lub modułu. Jednak ostatnia wartość nie jest przetwarzana do momentu wykonania metody `show_a` przez obiekt `d`.

Jak widać, zasięg lokalny często się zmienia. Dotyczy to również tożsamości obiektu `self`. Czasami, ale tylko czasami, zmienia się zarówno zasięg, jak i obiekt `self`. Przyjrzyjmy się trochę bliżej relacji między zasięgiem i obiektem `self`.

### 5.2.3. Interakcja między zasięgiem lokalnym i obiektem self

W momencie rozpoczynania bloku definicji (metody, klasy lub modułu) inicjowany jest nowy zasięg lokalny, a ponadto tworzony jest blok kodu z konkretnym obiektem `self`. Jednak zasięg lokalny i obiekt `self` nie funkcjonują całkowicie równolegle, nie tylko dlatego, że nie są tym samym, ale też dlatego, że nie są elementami tego samego *rodzaju*.

Przeanalizuj listing 5.5. W programie wykorzystano *rekurencję*: metoda instancji `x` wywołuje samą siebie. Celem listingu jest zademonstrowanie tego, że każdorazowe wywołanie metody (jeśli nawet poprzednie wywołanie metody w dalszym ciągu działa) powoduje wygenerowanie nowego zasięgu lokalnego.

**Listing 5.5. Demonstracja generowania nowego zasięgu lokalnego dla wywołania metody**

```
class C
  def x(value_for_a,recurse=false)                    ❶
    a = value_for_a                                   ❷
    print "Łańcuch inspekcji dla obiektu self:"
    p self                                            ❸
    puts "Wartość zmiennej a:"
    puts a
    if recurse                                        ❹
      puts "Wywoływanie samej siebie (rekurencja)..."
      x("Druga wartość zmiennej a")                   ❺
      puts "Powrót po rekurencji. Wartość zmiennej a:" ❻
      puts a
    end
  end
end
c = C.new
c.x("Pierwsza wartość zmiennej a", true)              ❼
```

Metoda instancji `C#x` pobiera dwa argumenty: wartość przypisywaną zmiennej `a` oraz flagę informującą metodę, czy ma wywołać samą siebie ❶ (użycie flagi zapewnia sposób zapobiegnięcia nieskończonej rekurencji). W pierwszym wierszu kodu metody inicjowana jest zmienna `a` ❷. Kilka następnych wierszy metody wyświetla reprezentację łańcuchową obiektu `self` i wartość zmiennej `a` ❸.

Pora na podjęcie decyzji: stosować rekurencję czy nie? Jest to zależne od wartości zmiennej `recurse` ❹. Jeśli rekurencja ma miejsce, powoduje wywołanie metody `x` bez określania wartości dla parametru `recurse` ❺. Jego wartością domyślną będzie `false`. Rekurencja nie wystąpi drugi raz.

W wywołaniu rekurencyjnym używana jest inna wartość argumentu `value_for_a`. A zatem w czasie tego wywołania będą wyświetlane różne informacje. Jednak od razu po zwróceniu sterowania z wywołania rekurencyjnego stwierdzamy, że nie zmieniła się wartość zmiennej `a` w tym uruchomieniu metody `x` ❻. Mówiąc w skrócie, każde wywołanie metody `x` powoduje utworzenie nowego zasięgu lokalnego, nawet pomimo tego, że nie zmienia się obiekt `self`.

Dane wyjściowe w przypadku wywołania metody `x` w instancji klasy `C` i ustawienia flagi `recurse` na wartość `true` ❼ mają następującą postać:

```
Łańcuch inspekcji dla obiektu self: #<C:0x00000101b25be0>
Wartość zmiennej a:
Pierwsza wartość zmiennej a
Wywoływanie samej siebie (rekurencja)...
Łańcuch inspekcji dla obiektu self: #<C:0x00000101b25be0>
Wartość zmiennej a:
```

```
Druga wartość zmiennej a
Powrót po rekurencji. Wartość zmiennej a:
Pierwsza wartość zmiennej a
```

Nie zmienia się obiekt `self`, ale resetowane są zmienne lokalne.

> **WSKAZÓWKA** Zamiast wyświetlać domyślną reprezentację łańcuchową obiektu, możesz też użyć metody `object_id` w celu identyfikowania obiektu w unikatowy sposób. Spróbuj w poprzednim przykładzie zmienić `p self` na `puts self.object_id` oraz `puts a` na `puts a.object_id`.

Jeśli powyższy listing sprawia wrażenie okrężnego sposobu pokazania, że każde wywołanie metody ma własny zasięg lokalny, potraktuj to jako szablon lub model odnoszący się do rodzajów demonstracji, jakie możesz sam wypróbować podczas procesu coraz głębszego poznawania sposobu działania zasięgu i obiektu `self` (osobno i razem).

> **UWAGA** Możliwe jest też uzyskanie efektu odwrotnego do tego, co demonstruje listing 5.5, czyli zmiany obiektu `self` bez aktywowania nowego zasięgu lokalnego. Jest to osiągane za pomocą metod `instance_eval` i `instance_exec`, które zostaną omówione w dalszej części książki.

Podobnie jak zmienne, tak i stałe są zarządzane za pomocą reguł zasięgu. Wkrótce przyjrzymy się zasadom działania tych reguł.

### 5.2.4. Zasięg i określanie stałych

Jak pokazano, stałe mogą być definiowane wewnątrz bloków definicji klas i metod. Jeśli znasz łańcuch zagnieżdżonych definicji, dostęp do stałej możesz uzyskać z dowolnego miejsca. Przeanalizuj następujące zagnieżdżenie:

```
module M
  class C
    class D
      module N
        X = 1
      end
    end
  end
end
```

Możesz odwołać się do modułu `M`, klasy `M::C` itd., aż do zwykłej stałej `M::C::D::N::X` (odpowiada wartości 1).

W przypadku stałych występuje swego rodzaju globalna widoczność lub osiągalność: dopóki znasz ścieżkę do stałej, która obejmuje klasy i/lub moduły z zagnieżdżoną stałą, możesz uzyskać do niej dostęp. Gdy jednak pominie się zagnieżdżanie stałych, zdecydowanie nie są one globalne. Stała `X` w jednym zasięgu nie jest stałą `X` w innym zasięgu:

```
module M
  class C
    X = 2
```

```
    class D
      module N
        X = 1
      end
    end
  end
end
puts M::C::D::N::X  ←❶
puts M::C::X  ←❷
```

Zgodnie z hierarchią zagnieżdżania pierwsza instrukcja `puts` ❶ zapewnia wartość 1, a druga ❷ taka instrukcja zwraca wartość 2. Konkretny identyfikator stałej (np. `X`) nie ma absolutnie żadnego wpływu na działanie zmiennej globalnej (np. `$x`).

*Wyszukiwanie stałych*, czyli proces określania identyfikatora stałych lub znajdowania właściwego dopasowania dla nich, w dużym stopniu przypomina wyszukiwanie pliku w systemie plików w konkretnym katalogu. Stałe są identyfikowane względem punktu wykonywania. Ilustruje to inny wariant wcześniejszego przykładu:

```
module M
  class C
    class D
      module N
        X = 1
      end
    end
    puts D::N::X  ←  Dane wyjściowe: 1
  end
end
```

W tym kodzie identyfikator `D::N::X` jest interpretowany względem miejsca jego występowania, czyli w obrębie bloku definicji klasy `M::C`. Z punktu widzenia tej klasy klasa `D` jest położona tylko jeden poziom dalej. Nie ma potrzeby używania `M::C::D::N::X`, gdy `D::N::X` po prostu wskazuje ścieżkę bezpośrednio do właściwej stałej. Z całą pewnością uzyskujemy to, czego żądamy, czyli wyświetlenie liczby 1.

### WYMUSZANIE ŚCIEŻKI BEZWZGLĘDNEJ STAŁEJ

Czasami ścieżka względna nie jest wymagana. Czasami tak naprawdę chcesz rozpocząć proces wyszukiwania stałych na najwyższym poziomie, tak samo jak czasami po prostu musisz użyć dla pliku ścieżki bezwzględnej.

Może to mieć miejsce, gdy tworzysz klasę lub moduł o nazwie podobnej do nazwy wbudowanej klasy lub modułu języka Ruby. Na przykład język ten oferuje klasę `String`. Jeśli jednak utworzysz klasę `Violin`, możesz też zastosować własną klasę o nazwie `String`:

```
class Violin
  class String
    attr_accessor :pitch
    def initialize(pitch)
      @pitch = pitch
    end
  end
  def initialize
```

```
@e = String.new("E")   ←❶
@a = String.new("A")
...itd....
```

Stała `String` w tym kontekście ❶ zgodnie z definicją określa `Violin::String`. Załóżmy, że gdzieś w ogólnej definicji klasy `Violin` musisz odwołać się do wbudowanej klasy `String` języka Ruby. W przypadku zwykłego użycia odwołania do klasy `String` zostanie dla niego określona klasa `Violin::String`. Aby zapewnić, że odwołanie dotyczy wbudowanej, oryginalnej klasy `String`, niezbędne jest umieszczenie na początku nazwy klasy separatora ścieżki klasy w postaci dwóch dwukropków (`::`):

```
def history
  ::String.new(maker + ", " + date)
end
```

Dzięki temu zamiast obiektu `Violin::String` uzyskuje się obiekt `String` języka Ruby. Podobnie jak znak `/` na początku ścieżki pliku znaki `::` znajdujące się na początku nazwy stałej przekazują informację: „rozpocznij wyszukiwanie tego na najwyższym poziomie" (zgadza się, możesz za pomocą interpolacji po prostu połączyć w całość łańcuch w obrębie cudzysłowów i pominąć `String.new`, ale w takiej sytuacji nie uzyskalibyśmy tak obrazowego przykładu kolizji nazw!).

Oprócz stałych oraz zmiennych lokalnych, zmiennych instancji i zmiennych globalnych język Ruby oferuje też *zmienne klasy*, czyli kategorię identyfikatora powiązaną z kilkoma specyficznymi regułami określania zasięgu.

### 5.2.5. Składnia zmiennych klasy, zasięg i widoczność

Nazwa zmiennych klasy rozpoczyna się od dwóch znaków `@` (na przykład `@@var`). Wbrew swojej nazwie zmienne klasy nie mają zasięgu w postaci klasy. Ich zasięgiem jest raczej hierarchia klas, z występującym czasem wyjątkiem. Nie martw się, zostanie to omówione dokładnie. Po przyjrzeniu się sposobowi działania zmiennych klasy sprawdzimy, jak dobrze radzą sobie one z realizowaniem roli polegającej na utrzymywaniu stanu klasy.

#### ZMIENNE KLASY W KLASACH I INSTANCJACH

Zasadniczo celem zmiennej klasy jest zapewnienie mechanizmu przechowywania, który jest współużytkowany między klasą i jej instancjami, a ponadto nie jest widoczny dla żadnego innego obiektu. Takiej roli nie może pełnić żaden inny element. Zmienne lokalne nie pozostaną po zmianie zasięgu między definicjami klasy i ich wewnętrznymi definicjami metody. Zmienne globalne pozostają, ale są widoczne i mogą się zmieniać w dowolnym innym miejscu programu, a nie tylko w jednej klasie. Nie inaczej jest w przypadku stałych: metody instancji mogą wykryć stałe zdefiniowane w klasie, w której metody zdefiniowano, ale stałe te mogą być też widoczne w reszcie programu. Oczywiście zmienne instancji są widoczne wyłącznie dla poszczególnych obiektów. Klasa nie jest tym samym obiektem co dowolna z jej instancji. Żadne dwie instancje klasy nie są identyczne. A zatem z założenia niemożliwe jest współużytkowanie zmiennej instancji przez klasę ze swoimi instancjami.

Oznacza to, że zmienne klasy mogą wypełnić powstałą „niszę”, czyli widoczność dla klasy i jej instancji, ale nie dla żadnego innego obiektu. Zwykle należy przez to rozumieć widoczność w definicjach metody klasy i metod instancji, a czasem na najwyższym poziomie definicji klasy.

Oto przykład: niewielki program śledzący samochody. Zacznijmy od próby i uzyskania danych wyjściowych. Następnie przyjrzymy się działaniu programu. Załóżmy, że chcemy zarejestrować nazwy producentów samochodów, co umożliwi użycie metody klasy `Car.add_make(make)`. Po zarejestrowaniu producenta możemy produkować jego samochody za pomocą metody `Car.new(make)`. Zarejestrujemy producentów Honda i Ford, po czym wyprodukujemy dwa samochody marki Honda i jeden marki Ford:

```
Car.add_make("Honda")
Car.add_make("Ford")
h = Car.new("Honda")
f = Car.new("Ford")
h2 = Car.new("Honda")
```

Program informuje, jakie samochody są produkowane:

```
Produkowanie nowego samochodu Honda!
Produkowanie nowego samochodu Ford!
Produkowanie nowego samochodu Honda!
```

Na tym etapie możemy uzyskać trochę informacji. Ile istnieje samochodów tego samego producenta `h2`? Aby się tego dowiedzieć, użyjemy metody instancji `make_mates`, interpolując wynik do postaci łańcucha:

```
puts "Liczenie samochodów tego samego producenta h2..."
puts "Liczba samochodów: #{h2.make_mates}."
```

Zgodnie z oczekiwaniami istnieją dwa samochody producenta `h2` (czyli firmy Honda).

Jaka jest łączna liczba istniejących samochodów? Tego rodzaju informacje znajdują się w klasie, a nie w obiektach poszczególnych samochodów, dlatego żądanie zostanie skierowane do klasy:

```
puts "Liczenie wszystkich samochodów..."
puts "Liczba samochodów: #{Car.total_count}."
```

Oto dane wyjściowe:

```
Liczenie wszystkich samochodów...
Liczba samochodów: 3.
```

Na końcu spróbujemy wyprodukować samochód nieistniejącego producenta:

```
x = Car.new("Marka X")
```

Ponieważ programowi się to „nie spodobało”, wystąpił błąd krytyczny:

```
car.rb:21:in `initialize': Brak takiego producenta: Marka X. (RuntimeError)
```

W tym przypadku głównym działaniem jest produkcja samochodów, a także możliwość przechowywania obiektów poszczególnych samochodów i klasy `Car` oraz zwracania statystyk dotyczących wyprodukowanych samochodów. Listing 5.6 prezentuje program.

Jeśli zapiszesz kod z tego listingu, a następnie na końcu pliku dodasz wcześniejszy przykładowy kod, możesz uruchomić cały plik i zobaczyć dane wyjściowe kodu.

**Listing 5.6. Śledzenie statystyk dotyczących produkcji samochodów za pomocą zmiennych klasy**

```
class Car
  @@makes = []
  @@cars = {}                    ❶
  @@total_count = 0
  attr_reader :make              ❷
  def self.total_count           ❸
    @@total_count
  end
  def self.add_make(make)        ❹
    unless @@makes.include?(make)
      @@makes << make
      @@cars[make] = 0
    end
  end
  def initialize(make)
    if @@makes.include?(make)
      puts "Produkowanie nowego samochodu #{make}!"
      @make = make               ❺
      @@cars[make] += 1          ❻
      @@total_count += 1
    else      raise "Brak takiego producenta: #{make}."   ❼
    end
  end
  def make_mates                 ❽
    @@cars[self.make]
  end
end
```

Kluczowa dla programu jest obecność trzech zmiennych klasy zdefiniowanych na początku definicji klasy ❶. `@@makes` to tablica, która przechowuje nazwy producentów. `@@cars` to *tablica asocjacyjna*, czyli struktura, której klucze są producentami samochodów, a odpowiadające im wartości określają liczbę samochodów każdego producenta. Z kolei zmienna klasy `@@total_count` to działający rejestr liczby wszystkich wyprodukowanych samochodów.

Klasa `Car` zawiera też atrybut odczytujący `make` ❷, który umożliwia skierowanie do każdego obiektu samochodu zapytania dotyczącego jego producenta. Wartość tego atrybutu musi zostać ustawiona w momencie tworzenia obiektu samochodu. Nie istnieje atrybut zapisujący dla producentów samochodów, ponieważ nie chcemy, aby poza obrębem klasy występował kod, który zmienia producentów już istniejących samochodów.

W celu zapewnienia dostępu do zmiennej klasy `@@total_count` klasa `Car` definiuje metodę `total_count` ❸, która zwraca bieżącą wartość tej zmiennej. Istnieje też metoda klasy o nazwie `add_make` ❹, która pobiera pojedynczy argument i dodaje go do tablicy znanych producentów samochodów za pomocą operatora dołączania do tablicy `<<`.

Metoda upewnia się najpierw, czy tablica producentów nie zawiera już danego konkretnego producenta. Zakładając, że wszystko jest bez zarzutu, metoda dodaje producenta, a ponadto ustawia na zero licznik samochodów tego producenta. A zatem podczas rejestrowania producenta Honda określamy również to, że nie istnieje żaden samochód tego producenta.

Przejdziemy teraz do metody `initialize`, w której tworzone są nowe obiekty samochodu. Każdy nowy samochód wymaga producenta. Jeśli producent nie istnieje (czyli nie ma go w tablicy `@@makes`), zgłaszany jest błąd krytyczny ❼. W przypadku istnienia producenta dla atrybutu `make` tego samochodu ustawiana jest odpowiednia wartość ❺, o jeden zwiększana jest liczba samochodów danego producenta rejestrowanych w tablicy asocjacyjnej `@@cars` ❻, a ponadto o jeden powiększana jest całkowita liczba istniejących samochodów przechowywana w zmiennej `@@total_count` (być może przypuszczałeś, że ta zmienna reprezentuje sumę wszystkich wartości w tablicy `@@cars`; przechowywanie tej sumy osobno eliminuje utrudnienie związane z sumowaniem wszystkich wartości każdorazowo, gdy żądane jest wyświetlenie sumy). Istnieje również implementacja metody instancji `make_mates` ❽, która zwraca listę wszystkich samochodów danego producenta.

Metoda `initialize` intensywnie korzysta ze zmiennych klasy zdefiniowanych na początku kodu na poziomie zewnętrznym względem definicji klasy. Jest to zasięg lokalny całkowicie inny niż zasięg z wnętrza metody `initialize`, lecz nie tak odmienny w przypadku zastosowania do celów związanych z widocznością zmiennych klasy. Te zmienne klasy były także używane w metodach klasy `Car.total_count` i `Car.add_make`, z których każda ma też własny zasięg lokalny. Możesz zauważyć, że zmienne klasy bazują na własnych regułach: ich widoczność i zasięg nie są zbieżne z widocznością i zasięgiem zmiennych lokalnych. Zmienne klasy uwzględniają wiele wartości obiektu `self` (pamiętaj, że w przypadku zewnętrznego poziomu definicji klasy i wnętrza metod klasy obiektem `self` jest obiekt klasy `Car`, natomiast w metodach instancji obiekt `self` to instancja klasy `Car`, która wywołuje metodę).

Do tej pory zaprezentowano najprostsze aspekty zmiennych klasy. Nawet na takim poziomie różnią się opinie odnośnie do tego (a przynajmniej częstości stosowania takiego rozwiązania), czy dobrym pomysłem jest tworzenie zmiennych, które obejmują wiele obiektów `self`. Czy to, że samochód jest instancją klasy `Car`, naprawdę oznacza, że obiekt `car` i obiekt klasy `Car` muszą współużytkować dane? Czy obiekty te powinny być cały czas traktowane jak dwa osobne obiekty, którymi są?

Nie ma jednej (lub prostej) odpowiedzi. Działanie zmiennych klasy jest jednak trochę bardziej złożone. Bez wątpienia dojdziesz do wniosku, że powinny one być używane z uwagą.

### ZMIENNE KLASY ORAZ HIERARCHIA KLAS

Jak wcześniej wspomniano, zmienne klasy nie są objęte zasięgiem klasy. Są to zmienne, których zasięgiem jest hierarchia klas.

Poniżej przedstawiono przykład. Czego możesz oczekiwać jako danych wyświetlanych przez poniższy kod?

```
class Parent
  @@value = 100  ←——— Ustawia zmienną klasy w klasie Parent
end
class Child < Parent
  @@value = 200  ←——— Ustawia zmienną klasy w klasie Child, czyli podklasie klasy Parent
end
class Parent
  puts @@value   ←——— Powrót do klasy Parent: jakie są dane wyjściowe?
end
```

Wyświetlana jest wartość 200. Klasa `Child` jest podklasą klasy `Parent`. Oznacza to, że klasy `Parent` i `Child` współużytkują te same zmienne klasy — nie różne zmienne o takiej samej nazwie, lecz faktycznie identyczne zmienne. W momencie przypisywania wartości do zmiennej `@@value` w klasie `Child` ustawiasz jedną i tylko jedną taką zmienną, która jest współużytkowana w hierarchii, czyli przez klasy `Parent` i `Child` oraz wszystkie inne klasy potomne dowolnej z nich. Termin *zmienna klasy* staje się trochę trudny do pogodzenia z tym, że dwie (a potencjalnie znacznie więcej) klasy współużytkują dokładnie te same zmienne.

Zgodnie z obietnicą niniejszy podrozdział zostanie zakończony analizą zalet i wad związanych z zastosowaniem zmiennych klasy do utrzymywania stanu w klasie.

### ANALIZOWANIE ZALET I WAD ZMIENNYCH KLASY

Zwykłym sposobem utrzymywania stanu w obiekcie jest użycie zmiennej instancji. Zmienne klasy okazują się przydatne, ponieważ dokonują „rozbicia tamy" między obiektem klasy i jej instancjami. Dzięki działaniu w ten sposób, a w szczególności z powodu ich zasięgu bazującego na hierarchii, zmienne klasy nabierają swego rodzaju jakości quasi-globalnych: zmienna klasy nie jest globalna, ale z pewnością widoczna dla wielu obiektów po dodaniu wszystkich podklas i każdej ich instancji.

Rzecz w tym, że przydatna jest możliwość utrzymywania stanu w klasie. Widoczne to było nawet w przykładzie prostej klasy `Car`. Pożądane było gromadzenie gdzieś informacji związanych z klasą, takich jak producenci samochodów oraz łączna liczba wyprodukowanych aut. Wymagane było również uzyskanie dostępu do tych informacji, zarówno z poziomu metod klasy, jak i metod instancji. Zmienne klasy są popularne, ponieważ stanowią najprostszy sposób dystrybucji danych w takiej konfiguracji.

Zmienne klasy nie są jednak pozbawione mankamentów. W ich posiadaniu może być zbyt wiele obiektów. Załóżmy, że chcemy utworzyć podklasę klasy `Car` o nazwie `Hybrid` w celu przechowywania liczby wyprodukowanych pojazdów, które częściowo korzystają z napędu elektrycznego. Nie będzie możliwe użycie następującego kodu:

```
class Hybrid < Car
end
hy = Hybrid.new("Honda")
puts "Liczba istniejących pojazdów z napędem hybrydowym: #{Hybrid.total_count}!"
```

Wynika to stąd, że metoda `Hybrid.total_count` jest tą samą metodą co metoda `Car.total_`
`↪count`, a ponadto opakowuje identyczną zmienną. Zmienne klasy nie są ponownie inicjowane od nowa dla każdej podklasy, tak jak ma to miejsce w przypadku zmiennych instancji dla każdego obiektu.

Aby osobno śledzić samochody z napędem hybrydowym, konieczne będzie zastosowanie następującego kodu:

```
class Hybrid < Car
  @@total_hybrid_count = 0
  # itp.
end
```

Choć dostępne są sposoby określania abstrakcji i stosowania częściowej automatyzacji tego rodzaju rozdzielania kodu według przestrzeni nazw klas, technika ta nie należy do najłatwiejszych lub najbardziej przejrzystych na świecie.

Jaka jest alternatywa?

### UTRZYMYWANIE STANU POSZCZEGÓLNYCH KLAS ZA POMOCĄ ZMIENNYCH INSTANCJI OBIEKTÓW KLASY

Alternatywą jest powrót do podstaw. Niezbędne jest miejsce, w którym możemy umieścić wartość (łączną liczbę). Dla każdej klasy powinno to być osobne miejsce. Inaczej mówiąc, niezbędne jest utrzymywanie stanu dla poszczególnych klas. Ponieważ klasy są obiektami, oznacza to utrzymywanie stanu dla poszczególnych obiektów (dla określonej grupy obiektów, a mianowicie obiektów klasy). Stan poszczególnych obiektów, niezależnie od tego, czy rozpatrywanego obiektu w klasie, czy czegoś innego, wskazują zmienne instancji.

Listing 5.7 prezentuje przebudowaną klasę `Car` z listingu 5.6. Choć nadal występują dwie zmienne klasy, zmienna `@@total_count` została przekształcona w zmienną instancji.

**Listing 5.7. Klasa Car ze zmienną @@total_count zastąpioną przez zmienną instancji @total_count**

```
class Car
  @@makes = []
  @@cars = {}
  attr_reader :make
  def self.total_count
    @total_count ||= 0      ←— ❶
  end
  def self.total_count=(n)
    @total_count = n        ←— ❷
  end
  def self.add_make(make)
    unless @@makes.include?(make)
      @@makes << make
      @@cars[make] = 0
    end
  end
  def initialize(make)
    if @@makes.include?(make)
      puts "Produkowanie nowego samochodu #{make}!"
      @make = make
      @@cars[make] += 1
      self.class.total_count += 1  ←— ❸
    else
      raise "Brak takiego producenta: #{make}."
    end
```

```
  end
  def make_mates
    @@cars[self.make]
  end
end
```

W tym przypadku kluczem jest przechowywanie licznika w zmiennej instancji należącej do obiektu klasy `Car`, a także opakowywanie tej zmiennej w metodach akcesora (co prawda ręcznie tworzone, ale tego typu metody). Metody akcesora to `Car.total_count` i `Car.total_count=`. Pierwsza z tych metod realizuje zadanie polegające na inicjowaniu zmiennej `@total_count` na wartość zero ❶. Inicjowanie przebiega warunkowo przy użyciu operatora `||=`, dlatego przy drugim i kolejnych wywołaniach metody `total_count` zwracana jest po prostu zmienna instancji.

Metoda `total_count=` jest metodą zapisującą atrybut i jest tworzona podobnie do metody klasy. Dzięki temu obiektem klasy ❷ jest obiekt, którego zmienna instancji jest używana. Dysponując takimi metodami, możemy teraz zwiększyć wartość łącznej liczby w obrębie metody instancji `initialize` przez wywołanie metody `self.class.`
`↪total_count=` ❸.

Korzyści pojawiają się w momencie tworzenia podklasy klasy `Car`. Przyjrzymy się ponownie klasie `Hyvbrid` i przykładowemu kodowi, który z niej korzysta:

```
class Hybrid < Car
end
h3 = Hybrid.new("Honda")
f2 = Hybrid.new("Ford")
puts "Liczba pojazdów z napędem hybrydowym na drodze: #{Hybrid.total_count}!"
```

**Dane wyjściowe: Liczba pojazdów z napędem hybrydowym na drodze: 2!**

`Hybrid` to nowy obiekt klasy. Różni się on od obiektu klasy `Car`. A zatem obiekt ma własne zmienne instancji. W momencie tworzenia nowej instancji klasy `Hybrid` wykonywana jest metoda `initialize` klasy `Car`. Jednak tym razem wyrażenie:

```
self.class.total_count += 1
```

ma inne znaczenie. Odbiorcą komunikatu `total_count=` jest obiekt `Hybrid`, a nie `Car`. Oznacza to, że podczas wykonywania metody klasy `total_count=` zmienna instancji `@total_count` należy do obiektu `Hybrid` (zmienna instancji zawsze należy do obiektu `self`). A zatem dodanie wartości do łącznej liczby obiektu `Hybrid` nie wpłynie na łączną liczbę obiektu `Car`.

Postąpiono w ten sposób, aby podklasa klasy `Car` mogła utrzymać własny stan, ponieważ dokonaliśmy przejścia ze zmiennej klasy do zmiennej instancji. Każdorazowo przy wywołaniu metody `total_count` lub `total_count=` zmienna `@total_count`, do której metoda się odwołuje, jest zmienną należącą do obiektu `self` z chwili wykonywania. Ponownie korzystamy ze zmiennych instancji w celu utrzymania stanu dla poszczególnych obiektów (w tym przypadku obiektów klasy).

Największą przeszkodą w zrozumieniu przedstawionych przykładów jest uzmysłowienie sobie tego, że klasy są obiektami. Każdy obiekt, niezależnie od tego, czy samochód, osoba, *czy klasa*, uzyskuje własny magazyn zmiennych instancji. Dzięki temu, jak zmienne instancji są izolowane dla poszczególnych obiektów, klasy `Car` i `Hybrid` mogą niezależnie śledzić liczby wyprodukowanych samochodów.

Osiągnęliśmy kres naszej podróży po zasięgach identyfikatorów. Zaprezentowano sporą część możliwości zmiennych i stałych (jak również tego, czego nie umożliwiają), a także w jaki sposób możliwości te są łączone z regułami zarządzającymi zasięgiem i obiektem `self`. Z zamiarem zrealizowania celu tego rozdziału, czyli umożliwienia Ci orientowania się w tym, jaki element ma jakie zadania do wykonania i gdzie w kodzie Ruby, przyjrzymy się jeszcze jednemu istotnemu zagadnieniu. Mowa o systemie reguł uzyskiwania dostępu do metod w języku Ruby.

## 5.3. Wdrażanie reguł uzyskiwania dostępu do metod

Jak wcześniej wspomniano, głównym celem programu Ruby jest wysyłanie komunikatów do obiektów. Z kolei podstawowym zadaniem obiektu jest odpowiadanie na komunikaty. Czasami ma potrzebę uzyskania możliwości wysyłania do samego siebie komunikatów, które nie mają być do niego wysyłane przez żaden inny element. W przypadku takiego scenariusza język Ruby zapewnia możliwość utworzenia metody prywatnej.

Istnieją dwa inne poziomy dostępu niż prywatny: chroniony (trochę zmieniona wersja poziomu publicznego) i publiczny. Poziom publiczny to domyślny poziom dostępu. Jeśli nie określono, że metoda jest chroniona lub prywatna, oznacza to, że jest publiczna. Publiczne metody instancji są typowym elementem programowania z wykorzystaniem języka Ruby. Większość komunikatów wysyłanych do obiektów powoduje wywoływanie metod publicznych.

W tym miejscu skoncentrujemy się na metodach, które nie są publiczne, rozpoczynając od metod prywatnych.

### 5.3.1. Metody prywatne

Pomyśl o obiekcie jak o kimś, kogo prosisz o wykonanie dla Ciebie zadania. Załóżmy, że prosisz tę osobę o upieczenie ciasta. W ramach tej czynności ciastkarz prawdopodobnie wykona wiele drobnych zadań: odważenie cukru, rozbicie jajka, wyrobienie ciasta itp.

Ciastkarz wykonuje wszystkie te czynności, ale nie wszystkie mają jednakowy status odnośnie do tego, co cukiernik zamierza zrobić w odpowiedzi na prośby innych osób. Byłoby dziwne, gdybyś po skontaktowaniu się z ciastkarzem powiedział do niego: „Proszę, wyrób trochę ciasta” lub „Proszę, rozbij jajko”. Powiesz natomiast: „Proszę, upiecz mi ciasto”, umożliwiając ciastkarzowi zajęcie się szczegółami z tym związanymi.

Przeprowadźmy modelowanie scenariusza pieczenia. Użyjemy uproszczonych klas zastępczych dla niektórych obiektów związanych z pieczeniem, ale bardziej szczegółowo zdefiniujemy klasę `Baker`.

Zapisz kod z listingu 5.8 w pliku o nazwie *baker.rb*.

**Listing 5.8. Klasa Baker oraz inne klasy związane z pieczeniem**

```
class Cake
  def initialize(batter)
    @batter = batter
```

```
    @baked = true
  end
end
class Egg
end
class Flour
end
class Baker
  def bake_cake
    @batter = []              ←——— Implementuje @batter jako tablicę obiektów (składniki)
    pour_flour
    add_egg
    stir_batter
    return Cake.new(@batter) ←——— Zwraca nowy obiekt Cake
  end
  def pour_flour
    @batter.push(Flour.new)  ←——— Dodaje element (składnik) do tablicy @batter
  end
  def add_egg
    @batter.push(Egg.new)
  end
  def stir_batter
  end
  private :pour_flour, :add_egg, :stir_batter ←——❶
end
```

W tym kodzie pojawia się coś nowego: metoda `private` ❶, która jako argumenty pobiera listę metod mających zostać metodami prywatnymi (jeśli nie chcesz podawać żadnych argumentów, wywołanie metody `private` działa jak przełącznik: oznacza to, że do momentu odwrócenia jego efektu przez wywołanie metody `public` lub `protected` wszystkie metody instancji zdefiniowane poniżej metody `private` będą prywatne).

Metoda prywatna oznacza, że nie może ona być wywoływana za pomocą jawnego odbiorcy. Nie możesz użyć następującego kodu:

```
b = Baker.new
b.add_egg
```

Jeśli spróbujesz użyć tego kodu, to wywołanie w ten sposób metody `add_egg` spowoduje wygenerowanie błędu krytycznego:

```
`<main>': private method `add_egg' called for #<Baker:0x00000002aeae50>
(NoMethodError)
```

`add_egg` to metoda prywatna, ale jawnie określono obiekt odbierający `b`. Jest to niedozwolone.

Dobrze. Skorzystajmy z reguł. Nie będziemy określać odbiorcy. Użyjemy jedynie następującego kodu:

```
add_egg
```

Nie tak szybko. Czy możemy wywołać metodę `add_egg` w izolacji? Dokąd zostanie skierowany komunikat? Jak może zostać wywołana metoda, jeśli nie ma obiektu obsługującego komunikat?

Odpowiedzi na te pytania pozwoli uzyskać trochę działań detektywa.

Jeśli nie użyjesz jawnego odbiorcy dla wywołania metody, interpreter języka Ruby przyjmuje, że zamierzasz wysłać komunikat do bieżącego obiektu `self`. Myśląc logicznie, możesz dojść do wniosku, że komunikat `add_egg` dysponuje obiektem, do którego trafi tylko wtedy, gdy obiekt `self` jest obiektem odpowiadającym metodzie `add_egg`. Inaczej mówiąc, metodę instancji `add_egg` klasy `Baker` możesz wywołać tylko wtedy, gdy obiekt `self` jest instancją tej klasy.

A kiedy obiekt `self` jest instancją klasy `Baker`?

Jest tak w przypadku wykonywania dowolnej metody instancji klasy `Baker`. Na przykład w obrębie definicji metody `bake_cake` możesz wywołać metodę `add_egg`. Interpreter języka Ruby będzie „wiedzieć", jak ma postąpić. Każdorazowo, gdy interpreter napotka takie wywołanie metody `add_egg` wewnątrz tej definicji metody, wyśle komunikat `add_egg` do obiektu `self`, którym jest obiekt `Baker`.

**Metoda prywatna i metoda obiektu pojedynczego nie są tym samym**

Godna uwagi jest różnica między metodą prywatną i metodą obiektu pojedynczego. Metoda obiektu pojedynczego jest „prywatna" w takim nieformalnym i luźnym znaczeniu, że należy do tylko jednego obiektu. Nie jest jednak prywatna w sensie technicznym (możliwe jest ustanowienie takiej metody prywatną, ale domyślnie nią nie jest). Z kolei prywatna metoda instancji, która nie jest metodą obiektu pojedynczego, może być współużytkowana przez dowolną liczbę obiektów, ale może być wywołana w przypadku wystąpienia właściwych okoliczności. Tym, co decyduje, czy możesz wywołać metodę prywatną, nie jest obiekt, do którego wysyłasz komunikat, lecz to, jaki obiekt jest obiektem `self` w momencie wysyłania komunikatu.

Wygląda to następująco: oznaczając metodę `add_egg` jako prywatną, powodujesz, że obiekt `Baker` wyśle ten komunikat do samego siebie (ciastkarz może sam zdecydować o dodaniu jajka do ciasta). Jednak żaden inny element nie może wysłać komunikatu do obiektu `Baker` (jako osoba z zewnątrz nie możesz nakazać ciastkarzowi dodania jajka do ciasta). W języku Ruby tego rodzaju prywatność jest wymuszana za pośrednictwem mechanizmu niedopuszczania jawnego odbiorcy. Jedyną sytuacją umożliwiającą pominięcie odbiorcy jest idealna sytuacja, w której dozwolone jest wywołanie metody prywatnej.

Wszystko to zostało w elegancki sposób skonstruowane. Pojawia się jednak „łyżka dziegciu w beczce miodu".

### PRYWATNE METODY USTAWIAJĄCE (ZE ZNAKIEM =)

Implementacja dostępu prywatnego z wykorzystaniem reguły „bez jawnego odbiorcy" ma miejsce w przypadku metod o nazwach zakończonych znakiem równości. Jak wcześniej wspomniano, w momencie wywoływania metody ustawiającej konieczne jest określenie odbiorcy. Nie jest możliwe użycie następującego kodu:

```
dog_years = age * 7
```

Wynika to stąd, że interpreter języka Ruby uzna, że `dog_years` to zmienna lokalna. Niezbędne jest zastosowanie następującego kodu:

```
self.dog_years = age * 7
```

Wymóg użycia jawnego odbiorcy utrudnia jednak zadeklarowanie metody `dog_years=` jako prywatnej, a przynajmniej zgodnie z logiką reguły „bez jawnego odbiorcy", która musi zostać użyta w przypadku wywoływania metod prywatnych.

Rozwiązaniem tej zagadki jest to, że interpreter języka Ruby nie stosuje reguły w odniesieniu do metod ustawiających. Jeśli zadeklarujesz metodę `dog_years=` jako prywatną, możesz wywołać ją z odbiorcą, pod warunkiem że odbiorcą jest obiekt `self`. Nie może to być kolejne odwołanie do obiektu `self`. Musi to być słowo kluczowe `self`.

Oto implementacja klasy `Dog`, która rozpoznaje metodę `dog_years=`:

```
class Dog
  attr_reader :age, :dog_years
  def dog_years=(years)
    @dog_years = years
  end
  def age=(years)
    @age = years
    self.dog_years = years * 7
  end
  private :dog_years=
end
```

W kodzie wskazujesz wiek psa, po czym automatycznie jest podawany wiek w skali ludzkiej:

```
rover = Dog.new
rover.age = 10
puts "W skali psiej wiek Reksa wynosi: #{rover.dog_years}."
```

**Dane wyjściowe:**
**W skali ludzkiej**
**wiek Reksa wynosi: 70.**

Metoda ustawiająca `age=` realizuje usługę polegającą na ustawianiu wieku w skali psiej. Polega to na wywołaniu metody prywatnej `dog_years=`. Podczas tego działania metoda ustawiająca korzysta z jawnego odbiorcy `self`. Wykonanie tego działania w dowolny inny sposób nie zapewni żądanego efektu. Bez określonego odbiorcy należałoby ustawić zmienną lokalną. Jeśli użyjesz tego samego obiektu, lecz z inną nazwą, taką jak w następującym kodzie:

```
def age=(years)
  @age = years
  dog = self
  dog.dog_years = years * 7
end
```

Wykonywanie kodu zostanie zatrzymane z powodu następującego błędu krytycznego:

```
NoMethodError: private method 'dog_years=' called for
#<Dog:0x00000101b0d1a8 @age=10>
```

W języku Ruby dozwolone jest zastosowanie jawnego odbiorcy dla prywatnych metod ustawiających, ale niezbędne jest zapewnienie równowagi przez zadbanie o to, aby odbiorcą był dokładnie obiekt `self`.

Trzeci poziom dostępu do metod, wraz z metodami publicznymi i prywatnymi, jest chroniony.

### *5.3.2. Metody chronione*

Metoda chroniona przypomina metodę prywatną, która jest trochę bardziej „łagodna" i „uprzejma". W przypadku metod chronionych obowiązuje następująca reguła: metoda chroniona może zostać wywołana w obiekcie `x`, pod warunkiem że obiekt domyślny (`self`) jest instancją tej samej klasy co obiekt `x` bądź klasy nadrzędnej lub potomnej klasy obiektu `x`.

Reguła ta wygląda na zawiłą. Przeważnie jednak używana jest z konkretnego powodu: wymagane jest, aby jedna instancja określonej klasy realizowała jakieś działanie wraz z inną instancją własnej klasy. Taki przypadek prezentuje listing 5.9.

**Listing 5.9. Przykład użycia metody chronionej**

```
class C
  def initialize(n)
    @n = n
  end
  def n
    @n
  end
  def compare(c)
    if c.n > n
      puts "Wartość metody n drugiego obiektu jest większa."
    else
      puts "Wartość metody n drugiego obiektu jest identyczna lub mniejsza."
    end
  end
  protected :n
end
c1 = C.new(100)
c2 = C.new(101)
c1.compare(c2)   ←——— Dane wyjściowe: Wartość metody n drugiego obiektu jest większa
```

Celem kodu z tego listingu jest porównanie jednej instancji klasy `C` z jej kolejną instancją. Porównanie zależy od wyniku wywołania metody `n`. Obiekt wykonujący operację porównania (w przykładzie `c1`) musi zażądać od drugiego obiektu (`c2`) wykonania swojej metody `n`. Oznacza to, że metoda ta nie może być prywatna.

W tym miejscu „do gry wchodzi" poziom chroniony. Gdy metoda `n` jest chroniona, a nie prywatna, obiekt `c1` może zażądać od obiektu `c2` wykonania metody `n`, ponieważ oba obiekty są instancjami tej samej klasy. Jeśli jednak spróbujesz wywołać metodę `n` obiektu klasy `C`, gdy obiektem `self` nie jest nic innego jak instancja klasy `C` (albo jeden z przodków lub potomków tej klasy), metoda nie zadziała.

A zatem metoda chroniona przypomina metodę prywatną, z wyłączeniem kilku przypadków, w których klasa obiektu `self` (`c1`) oraz klasa obiektu z wywoływaną w nim metodą (`c2`) są takie same lub powiązane dziedziczeniem.

Ostatnim zagadnieniem, jakie zostanie omówione w rozdziale, są metody najwyższego poziomu. Jak się okaże, metody najwyższego poziomu mają szczególny status. Jednak nawet pomimo tego status ten łączy się logicznie z aspektami projektu programu Ruby, jakie zaprezentowano w tym rozdziale.

**Dziedziczenie i dostęp do metod**

Podklasy dziedziczą reguły dostępu do metod swoich superklas. Dla danej klasy `C` ze zbiorem reguł dostępu oraz klasy `D`, która jest podklasą klasy `C`, instancje klasy `D` cechują się takim samym zachowaniem związanym z dostępem co instancje klasy `C`. Możliwe jest jednak zdefiniowanie nowych reguł wewnątrz definicji klasy `D`. W tym przypadku nowe reguły dla instancji klasy `D` mają pierwszeństwo przed regułami dziedziczonymi z klasy `C`.

## 5.4. Tworzenie i używanie metod najwyższego poziomu

Najbardziej naturalną rzeczą do zrealizowania w przypadku języka Ruby jest zaprojektowanie klas i modułów oraz utworzenie instancji klas. Czasami jednak wymagane jest jedynie napisanie i wykonanie krótkiego skryptu, który zawiera kilka poleceń umieszczonych w pliku. Niekiedy wygodniejsze od opakowania wszystkiego w definicjach klasy jest utworzenie definicji metody na najwyższym poziomie skryptu, a następnie wywołanie ich w obiektach najwyższego poziomu. W takim przypadku kod jest tworzony w kontekście domyślnego obiektu najwyższego poziomu `main`, który jest instancją klasy `Object` tworzoną automatycznie z jedynego powodu, jakim jest to, że coś musi być obiektem `self`, nawet na najwyższym poziomie.

Ponieważ jednak nie aktywowano definicji modułu lub klasy, co dokładnie ma miejsce podczas definiowania metody?

### 5.4.1. Definiowanie metody najwyższego poziomu

Załóżmy, że na najwyższym poziomie zdefiniujesz następującą metodę:

```
def talk
  puts "Witaj"
end
```

Ponieważ powyższa definicja nie znajduje się w obrębie bloku definicji klasy lub modułu, nie będzie metodą instancji klasy lub modułu. A zatem czym ona jest?

Metoda definiowana na najwyższym poziomie jest przechowywana jako prywatna metoda instancji klasy `Object`. Wcześniejszy kod jest równoważny następującemu kodowi:

```
class Object
  private
def talk
    puts "Witaj"
  end
end
```

Definiowanie prywatnych metod instancji klasy `Object` powoduje kilka interesujących konsekwencji.

Po pierwsze, metody te nie tylko mogą, ale *muszą* być wywoływane w stylu uproszczonym. Z jakiego powodu? Wynika to z tego, że są prywatne. Metody takie mogą być wywoływane tylko w obiekcie `self`, a ponadto wyłącznie bez jawnego odbiorcy (tradycyjnie z wyłączeniem prywatnych metod ustawiających, które wymagają wywoływania z obiektem `self` jako odbiorcą).

Po drugie, prywatne metody instancji klasy `Object` mogą być wywoływane w dowolnym miejscu kodu, ponieważ klasa ta znajduje się w ścieżce wyszukiwania metody każdej klasy (z wyjątkiem klasy `BasicObject`, ale jest to zbyt szczególny przypadek, aby się nim przejmować). A zatem metoda najwyższego poziomu jest zawsze dostępna. Niezależnie od tego, co jest obiektem `self`, będzie on w stanie rozpoznać wysłany do niego komunikat, jeśli zostanie dla niego określona prywatna metoda instancji klasy `Object`.

W celu zilustrowania tego rozszerzmy przykład z metodą `talk`. Oto ponownie ta metoda poszerzona o kod, który ją testuje:

```
def talk
  puts "Witaj"
end
puts "Podejmowanie próby użycia metody talk bez odbiorcy..."
talk                                                          ←❶
puts "Podejmowanie próby użycia metody talk z jawnym odbiorcą..."
obj = Object.new
obj.talk    ←❷
```

Pierwsze wywołanie metody `talk` kończy się powodzeniem ❶. Drugie wywołanie kończy się niepowodzeniem z błędem krytycznym ❷, ponieważ podejmowana jest próba wywołania metody prywatnej z jawnym odbiorcą.

W działaniu metod najwyższego poziomu ciekawe jest to, że zapewniają przydatną funkcjonalność (czyli proste polecenia skryptowe w stylu proceduralnym), co odbywa się w całkowitej zgodności z regułami języka Ruby: metody prywatne muszą domyślnie określić obiekt `self` jako odbiorcę, a metody zdefiniowane w klasie `Object` są widoczne dla wszystkich obiektów. Nie są stosowane żadne dodatkowe konstrukcje językowe, a jedynie kombinacja już istniejących, które cechują się elegancją i dużymi możliwościami.

Reguły powiązane z definicją i użyciem metod najwyższego poziomu zapewniają w pełni powrót do niektórych metod uproszczonych, które zostały przez nas zastosowane już w rozdziale 1. Możesz teraz zrozumieć, jak te metody działają.

### *5.4.2. Predefiniowane (wbudowane) metody najwyższego poziomu*

Począwszy od pierwszych zamieszczonych przykładów, stosowaliśmy uproszczone wywołania metod `puts` i `print`, takie jak następujące:

```
puts "Witaj"
```

Metody `puts` i `print` to wbudowane, prywatne metody instancji modułu `Kernel`, a nie klasy `Object`, tak jak to jest w przypadku metod tworzonych samemu. Efekt końcowy jest jednak podobny (wynika to stąd, że klasa `Object` dołącza moduł `Kernel`): tego rodzaju metody możesz wywołać w dowolnym momencie, a wywołanie nie może zawierać odbiorcy. Moduł `Kernel` zapewnia pokaźny pakiet narzędziowy w postaci metod imperatywnych, takich jak `puts` i `print`, które zwiększają możliwości języka Ruby jako języka skryptowego. Skrypty Ruby, które są pozbawione definicji klasy, modułu lub metody,

umożliwiają zrealizowanie mnóstwa rzeczy, ponieważ na bardzo wiele (odczyt i zapis, uruchamianie poleceń systemowych, zamykanie programu itp.) pozwalają metody najwyższego poziomu języka Ruby.

Aby wyświetlić wszystkie prywatne metody instancji udostępniane przez moduł `Kernel`, wypróbuj następujące polecenie:

```
$ ruby -e 'p Kernel.private_instance_methods.sort'
```

Metoda `private_instance_methods` oferuje tablicę z wszystkimi odpowiednimi metodami, a metoda `sort` sortuje tablicę nazw metod w celu ułatwienia odczytu. Jak można zauważyć, metody te, choć często przydatne w imperatywnym programowaniu skryptowym, pod względem użyteczności nie są ograniczone do tego typu zastosowania. Metody te obejmują takie metody jak `require`, `load` i `raise` (zgłasza wyjątek) oraz inne, które należą do najczęściej stosowanych technik we wszystkich programach Ruby (niezależnie od cechującego je stylu projektowania).

## 5.5. Podsumowanie

W rozdziale omówiono następujące zagadnienia:

- Przechodzenie roli obiektu `self` (obiekt domyślny lub bieżący).
- Obiekt `self` jako odbiorca wywołań metod bez jawnie określonego odbiorcy.
- Obiekt `self` jako właściciel zmiennych instancji.
- Implikacje związane z regułą „klasy to również obiekty".
- Zasięg zmiennych oraz widoczność zmiennych lokalnych, zmiennych globalnych i zmiennych klasy.
- Reguły wyszukiwania i odwoływania się do stałych.
- Poziomy dostępu (publiczny, prywatny i chroniony) do metod w języku Ruby.
- Tworzenie definicji metod najwyższego poziomu i korzystanie z nich.

Techniki zaprezentowane w rozdziale mają duże znaczenie dla języka Ruby. Kluczowe znaczenie mają takie kwestie jak różnica między zmiennymi instancji w definicji klasy i zmiennymi instancji w definicji metod instancji. Z łatwością można przyjrzeć się programowi Ruby i uzyskać ogólną orientację w tym, co ma w nim miejsce. Aby jednak dogłębnie zrozumieć program, a ponadto nabyć umiejętność pisania dobrze zorganizowanych i niezawodnych programów, musisz wiedzieć, jak identyfikować początek i koniec różnych zasięgów lokalnych, w jaki sposób określane są stałe, zmienne instancji i inne identyfikatory, a także jak oceniać wpływ ciągle zmieniającej się roli obiektu `self`.

W rozdziale pokazano, jak zaznajomić się z programem Ruby. Zaprezentowano również techniki, jakie mogą zostać dokładniej i efektywniej wykorzystane w kodzie z racji własnych umiejętności. Jednak w odniesieniu do tego, co jest możliwe do zrealizowania w „krajobrazie" programu, poza jego zrozumieniem jest jeszcze znacznie więcej do poznania. W następnym rozdziale poświęconym zagadnieniu przepływu sterowania zostaną przedstawione niektóre z tych technik.

# 6 Techniki przepływu sterowania

**W tym rozdziale:**

- Wykonywanie warunkowe.
- Pętle i techniki ich wykonywania.
- Iteratory.
- Wyjątki i obsługa błędów.

Jak już zaprezentowano w przypadku wywołań metod, w których sterowanie programu przechodzi z miejsca, gdzie dokonano wywołania, do treści definicji metody, programy nie działają w sposób prostolinijny. Kolejność wykonywania jest określana przez różne reguły i konstrukcje programistyczne, które zbiorczo są nazywane technikami *przepływu sterowania*.

Techniki przepływu sterowania są następujące:

- *Wykonywanie warunkowe*. Wykonywanie zależy od prawdziwości wyrażenia.
- *Wykonywanie pętli*. Pojedynczy segment kodu jest wielokrotnie wykonywany.
- *Iteracja*. Wywołanie metody jest uzupełniane segmentem kodu, który wykonywana metoda może wywołać co najmniej raz.
- *Wyjątki*. Warunki błędu są obsługiwane przez specjalne reguły przepływu sterowania.

Kolejno przyjrzymy się każdej technice. Wszystkie są niezbędne do zrozumienia języka Ruby i korzystania z niego. Pierwsza z wymienionych technik, czyli wykonywanie

warunkowe (instrukcja `if` i inne powiązane z nią), to fundamentalne i proste narzędzie programistyczne, które jest obecne w niemal każdym języku programowania. Wykonywanie pętli to bardziej specjalizowana technika, ale blisko powiązana z wykonywaniem warunkowym; w języku Ruby technika ta może zostać zastosowana na kilka sposobów. Po zajęciu się iteracją znajdziemy się w obszarze, który naprawdę jest charakterystyczny dla języka Ruby. Technika ta nie występuje wyłącznie w języku Ruby, ale jest elementem stosunkowo rzadko pojawiającym się w innych językach programowania. Tymczasem w języku Ruby jest bardzo widoczna. Na końcu przyjrzymy się rozbudowanemu mechanizmowi języka Ruby obsługującemu warunki błędu z wykorzystaniem wyjątków. Wyjątki zatrzymują przepływ programu całkowicie lub do momentu zakończenia obsługi warunku błędu. Wyjątki są obiektami. Możesz tworzyć własne klasy wyjątków dziedziczące z klas wbudowanych języka Ruby, które będą zapewniać specjalizowaną obsługę warunków błędów w programach.

## 6.1. Warunkowe wykonywanie kodu

*Jeśli hasło jest poprawne, zezwól użytkownikowi na uzyskanie dostępu do lokalizacji. Wyświetl komunikat o błędzie, jeśli żądany element nie istnieje. Przyznaj się do porażki, jeśli król został zaszachowany.* Nieskończona jest lista zastosowań warunkowego sterowania przepływem programu, czyli wykonywania konkretnych wierszy lub segmentów kodu tylko w przypadku spełnienia określonych warunków. Bez zbytniego filozofowania możemy nawet stwierdzić, że podejmowanie decyzji na podstawie nieprzewidywalnych, lecz dostrzegalnych warunków jest tak częste podczas programowania jak w życiu.

W języku Ruby oferowanych jest kilka sposobów sterowania przepływem programu na podstawie warunków. Najważniejsze metody zaliczają się do następujących dwóch kategorii:

- instrukcja `if` i powiązane z nią słowa kluczowe,
- instrukcje `case`.

W niniejszym podrozdziale przyjrzymy się obu kategoriom.

### 6.1.1. Instrukcja if i powiązane z nią instrukcje

Nie jest zaskoczeniem to, że siłą napędową wykonywania warunkowego jest słowo kluczowe `if`. Klauzule z instrukcją `if` mogą przyjmować kilka postaci. Oto najprostsza z nich:

```
if condition
  # w tym miejscu znajduje się kod wykonywany, gdy warunek condition jest prawdziwy
end
```

Kod w obrębie bloku instrukcji warunkowej może być dowolnej długości, a ponadto może obejmować zagnieżdżone bloki warunkowe.

Całą klauzulę `if` możesz również umieścić w pojedynczym wierszu za pomocą słowa kluczowego `then` występującego po warunku:

```
if x > 10 then puts x end
```

Możliwe jest też zastosowanie średników w celu imitowania podziałów wierszy i podkreślenia słowa kluczowego `end`:

```
if x > 10; puts x; end
```

Wykonywanie warunkowe często uwzględnia więcej niż jedną gałąź. Może być pożądane zrealizowanie jednego działania w przypadku spełnienia warunku lub innego działania, gdy warunek nie zostanie spełniony. Jeśli na przykład *hasło jest poprawne, zezwól użytkownikowi na uzyskanie dostępu. W przeciwnym razie wyświetl komunikat o błędzie*. Udostępniając instrukcje `else` i `elsif`, w języku Ruby zapewnia się pełną obsługę wielu gałęzi warunkowych.

### SŁOWA KLUCZOWE ELSE I ELSIF

W instrukcji `if` możesz umieścić gałąź `else` w następujący sposób:

```
if condition
  # kod wykonywany, gdy warunek condition jest prawdziwy
else
  # kod wykonywany, gdy warunek condition nie jest prawdziwy
end
```

Dostępne jest też słowo kluczowe `elsif`. Umożliwia ono tworzenie kaskady logiki warunkowej przy użyciu większej liczby poziomów niż w przypadku samych tylko instrukcji `if` i `else`:

```
if condition1
  # kod wykonywany, gdy warunek condition1 jest prawdziwy
elsif condition2
  # kod wykonywany, gdy warunek condition1 nie jest prawdziwy,
  # a warunek condition2 jest prawdziwy
elsif condition3
  # kod wykonywany, jeśli ani warunek condition1,
  # ani warunek condition2 nie jest prawdziwy, lecz warunek condition3
end
```

W danej instrukcji `if` może występować dowolna liczba klauzul `elsif`. Wykonywany jest segment kodu odpowiadający pierwszej pomyślnej instrukcji `if` lub `elsif`, a reszta instrukcji jest ignorowana:

```
print "Wprowadź liczbę całkowitą: "
n = gets.to_i
if n > 0
  puts "Podana liczba jest dodatnia."
elsif n < 0
  puts "Podana liczba jest ujemna."
else
  puts "Podana liczba wynosi zero."
end
```

Zauważ, że ostatniej instrukcji `else` możesz użyć nawet wtedy, gdy wystąpiła już co najmniej jedna instrukcja `elsif`. Klauzula `else` jest wykonywana, gdy nie powiódł się żaden z wcześniejszych testów prawdziwości. Jeżeli żaden z warunków nie jest prawdziwy, a ponadto nie ma klauzuli `else`, cała instrukcja `if` kończy się bez wykonywania działania.

Czasami pożądane jest, aby warunek instrukcji `if` był negujący: *jeśli coś nie jest prawdziwe, wykonaj dany segment kodu*. Możliwe jest to do zrealizowania na kilka sposobów.

### NEGACJA WARUNKÓW ZA POMOCĄ SŁOWA KLUCZOWEGO NOT I ZNAKU !

Jeden ze sposobów negacji warunku polega na użyciu słowa kluczowego `not`:

```
if not (x == 1)
```

Możliwe jest też zastosowanie operatora negacji w postaci wykrzyknika `!`:

```
if !(x == 1)
```

W obu powyższych przykładach użyto nawiasów okrągłych, aby wyróżnić testowane wyrażenie. W pierwszym przykładzie nawiasy nie są konieczne. Możliwe jest użycie następującego kodu:

```
if not x == 1
```

Z kolei w drugim przykładzie nawiasy są *niezbędne*, ponieważ operator negacji `!` ma pierwszeństwo względem operatora ==. Inaczej mówiąc, zastosowanie następującego kodu:

```
if !x == 1
```

Spowoduje w rzeczywistości porównanie negacji wartości `x` z liczbą całkowitą `1`:

```
if (!x) == 1
```

Najlepszym rozwiązaniem jest stosowanie nawiasów okrągłych w większości przypadków, a nawet za każdym razem podczas tworzenia tego rodzaju konstrukcji. Jeśli nawet nawiasy nie są absolutnie konieczne, ułatwiają twórcy kodu oraz innym osobom jego zrozumienie i w razie potrzeby późniejsze modyfikowanie.

Trzeci sposób wyrażania warunku negującego polega na użyciu słowa kluczowego `unless`.

### SŁOWO KLUCZOWE UNLESS

Słowo kluczowe `unless` zapewnia naturalniej wyglądającą metodę wyrażania tej samej semantyki co w przypadku instrukcji `if not` lub `if !`:

```
unless x == 1
```

Wspomnianą wyżej „naturalność" traktuj jednak z pewną rezerwą. Programy Ruby są tworzone w języku Ruby, a nie w języku angielskim. Należy dążyć do uzyskania dobrego stylu kodu Ruby bez zbytniego przejmowania się tym, jak napisany kod czyta się jako anglojęzyczną prozę. Nie oznacza to, że sporadycznie język angielski nie może pełnić roli przewodnika: na przykład prześledzenie sekwencji `unless/else` dokonującej cofnięcia efektu negowania, która standardowo nie jest powiązana z użyciem słowa *unless*, może okazać się trochę utrudnione:

```
unless x > 100
  puts "Mała liczba!"
else
  puts "Duża liczba!"
end
```

Generalnie sekwencję `if/else` odczytuje się lepiej niż sekwencję `unless/else`. Przez odwrócenie logiki warunku zawsze możesz zastąpić drugą sekwencję pierwszą:

```
if x <= 100
  puts "Mała liczba!"
else
  puts "Duża liczba!"
end
```

Jeśli spotkasz się z przypadkiem, w którym negacja logiki wydaje się bardziej niewygodna niż łączenie ze sobą instrukcji `unless` i `else`, pozostań przy instrukcji `unless`. W przeciwnym razie, gdy występuje klauzula `else`, instrukcja `if` jest zwykle lepszą opcją niż instrukcja `unless`.

Możliwe jest również umieszczenie testów warunkowych w pozycji *modyfikatora* bezpośrednio za instrukcją.

#### MODYFIKATORY WARUNKOWE

Częstą sytuacją jest występowanie modyfikatora warunkowego na końcu instrukcji w przypadku podobnym do następującego:

```
puts "Duża liczba!" if x > 100
```

Odpowiada to następującemu kodowi:

```
if x > 100
  puts "Duża liczba!"
end
```

Możliwe jest też zrealizowanie tego działania za pomocą instrukcji `unless`:

```
puts "Duża liczba!" unless x <= 100
```

Modyfikatory warunkowe cechują się konwersacyjnym stylem działania. W ich przypadku nie trzeba się martwić słowem kluczowym `end`. Co prawda modyfikatory te nie pozwalają na zbyt wiele w przypadku elementu (na przykład brak możliwości tworzenia gałęzi instrukcji `else` lub `elsif`), ale w sytuacji, gdy niezbędny jest prosty kod warunkowy, często modyfikatory okazują się odpowiednie. Próbuj jednak unikać używania naprawdę długich instrukcji, które są zakończone modyfikatorami warunkowymi. Mogą one być trudne do odczytania, a także do zapamiętania w przypadku oczekiwania na końcu wykonywania na modyfikator:

```
puts "done" && return (x > y && a < b) unless c == 0
```

**Potencjalnie wprowadzające zamieszanie dołączanie instrukcji `unless` do już długiego wiersza kodu**

Podobnie jak dla innych instrukcji w języku Ruby dla każdej instrukcji `if` określany jest obiekt. Przyjrzyjmy się, jak to przebiega.

**„Życie" bez niejednoznaczności instrukcji else**

W niektórych językach bez użycia specjalnej reguły nie możesz stwierdzić, jaka klauzula `else` powiązana jest z jaką klauzulą `if`. Na przykład w języku C instrukcja `if` może mieć następującą postać:

```
if (x)
    if (y) { wykonaj ten kod }
    else   { wykonaj ten kod }; ←
```

**Warunek** `x` **jest prawdziwy, ale warunek** `y` **już nie**

Zastanówmy się jednak przez chwilę: czy kod zachowuje się w sposób, na jaki wskazuje wcięcie (instrukcja `else` należy do drugiej instrukcji `if`)? Czy może kod działa w poniższy sposób?

```
if (x)
    if (y){ wykonaj ten kod }
else { wykonaj ten kod }; ←
```

**Warunek** `x` **nie jest prawdziwy**

W powyższym kodzie zmieniło się jedynie wcięcie trzeciego wiersza (nie ma to znaczenia dla kompilatora języka C; wcięcie powoduje tylko, że pod względem wizualnym niejednoznaczność jest oczywista). Do jakiej instrukcji `if` należy instrukcja `else`? W jaki sposób to stwierdzić?

Stwierdzenie tego umożliwia znajomość następującej reguły w języku C: zmieniająca położenie instrukcja `else` powiązana jest z ostatnią niedopasowaną instrukcją `if` (w omawianym przykładzie dotyczy to pierwszego z dwóch wariantów działania kodu). Jednak w języku Ruby z pomocą przychodzi instrukcja `end`:

```
if x > 50
  if x > 100
    puts "Duża liczba"
  else
    puts "Średnia liczba"
  end
end
```

Pojedyncza instrukcja `else` w powyższym kodzie musi należeć do drugiej instrukcji `if`, ponieważ nie natrafiła ona jeszcze na swoją instrukcję `end`. Pierwsza instrukcja `if` i ostatnia instrukcja `end` zawsze są ze sobą powiązane, druga instrukcja `if` i druga od końca instrukcja `end` ciągle tworzą parę itd. Pary instrukcji `if`/`end` „otaczają" to, co do nich należy, w tym instrukcję `else`. Oczywiście oznacza to konieczność poprawnego umiejscowienia słów kluczowych `end`.

## WARTOŚĆ INSTRUKCJI IF

Jeśli wykonanie instrukcji `if` powiedzie się, dla całej instrukcji określane jest cokolwiek, co jest reprezentowane przez kod w gałęzi, dla której wystąpiło powodzenie. Następujący kod wpisz w sesji narzędzia irb, aby przekonać się, jak wygląda to w praktyce:

```
x = 1
if x < 0
  "ujemna"
elsif x > 0
  "dodatnia"
else
  "zero"
end
```

Narzędzie irb poinformuje, że wartością tej całej instrukcji `if` jest łańcuch `"dodatnia"`.

Instrukcja `if`, której działanie w dowolnym miejscu zakończy się niepowodzeniem, zwraca wartość `nil`. Oto kompletny przykład sesji narzędzia irb dla takiej sytuacji:

```
>> x = 1
=> 1
>> if x == 2
>>   "to jest 2!"
>> elsif x == 3
>>   "to jest 3!"
>> end
=> nil
```

**Dla całej instrukcji `if` określana jest wartość `nil`, ponieważ działanie instrukcji kończy się niepowodzeniem**

Instrukcje warunkowe na kilka sposobów prowadzą interakcję z innymi elementami składni języka Ruby, o których trzeba wiedzieć. W szczególności dotyczy to składni przypisań. Warto bardziej szczegółowo przyjrzeć się zachowaniu instrukcji warunkowych w przypisaniach, ponieważ obejmuje to kilka interesujących kwestii związanych ze sposobem analizowania kodu przez interpreter języka Ruby.

### 6.1.2. Składnia przypisania w treści instrukcji warunkowych i testach

Składnia przypisania i wyrażenia warunkowe krzyżują się w dwóch miejscach: w treści wyrażeń warunkowych, gdzie przypisania mogą występować albo wcale ich nie ma, a także w samych testach warunkowych:

```
if x = 1
  y = 2
end
```

**Przypisanie w treści instrukcji warunkowej**

**Przypisanie w teście warunkowym**

Co ma miejsce (albo nie ma) w przypadku użycia tych idiomów? Przyjrzymy się obu, zaczynając od przypisania zmiennej w treści instrukcji warunkowej. Dokładniej rzecz biorąc, będzie to przypisanie zmiennej lokalnej, które w tym kontekście po części przejawia być może nieoczekiwane zachowanie.

#### PRZYPISANIE ZMIENNEJ LOKALNEJ W TREŚCI INSTRUKCJI WARUNKOWEJ

W przypadku języka Ruby — w przeciwieństwie do języków kompilowanych — nie jest tak wyraźnie widoczna granica między „czasem kompilacji" i „czasem wykonywania". Interpreter analizuje jednak kod przed uruchomieniem go. Ponadto określone decyzje są podejmowane w czasie trwania tego procesu. Istotną kwestią jest rozpoznawanie i alokacja zmiennych lokalnych.

Gdy analizator kodu Ruby napotyka sekwencję złożoną z *identyfikatora*, *znaku równości* i *wartości*, tak jak w przypadku następującego wyrażenia:

```
x = 1
```

dokonuje alokacji obszaru dla zmiennej lokalnej o nazwie `x`. Tworzenie zmiennej (nie przypisywanie do niej wartości, lecz wewnętrzne tworzenie zmiennej) zawsze jest wynikiem tego rodzaju wyrażenia, nawet gdy kod nie jest wykonywany!

Przeanalizuj następujący przykład:

```
if false
  x = 1
```

```
end
p x  ←——— Dane wyjściowe: nil
p y  ←——— Błąd krytyczny: y jest nieznane
```

Przypisanie do `x` nie jest wykonywane, ponieważ zostało opakowane w teście warunkowym zakończonym niepowodzeniem. Analizator języka Ruby napotyka jednak sekwencję `x = 1`, na podstawie której określa, że program uwzględnia zmienną lokalną `x`. Analizatora nie interesuje to, czy zmiennej `x` w ogóle przypisano wartość. Jego zadaniem jest po prostu sprawdzenie kodu pod kątem zmiennych lokalnych, dla których niezbędne jest przydzielenie miejsca.

Powoduje to, że zmienna `x` znajduje się w dziwnym rodzaju stanu zawieszenia zmiennych. Zmienna została zainicjowana za pomocą wartości `nil`. Pod tym względem zmienna `x` różni się od zmiennej, która wcale nie istnieje. Jak można zauważyć w przykładzie, sprawdzenie zmiennej `x` pozwala uzyskać wartość `nil`, natomiast próba przeprowadzenia inspekcji dla nieistniejącej zmiennej `y` kończy się błędem krytycznym. Choć zmienna `x` istnieje, nie odegrała w programie żadnej roli. Istnieje ona tylko jako artefakt procesu analizowania.

Nic takiego nie ma miejsca w przypadku zmiennych klasy, zmiennych instancji lub zmiennych globalnych. Wszystkie te typy zmiennych mogą być rozpoznane dzięki ich postaci (`@@x`, `@x`, `$x`). Zmienne lokalne wyglądają jednak tak jak wywołania metod. W języku Ruby niezbędne jest zastosowanie pewnej logiki w czasie analizowania, aby w takim zakresie, jaki jest możliwy, stwierdzić, co jest czym.

Trzeba też uważać podczas używania składni przypisań w części testowania instrukcji warunkowej.

### PRZYPISANIE W TEŚCIE INSTRUKCJI WARUNKOWEJ

W omawianym przykładzie zwróć uwagę na to, że test warunkowy jest przypisaniem (`x = 1`), a nie testem z operatorem równości (miałoby to postać `x==1`):

```
if x = 1
  puts "Witaj!"
end
```

Przypisanie to działa tak jak przypisania w ogólności: zmienna `x` ustawiana jest na wartość `1`. A zatem test przyjmuje skróconą postać instrukcji warunkowej `if 1`, która jest prawdziwa. Oznacza to, że wykonywana jest zawartość instrukcji warunkowej i wyświetlany łańcuch `"Witaj!"`.

Wyświetlane jest jednak również następujące ostrzeżenie:

```
warning: found = in conditional, should be ==
```

W takim przypadku „myślenie” interpretera języka Ruby jest następujące: wyrażenie testowe `if x = 1` zawsze się powiedzie, a treść instrukcji warunkowej będzie każdorazowo wykonywana. Oznacza to, że programista nie ma żadnego możliwego do wyobrażenia powodu, aby w ogóle wpisywać kod `if x = 1`. A zatem interpreter „wnioskuje”, że programista prawie na pewno miał zamiar wpisać coś jeszcze, dlatego wyświetla ostrzeżenie informujące o możliwej pomyłce. Dokładniej rzecz biorąc, ostrzeżenie sugeruje operator ==, który generuje rzeczywisty test (czyli test, który niekoniecznie zawsze jest prawdziwy).

Szczególnie przydatne w przypadku takiego mechanizmu ostrzegania jest to, że interpreter języka Ruby jest na tyle „inteligentny", że nie będzie ostrzegał w sytuacjach, w których nie ma pewności, że warunek okaże się prawdziwy. Jeśli prawa strona przypisania sama jest zmienną lub wywołaniem metody, nie zostanie wyświetlone ostrzeżenie:

```
if x = y    ←——— Brak ostrzeżenia
```

W przeciwieństwie do kodu `x = 1` wyrażenie przypisania `x = y` niekoniecznie może zakończyć się powodzeniem jako test warunkowy (niepowodzenie wystąpi, gdy wartość `y` będzie fałszywa). Oznacza to, że dla interpretera języka Ruby przekonujące jest to, że testowane będzie takie wyrażenie, dlatego nie wyświetla ostrzeżenia.

Dlaczego miałbyś użyć przypisania w teście warunkowym? Z pewnością nigdy nie będzie to konieczne. Zawsze możesz zastosować następujący kod:

```
x = y
if x
# itd.
```

Czasami warto jednocześnie zastosować przypisanie i testowanie, a zwłaszcza w przypadku korzystania z metody zwracającej wartość `nil` dla niepowodzenia oraz inną wartość dla powodzenia. Typowym przykładem jest dopasowywanie wzorców za pomocą metody `match`. Metoda ta, która znacznie obszerniej zostanie zaprezentowana w rozdziale 11., testuje łańcuch przy użyciu wyrażenia regularnego, zwracając wartość `nil`, gdy nie występuje dopasowanie, natomiast instancję obiektu `MatchData` w przypadku określenia dopasowania. Obiekt `MatchData` może być odpytywany w celu uzyskania informacji o szczegółach dopasowania. Zwróć uwagę na użycie wyrażenia regularnego literału `/ow/` w trakcie testowania dopasowania dla łańcucha `name`:

```
name = "Dawid K. Nowak"
if m = /ow/.match(name)      ←——— ❶
  puts "Znaleziono dopasowanie!"
  print "Oto niedopasowany początek łańcucha: "
  puts m.pre_match
  print "Oto niedopasowany koniec łańcucha: "
  puts m.post_match
else
  puts "Brak dopasowania"
end
```

Oto dane wyjściowe tego fragmentu kodu:

```
Znaleziono dopasowanie!
Oto niedopasowany początek łańcucha: Dawid K. N
Oto niedopasowany koniec łańcucha: ak
```

Metoda dopasowująca szuka wzorca `ow` w łańcuchu `"Dawid K. Nowak"`. Zmienna `m` jest przypisywana w instrukcji warunkowej ❶. W przypadku braku dopasowania zmienna będzie mieć wartość `nil`. Oczywiście przykład ten jest specjalnie zaaranżowany: występuje dopasowanie, dlatego zmienna `m` jest obiektem `MatchData` i może być odpytywana. W przykładzie od obiektu żądane są informacje o częściach łańcucha, które wystąpiły przed i po jego dopasowanej części. Obiekt zwraca odpowiednie podłańcuchy.

Jak zawsze możesz przebudować przypisanie i test warunkowy w następujący sposób:

```
m = /ow/.match(name)
if m
  # itd.
```

Nie ma potrzeby łączenia tych elementów w jedno wyrażenie. Jednak przynajmniej w tym przypadku pojawia się pewien aspekt semantyczny, który przemawia za dokonaniem połączenia: wyrażenie niekoniecznie może przejść pomyślnie test warunkowy, dlatego rozsądnym jest sprawdzenie wyrażenia.

Choć instrukcja `if` i powiązane z nią elementy stanowią typowe słowa kluczowe operacji warunkowych, nie są jedynymi. W dalszej kolejności przyjrzymy się instrukcjom `case`.

### 6.1.3. Instrukcje case

Instrukcja `case` rozpoczyna od wyrażenia (choć zazwyczaj jest to pojedynczy obiekt lub zmienna, może zostać użyte dowolne wyrażenie) i kolejno sprawdza listę możliwych dopasowań. Każde możliwe dopasowanie znajduje się w instrukcji `when`, która składa się z co najmniej jednego możliwego obiektu dopasowującego oraz segmentu kodu. Gdy wystąpi dopasowanie jednego z terminów w danej klauzuli `when`, jest ona uważana za tę, która zawiera „wygraną". Z tego powodu wykonywany jest jej segment kodu. „Wygraną" może być co najwyżej jedno dopasowanie.

Do opanowania instrukcji `case` lepszy będzie przykład niż opis. W listingu 6.1 zaprezentowano taką instrukcję, która testuje wiersz danych wprowadzonych za pomocą klawiatury i tworzy gałąź na podstawie wartości tych danych.

**Listing 6.1. Interpretowanie danych wprowadzonych przez użytkownika za pomocą instrukcji case**

```
print "Czy zakończyć program? (tak lub nie): "
answer = gets.chomp   ← Usuwa końcowy znak nowego wiersza z łańcucha wejściowego
case answer           ← ❶
when "tak"            ← ❷
  puts "Żegnaj!"
  exit
when "nie"
  puts "W takim razie będziemy kontynuować"
else                  ← ❸
  puts "Nieznana odpowiedź -- przy założeniu, że wybrano opcję 'nie'"
end                   ← ❹
puts "Kontynuowanie działania programu...."
```

Instrukcja `case` zaczyna się od słowa kluczowego `case` ❶, obejmuje dalej wszystkie bloki klauzuli `when` ❷ i opcjonalną klauzulę `else` ❸, a następnie kończy się słowem kluczowym `end` ❹. Co najwyżej jedno dopasowanie będzie pomyślne, co spowoduje wykonanie jego kodu. Jeśli dopasowanie to będzie należeć do bloku wartości `"tak"`, program zakończy działanie. Dowolne inne dane wejściowe oznaczają użycie bloku

wartości "nie" lub jakiejś innej wartości, która przez ten konkretny program jest interpretowana jako odpowiednik bloku wartości "nie", powodując kontynuowanie działania programu.

Możliwe jest umieszczenie w pojedynczym bloku klauzuli when więcej niż jednego możliwego dopasowania. Prezentuje to następujący fragment kodu:

```
case answer
when "t", "tak"
  puts "Żegnaj!"
  exit
   # itd.
```

Przecinek między wieloma warunkami w klauzuli when to swego rodzaju operator „lub". W przypadku powyższego kodu zostanie wyświetlony łańcuch Żegnaj!, po czym nastąpi zakończenie działania, gdy wartością answer będzie "t" lub "tak".

Przyjrzyjmy się wewnętrznemu mechanizmowi działania klauzuli when. Nie będziesz zaskoczony stwierdzeniem, że uwzględnia on wysyłanie komunikatów.

### ZASADY DZIAŁANIA KLAUZULI WHEN

Podstawową ideą działania struktury case/when jest to, że po pobraniu obiektu kolejno wykonywane są testy w celu uzyskania dopasowania, w przypadku którego działanie jest podejmowane na podstawie pomyślnie zakończonego testu. Czym jednak w tym kontekście jest *dopasowanie*? Co w omawianym przykładzie oznacza stwierdzenie, że answer dopasowuje słowo *tak*, słowo *nie* albo żadne?

W odniesieniu do instrukcji when w języku Ruby występuje konkretna definicja *dopasowania*.

Każdy obiekt języka Ruby zawiera metodę *równości przypadków* o nazwie === (trzy znaki równości, które czasami są nazywane operatorem *potrójnego znaku równości*). Wynikiem wywołania metody === jest określenie, czy dla klauzuli when wystąpiło dopasowanie.

Możesz to zauważyć wyraźnie po przyjrzeniu się najpierw instrukcji case, a następnie jej translacji do postaci kodu z trzema znakami równości. Spójrz ponownie na instrukcję case z listingu 6.1. Oto ten sam kod przebudowany w celu jawnego zastosowania operatora potrójnego znaku równości:

```
if "tak" === answer
  puts "Żegnaj!"
  exit
elsif "nie" === answer
  puts "W takim razie będziemy kontynuować"
else
  puts "Nieznana odpowiedź -- przy założeniu, że wybrano opcję 'nie'"
end
```

Metoda === w położeniu operatora wrostkowego (czyli między terminami po lewej i prawej stronie) stanowi prawdziwy „lukier" składniowy w przypadku wywołania metody:

```
if "tak".===(answer)
```

Instrukcja `when` opakowuje to wywołanie metody w jeszcze jednym „lukrze” składniowym: nie musisz jawnie używać metody `===` ani w położeniu operatora, ani metody. Jest to realizowane automatycznie.

Taka jest logika składni. Dlaczego jednak kod:

```
"tak" === answer
```

Zwraca wartość `true`, gdy `answer` zawiera `"tak"`?

Wywołanie metody zwraca wartość `true` z powodu sposobu definiowania dla łańcuchów metody trzech znaków równości. Gdy zażądasz od łańcucha użycia we własnym zakresie tej metody względem innego łańcucha (`string1 === string2`), oznacza to, że oczekujesz od łańcucha porównania swojej zawartości znak po znaku z drugim łańcuchem, a następnie przekazania wartości `true` w przypadku idealnego dopasowania lub wartości `false` w przeciwnym razie.

W powyższym objaśnieniu najważniejszy jest fragment „dla łańcuchów”. Każda klasa (a w teorii każdy pojedynczy obiekt, choć zwykle jest on obsługiwany na poziomie klasy) może zdefiniować własną metodę `===`, a tym samym własną logikę równości przypadków. W przypadku łańcuchów, a w rzeczywistości dowolnego obiektu, który nie przesłania metody `===`, działa ona tak samo jak metoda `==` (podstawowa metoda testująca, która sprawdza, czy jeden łańcuch jest równy drugiemu łańcuchowi). Inne klasy mogą jednak definiować test z metodą `===` w dowolny żądany sposób.

A zatem logika instrukcji `case/when` to tak naprawdę logika `obiekt === inny_obiekt` „w przebraniu”. Z kolei `obiekt === inny_obiekt` to w rzeczywistości kod `obiekt. === (inny_obiekt)` „w przebraniu”. Jednak definiując metodę trzech znaków równości, oczekujesz od własnych klas, że zapewnią możliwość pełnej kontroli tego, jak obiekty zachowują się w obrębie instrukcji `case`.

### „PROGRAMOWANIE ZACHOWANIA” INSTRUKCJI CASE OBIEKTÓW

Załóżmy, że decydujemy, iż obiekt `Ticket` powinien dopasowywać klauzulę `when` w instrukcji `case` na podstawie miejsca wydarzenia. W tym celu można użyć odpowiedniej metody `===`. Listing 6.2 prezentuje taką metodę, która jest wyposażona w wystarczającą funkcjonalność związaną z biletami, aby zapewnić kompletny, działający przykład.

**Listing 6.2. Implementowanie zachowania instrukcji case dla klasy Ticket**

```
class Ticket
  attr_accessor :venue, :date
  def initialize(venue, date)
    self.venue = venue
    self.date = date
  end
  def ===(other_ticket)          ←——❶
    self.venue == other_ticket.venue
  end
end
ticket1 = Ticket.new("Sala Miejska", "07/08/13")
ticket2 = Ticket.new("Centrum Konferencyjne", "07/08/13")
ticket3 = Ticket.new("Sala Miejska", "08/09/13")
```

```
puts "Bilet ticket1 dotyczy wydarzenia w miejscu: #{ticket1.venue}."
case ticket1
when ticket2                                        ←——❷
  puts "To samo miejsce co w przypadku biletu ticket2!"
when ticket3                                        ←——❸
  puts "To samo miejsce co w przypadku biletu ticket3!"
else
  puts "Brak dopasowania"
end
```

Dane wyjściowe kodu z tego listingu są następujące:

```
Bilet ticket1 dotyczy wydarzenia w miejscu: Sala Miejska.
To samo miejsce co w przypadku biletu ticket3!
```

Dopasowanie określono przez niejawne użycie metody instancji `===` klasy `Ticket` ❶. Wewnątrz instrukcji `case` pierwsze wyrażenie klauzuli `when` ❷ wyzwala ukryte wywołanie metody `===`, które odpowiada zastosowaniu następującego kodu:

```
if ticket2 === ticket1
```

Ponieważ metoda `===` zwraca wartość `true` lub `false` na podstawie porównania miejsc, a miejsce powiązane z obiektem `ticket2` nie jest tym samym co w przypadku obiektu `ticket1`, porównanie tych dwóch obiektów biletów powoduje zwrócenie wartości `false`. A zatem nie jest wykonywany kod, który stanowi zawartość odpowiedniej klauzuli `when`.

Dalej przeprowadzany jest następny test, czyli kolejne porównanie obiektów `ticket1` i `ticket3` za pomocą metody `===` ❸. Test ten zwraca wartość `true`. Wyrażenie klauzuli `when` kończy się powodzeniem i wykonywany jest zawarty w niej kod.

Tego rodzaju wewnętrzny przepływ między definicjami metody (`===`) i kodem, który nie przypomina wywoływania metod (`case/when`), jest typowy w języku Ruby. Struktura `case/when` zapewnia elegancki sposób przeprowadzania kaskadowych testów warunkowych. Ponadto fakt użycia grupy wywołań metody `===` oznacza, że definiując tę metodę we własnych klasach, za pomocą takiej grupy możesz osiągnąć to, co jest wymagane.

Instrukcja `case` występuje też w trochę skróconej postaci, która umożliwia bezpośrednie testowanie wartości prawdziwości warunku. Mowa o instrukcji `case` pozbawionej jej wyrażenia.

### PROSTY TEST PRAWDZIWOŚCI ZA POMOCĄ INSTRUKCJI CASE

Jeśli rozpoczniesz od instrukcji `case` w postaci tylko słowa kluczowego `case` (czyli bez wyrażenia testowego) z następującymi po niej kilkoma klauzulami `when`, „zwycięzcą" będzie pierwsza z nich, której warunek jest prawdziwy. W przypadku obiektu `user` z metodami `first_name` i `last_name` można sobie wyobrazić utworzenie następującej instrukcji `case`:

```
case
when user.first_name == "Dawid", user.last_name == "Nowak"
  puts "Możesz być Dawidem Nowakiem."
when Time.now.wday == 5        ←——— Liczba porządkowa dnia tygodnia (niedzieli odpowiada 0)
  puts "Nie jesteś Dawidem Nowakiem, ale przynajmniej jest piątek!"
```

```
else
  puts "Nie jesteś Dawidem Nowakiem, a poza tym nie ma piątku."
end
```

W tym przypadku zwykłe słowo kluczowe `case` stanowi alternatywny sposób zapisu instrukcji `if`. Okazuje się, że dowolna instrukcja `case` może zostać zapisana jako instrukcja `if`. Instrukcje `case` z jawnie określonymi argumentami często są znacznie krótsze od swych odpowiedników w postaci instrukcji `if`, które muszą uciekać się do wywoływania metody `===` lub innych metod porównujących. Instrukcje `case` bez jawnych argumentów testowych są zwykle krótsze niż ich odpowiedniki w postaci instrukcji `if`. Na przykład po zastosowaniu instrukcji `if` w poprzednim przykładzie zostałby uzyskany następujący kod:

```
if user.first_name == "Dawid" or user.last_name == "Nowak"
  puts "Możesz być Dawidem Nowakiem."
elsif Time.now.wday == 5
  puts "Nie jesteś Dawidem Nowakiem, ale przynajmniej jest piątek!"
else
  puts "Nie jesteś Dawidem Nowakiem, a poza tym nie ma piątku."
end
```

Zaletą instrukcji `case` bez argumentów testowych jest to, że nie ogranicza tego, co możesz testować za pomocą metody `===` dla danego obiektu. Gdy masz wybór między instrukcją `case` bez argumentów testowych i instrukcją `if`, należy kierować się tym, co uważasz, że będzie najlepiej wyglądać i zapewnia najlepsze możliwości odczytu.

### WARTOŚĆ ZWRACANA INSTRUKCJI CASE

Ostatnia istotna kwestia związana z instrukcjami `case`, o której trzeba pamiętać, dotyczy tego, że dla każdej takiej instrukcji określany jest pojedynczy obiekt. W przypadku wystąpienia klauzuli `when` lub `else` zakończonej powodzeniem wartość zwracana całej instrukcji `case` jest wartością zwracaną przez kod w tej klauzuli. W przeciwnym razie, gdy wyszukiwanie dopasowania przez instrukcję `case` zakończy się niepowodzeniem, dla całej instrukcji zostanie określona wartość `nil` podobnie jak w przypadku instrukcji `if` zakończonych niepowodzeniem.

A zatem możesz na przykład w następujący sposób przebudować kod, w którym nie określono warunków:

```
puts case
     when user.first_name == "Dawid", user.last_name == "Nowak"
       "Możesz być Dawidem Nowakiem."
     when Time.now.wday == 5
       "Nie jesteś Dawidem Nowakiem, ale przynajmniej jest piątek!"
     else
       "Nie jesteś Dawidem Nowakiem, a poza tym nie ma piątku."
     end
```

W tej wersji wywołania metody `puts` zostały usunięte poza klauzule `when`. Cała instrukcja `case` ma jedynie na celu znalezienie argumentu pojedynczego wywołania metody `puts` po lewej stronie instrukcji. Argumentem tym będzie dowolny z trzech łańcuchów zwracanych przez instrukcję `case`.

Instrukcje warunkowe, takie jak `if` i `case/when`, umożliwiają sterowanie przepływem programu, co polega na realizowaniu jednego działania zamiast innego. Czasami jednak konieczne jest wielokrotne wykonywanie pojedynczego zadania. Tego rodzaju powtarzanie może zostać osiągnięte za pomocą pętli, którym przyjrzymy się w następnym podrozdziale.

## 6.2. Powtarzanie działań za pomocą pętli

Funkcje języka Ruby służące do powtarzalnego wykonywania kodu w ramach pętli pozwalają też na zastosowanie logiki warunkowej: pętla może być wykonywana, *gdy* dany warunek jest prawdziwy (np. zmienna równa danej wartości). Ponadto pętla może być wykonywana do momentu, *aż* dany warunek będzie prawdziwy. Możliwe jest również *bezwarunkowe* wyjście z pętli, co powoduje zakończenie jej w określonym miejscu i wznowienie wykonywania kodu programu za pętlą.

Przyjrzymy się kilku krokom wykonywania pętli, począwszy, i słusznie, od metody o nazwie `loop`.

### 6.2.1. Bezwarunkowe wykonywanie pętli za pomocą metody loop

Metoda `loop` nie pobiera żadnych standardowych argumentów. Po prostu jest ona wywoływana. Metoda pobiera jednak blok kodu, czyli ustalony zestaw instrukcji programu utworzonych jako część wywołania metody `loop` i dostępnych do wykonania *z* jej poziomu (w dalszej części rozdziału znacznie dokładniej przyjrzymy się blokom kodu; w tym miejscu wystarczająca będzie tylko ogólna informacja). A zatem składnia wywołania metody `loop` wygląda następująco:

```
loop blok_kodu
```

Bloki kodu mogą być tworzone na jeden z dwóch sposobów: w nawiasach klamrowych (`{}`) lub za pomocą słów kluczowych `do` i `end`. Następujące dwa fragmenty kodu są równoznaczne:

```
loop { puts "Wykonywanie pętli bez końca!" }
loop do
  puts "Wykonywanie pętli bez końca!"
end
```

Luźna konwencja głosi, że jednowierszowe bloki kodu używają nawiasów klamrowych, a bloki wielowierszowe korzystają ze słów kluczowych `do` i `end`. W języku Ruby nie jest jednak wymuszana taka konwencja (okazuje się, że nawiasy klamrowe oraz te słowa kluczowe różnią się nieznacznie od siebie pod względem pierwszeństwa; nie musisz się teraz tym przejmować).

Ogólnie rzecz biorąc, niewskazane jest nieskończone wykonywanie pętli. Wymagane będzie zatrzymanie jej w pewnym momencie. Choć zwykle używana jest w tym celu kombinacja klawiszy *Ctrl+C*, istnieją też inne, bardziej programistyczne rozwiązania.

#### STEROWANIE PĘTLĄ

Jeden ze sposobów zatrzymania pętli polega na użyciu słowa kluczowego `break`, tak jak ma to miejsce w następującym, trzeba przyznać, że rozbudowanym, kodzie ustawiającym dla zmiennej `n` wartość 10:

```
n = 1
loop do
  n = n + 1
  break if n > 9
end
```

Inna metoda sprowadza się do przejścia do następnej iteracji pętli bez kończenia bieżącej iteracji. W tym celu użyj słowa kluczowego `next`:

```
n = 1
loop do
  n = n + 1
  next unless n == 10
  break
end
```

W tym przypadku sterowanie przechodzi do instrukcji `break` tylko wtedy, gdy równanie `n == 10` jest prawdziwe. Jeśli równanie to *nie* jest prawdziwe (`unless n == 10`), wykonywana jest instrukcja `next`, a sterowanie przechodzi z powrotem na początek pętli, zanim osiągnie ona instrukcję `break`.

Możesz też wykonywać pętlę w sposób warunkowy: *gdy* dany warunek jest prawdziwy lub do momentu, *aż* warunek stanie się prawdziwy.

### 6.2.2. Warunkowe wykonywanie pętli za pomocą słów kluczowych while i until

Warunkowe wykonywanie pętli jest możliwe dzięki słowom kluczowym `while` i `until`. Mogą one być używane na kilka sposobów, zależnie od tego, jak dokładnie ma przebiegać wykonywanie pętli.

#### SŁOWO KLUCZOWE WHILE

Słowo kluczowe `while` umożliwia uruchomienie pętli, gdy dany warunek jest prawdziwy. Blok rozpoczynający się tym słowem kluczowym musi zostać zakończony słowem kluczowym `end`. Kod znajdujący się między tymi dwoma słowami stanowi zawartość pętli `while`. Oto przykład:

```
n = 1
while n < 11
  puts n
  n = n + 1
end
puts "Gotowe!"
```

Kod wyświetla następujące dane:

```
1
2
3
4
```

```
5
6
7
8
9
10
Gotowe!
```

Dopóki warunek `n < 11` jest prawdziwy, pętla jest wykonywana. Z każdą iteracją pętli wartość zmiennej `n` jest zwiększana o 1. Gdy przy jedenastym sprawdzeniu warunku okazuje się on fałszywy (`n` nie jest już mniejsze niż 11), następuje zakończenie wykonywania pętli.

Możliwe jest też umieszczenie słowa kluczowego `while` na końcu pętli. W tym przypadku musisz użyć pary słów kluczowych `begin` i `end`, aby oznaczyć miejsce występowania pętli (w przeciwnym razie interpreter języka Ruby nie będzie w stanie stwierdzić, ile wierszy zlokalizowanych przed słowem `while` ma zostać uwzględnionych w pętli).

```
n = 1
begin
  puts n
  n = n + 1
end while n < 11
puts "Gotowe!"
```

Dane wyjściowe tego przykładowego kodu są takie same jak dane wyjściowe z poprzedniego przykładu.

Istnieje różnica między umieszczeniem słowa kluczowego `while` na początku pętli i na jej końcu. Jeśli słowo to wstawisz na początku pętli, a jego warunek jest fałszywy, kod nie jest wykonywany:

```
n = 10
while n < 10
  puts n
end
```

Ponieważ zmienna `n` ma wartość większą niż 10 w momencie wykonywania testu warunku `n < 10` po raz pierwszy, zawartość instrukcji nie jest wykonywana. Jeśli jednak słowo kluczowe `while` zostanie umieszczone na końcu pętli:

```
n = 10
begin
  puts n
end while n < 10
```

zostanie wyświetlona liczba 10. Oczywiście zmienna `n` w żadnym momencie nie jest mniejsza niż 10. Ponieważ jednak test warunku słowa kluczowego `while` znajduje się na końcu instrukcji, jej zawartość jest wykonywana jednokrotnie przed wykonaniem testu.

Podobnie jak instrukcje `if` i `unless`, również słowa kluczowe `while` i `until` pętli warunkowej tworzą parę.

### SŁOWO KLUCZOWE UNTIL

Słowo kluczowe `until` jest używane w ten sam sposób co słowo kluczowe `while`, ale bazując na odwrotnej logice. Oto kolejny wymagający sporego nakładu pracy sposób wyświetlania liczb całkowitych od 1 do 10, który tym razem ilustruje użycie słowa kluczowego `until`:

```
n = 1
until n > 10
  puts n
  n = n + 1
end
```

Zawartość pętli (w przykładzie odpowiada za wyświetlanie i inkrementowanie wartości zmiennej `n`) jest wielokrotnie wykonywana do momentu, aż warunek będzie prawdziwy.

Możliwe jest też zastosowanie słowa kluczowego `until` za blokiem kodu w połączeniu z parą słów kluczowych `begin/end`. Podobnie jak w przypadku słowa kluczowego `while` blok kodu zostanie wykonany raz przed sprawdzeniem warunku słowa kluczowego `until`.

Tak jak „kuzyni", czyli słowa kluczowe `if` i `unless`, słowa kluczowe `while` i `until` mogą być używane na pozycji modyfikatora w jednowierszowych instrukcjach.

### MODYFIKATORY WHILE I UNTIL

Oto nieznacznie krótszy sposób liczenia do dziesięciu za pomocą słowa kluczowego `until` umieszczonego na pozycji modyfikatora:

```
n = 1
n = n + 1 until n == 10
puts "Osiągnięto liczbę 10!"
```

W miejsce instrukcji `until` możesz również zastosować instrukcję `while n < 10`.

Zauważ, że wersje słów kluczowych `while` i `until` w postaci jednowierszowych modyfikatorów nie zachowują się w taki sam sposób jak te słowa kluczowe umiejscowione za blokiem kodu ze słowami kluczowymi `begin/end`. Inaczej mówiąc, w następującym przypadku:

```
a = 1
a += 1 until true
```

zmienna `a` nadal będzie mieć wartość 1. Instrukcja `a += 1` nie będzie wykonywana, ponieważ wartość `true` jest już prawdziwa. Jednak w następującym przypadku:

```
a = 1
begin
  a += 1
end until true
```

zawartość bloku ze słowami kluczowymi `begin/end` zostanie wykonana raz.

Oprócz bezwarunkowego (metoda `loop`) i warunkowego (instrukcje `while` i `until`) wykonywania pętli operację tę możesz zrealizować dla listy wartości, uruchamiając pętlę jednokrotnie dla każdej wartości. W języku Ruby oferowanych jest kilka sposobów wykonania tego zadania, z których jeden bazuje na słowie kluczowym `for`.

### 6.2.3. Wykonywanie pętli na bazie listy wartości

Załóżmy, że chcesz wyświetlić tabelę wartości stopni Fahrenheita odpowiadających stopniom Celsjusza. W tym celu możesz umieścić wartości stopni Celsjusza w tablicy, a następnie wykonać pętlę dla tablicy, korzystając z pary słów kluczowych `for` i `in`. Pętla jest uruchamiana jeden raz dla każdej wartości tablicy. Każdorazowo wartość jest przypisywana do określonej zmiennej:

```
celsius = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
puts "Celsjusza\tFahrenheita"  ←——— Nagłówek tabeli (\t powoduje wyświetlenie znaku tabulacji)
for c in celsius
  puts "#{c}\t#{Temperature.c2f(c)}"
end
```

Zawartość pętli (instrukcja `puts`) wykonywana jest 11 razy. Przy pierwszym wykonaniu wartość zmiennej `c` wynosi `0`. Przy drugim uruchomieniu pętli wartość tej zmiennej to `10`, przy trzecim wartość równa się `20` itd.

Instrukcja `for` to narzędzie o dużych możliwościach. Jednak dość dziwne jest to, że po bliższym przyjrzeniu się tej instrukcji okazuje się, że stanowi ona jedynie alternatywny sposób realizowania czegoś, co zapewnia jeszcze większe możliwości.

## 6.3. Iteratory i bloki kodu

Omówione dotychczas techniki przepływu sterowania uwzględniają kontrolowanie liczby wykonań segmentu kodu lub warunków, w jakich ma to miejsce. W podrozdziale przyjrzymy się innego rodzaju rozwiązaniu powiązanemu z przepływem sterowania. Techniki, które zostaną tutaj zaprezentowane, nie ograniczają się tylko do zrealizowania dla segmentu kodu operacji wykonywania lub pomijania. W ramach *iteracji* techniki te przekazują na zmianę sterowanie programem dwóm zasięgom.

### 6.3.1. Elementy iteracji

Skupienie się na działaniach między zasięgami lokalnymi może wyglądać na powrót do rozważań dotyczących wywołań metod. Przecież gdy dla obiektu wywoływana jest metoda, sterowanie jest przekazywane jej zawartości (inny zasięg). Z kolei po zakończeniu działania metody sterowanie powraca do miejsca tuż za miejscem, w którym wystąpiło wywołanie metody.

Naprawdę ponownie „znajdujemy się na terytorium" wywołań metod, ale poznajemy jego nowe aspekty, a nie tylko jeszcze raz prezentujemy znane już zagadnienia. Omawiamy nową konstrukcję o nazwie *blok kodu* oraz słowo kluczowe `yield`.

W punkcie 6.2.1 zaprezentowano przykładowy kod o następującej postaci:

```
loop { puts "Wykonywanie pętli bez końca!" }
```

Słowo `loop` oraz komunikat w łańcuchu wskazują, co uzyskasz po uruchomieniu kodu: komunikat ten będzie bez końca wyświetlany. Co *dokładnie* jednak ma miejsce? Dlaczego w ogóle ta instrukcja `puts` jest wykonywana, a także odbywa się to w pętli?

Odpowiedź brzmi: metoda `loop` jest *iteratorem*. Iterator to metoda języka Ruby, która w swojej składni wywołania zawiera dodatkowy element. Metoda ta oczekuje

udostępnienia jej bloku kodu. Nawiasy klamrowe w powyższym przykładzie pętli ograniczają blok. Kod w bloku jest złożony z instrukcji `puts`.

Metoda `loop` ma dostęp do kodu wewnątrz bloku: metoda może *wywołać* (wykonać) blok. Aby to zrealizować z poziomu własnego iteratora, używasz słowa kluczowego `yield`. Blok kodu (zapewniany przez kod wywołujący) i słowo kluczowe `yield` (wywoływane w obrębie metody) razem stanowią główne elementy iteracji.

Sama metoda `loop` została utworzona w języku C (korzysta z jego funkcji w celu osiągnięcia tego samego efektu co w przypadku słowa kluczowego `yield`). Jednak całe zagadnienie wykonywania pętli sugeruje interesującą operację, czyli ponowne implementowanie metody `loop` w czystym kodzie Ruby. Operacja ta pozwoli po raz pierwszy ogólnie zorientować się w działaniu słowa kluczowego `yield`.

### 6.3.2. Iteracja w zwykłym stylu

Zadaniem metody `loop` jest ciągłe przekazywanie sterowania blokowi kodu. Oto przykład tego, jak możesz utworzyć własną wersję tej metody:

```
def my_loop
  while true
    yield
  end
end
```

Możliwa jest jeszcze krótsza wersja:

```
def my_loop
  yield while true
end
```

Metodę tę należałoby wywołać tak jak metodę `loop`:

```
my_loop { puts "Wykonywanie mojej pętli bez końca!" }
```

Wynikowy komunikat powinien być bez końca wyświetlany.

Udostępniając blok kodu, metodzie `my_loop` zapewniasz coś (porcję kodu), czemu może ona przekazać sterowanie. Gdy metoda przekazuje sterowanie blokowi, zawarty w nim kod jest uruchamiany, po czym sterowanie powraca do metody. Przekazywanie nie jest tym samym co powrót z metody. Przekazywanie ma miejsce, gdy metoda w dalszym ciągu działa. Po wykonaniu bloku kodu sterowanie powraca do metody dla instrukcji następującej bezpośrednio po wywołaniu słowa kluczowego `yield`.

Blok kodu jest częścią wywołania metody, czyli jej składni. Jest to istotna uwaga: blok kodu nie jest argumentem. Argumenty metody to argumenty, a blok kodu to blok. Są to dwie osobne konstrukcje. Logikę związaną z tym rozróżnieniem zrozumiesz po ogólnym zaznajomieniu się z tym, jak wywołania metod są ze sobą łączone.

### 6.3.3. Anatomia wywołania metody

Każde wywołanie metody w języku Ruby ma następującą składnię:

- Obiekt lub zmienna odbiorcy (jeśli nie zostanie określony, domyślnie jest to obiekt `self`).

- Znak kropki (wymagany w przypadku występowania jawnego odbiorcy; w przeciwnym razie znak jest niedozwolony).
- Nazwa metody (wymagane).
- Lista argumentów (opcjonalne; domyślnie jest to łańcuch `()`).
- Blok kodu (opcjonalne; brak domyślnego bloku).

W szczególności zauważ, że lista argumentów i blok kodu są osobnymi elementami. Ich obecność zmienia się w sposób niezależny. Pod względem składniowym wszystkie następujące wywołania metody są poprawne w języku Ruby:

```
loop { puts "Witaj" }
loop() { puts "Witaj" }
string.scan(/[^,]+/)
string.scan(/[^,]+/) {|word| puts word }
```

W ostatnim wierszu tego przykładu pokazano parametr bloku `word`. Wkrótce powrócimy do parametrów bloku. Różnica między wywołaniem metody z blokiem i bez niego sprowadza się do tego, czy metoda może przekazać sterowanie. Jeśli istnieje blok, jest to możliwe. W przeciwnym razie nie będzie takiej możliwości, ponieważ metoda nie ma czemu przekazać sterowania.

Co więcej, niektóre metody są tak tworzone, aby przynajmniej wykonały *jakąś operację*, niezależnie od tego, czy przekazano im blok kodu, czy nie. Na przykład kod `String#split` powoduje podział swojego odbiorcy (oczywiście jest nim łańcuch) w miejscu występowania przekazanego separatora i zwrócenie tablicy elementów podziału. Jeśli metodzie `split` udostępnisz blok, przekaże ona też elementy podziału (po jednym na raz) do bloku. Blok może następnie wykonać dla każdego podłańcucha dowolne żądane działanie: wyświetlenie go, umieszczenie w kolumnie bazy danych itp.

Jeśli nauczysz się traktować blok kodu jak element składniowy wywołania metody, a nie jak jeden z argumentów, będziesz w stanie kontrolować wszystko, gdy zauważysz więcej wariantów w podstawowej formie iteracji.

Wcześniej wspomniano pobieżnie, że bloki kodu mogą być ograniczane za pomocą nawiasów klamrowych lub pary słów kluczowych `do` i `end`. Przyjrzymy się dokładniej temu, czym różnią się między sobą te dwie opcje ograniczania.

### 6.3.4. Porównanie nawiasów klamrowych oraz pary słów kluczowych do i end w składni bloku kodu

Różnica między dwoma sposobami ograniczania bloku kodu jest różnicą dotyczącą pierwszeństwa. Przyjrzyj się następującemu przykładowi, aby zrozumieć, na czym to polega:

```
>> array = [1,2,3]
=> [1, 2, 3]
>> array.map {|n| n * 10 }        ←— ❶
=> [10, 20, 30]
>> array.map do |n| n * 10 end    ←— ❷
=> [10, 20, 30]
>> puts array.map {|n| n * 10 }   ←— ❸
10
20
```

```
30
=> nil
>> puts array.map do |n| n * 10 end          ←——❹
   #<Enumerator:0x00000101132048>
=> nil
```

Metoda `map` przetwarza jeden element tablicy na raz, wywołując jeden raz blok kodu dla każdego elementu i tworząc nową tablicę, która jest złożona z wyników wszystkich tych wywołań bloku. Odwzorowanie tablicy `[1,2,3]` za pomocą bloku, który mnoży każdy element przez 10, generuje nową tablicę `[10,20,30]`. Co więcej, w przypadku prostej operacji odwzorowania nie ma znaczenia to, czy używane są nawiasy klamrowe ❶, czy słowa kluczowe `do/end` ❷. Uzyskiwane wyniki są identyczne.

Przyjrzyj się jednak temu, co ma miejsce w przypadku zastosowania wyniku operacji odwzorowania jako argumentu metody `puts`. Wersja z nawiasami klamrowymi powoduje wyświetlenie tablicy `[10,20,30]` (po jednym elemencie na wiersz, aby dopasować się do sposobu obsługi tablic przez metodę `puts`) ❸. Wersja ze słowami kluczowymi `do/end` powoduje jednak zwrócenie enumeratora, który jest dokładnie tym, co metoda `map` tworzy, gdy zostanie wywołana *bez* bloku kodu ❹. W rozdziale 10. zamieszczono więcej informacji o enumeratorach. W tym miejscu warto wspomnieć o tym, że dwie składnie z blokiem generują różne wyniki.

Powodem tego jest odmienne pierwszeństwo. Pierwsza instrukcja `puts` jest interpretowana w następujący sposób:

```
puts(array.map {|n| n * 10 })
```

Druga instrukcja `puts` jest interpretowana następująco:

```
puts(array.map) do |n| n * 10 end
```

W drugim przypadku blok kodu jest interpretowany jako część wywołania metody `puts`, a nie metody `map`. Jeśli wywołasz metodę `puts` z blokiem, zostanie on zignorowany. A zatem wersja ze słowami kluczowymi `do/end` w rzeczywistości jest równoważna następującemu kodowi:

```
puts array.map
```

Z tego właśnie powodu uzyskujemy enumerator.

Wywołanie metody `map` z wykorzystaniem bloku kodu bazującego na słowach kluczowych `do/end` ilustruje to, że jeśli udostępniono blok kodu, ale wywoływana metoda nie rozpoznaje go (lub nie szuka go), nie występuje żaden błąd: metody nie są zobligowane do przekazywania sterowania. Ponadto wiele metod (w tym metoda `map`) ma dobrze zdefiniowane zachowania w sytuacjach, w których blok kodu istnieje albo nie. Jeżeli metoda wydaje się ignorować blok, w przypadku którego oczekujesz, że przekaże mu sterowanie, sprawdź dokładnie reguły pierwszeństwa i upewnij się, że blok faktycznie jest dostępny dla metody.

W dalszym ciągu będziemy się zajmować iteratorami i iteracją, korzystając z kilku wbudowanych iteratorów języka Ruby, które zostały użyte w przypadku metody `loop`. Najpierw przyjrzymy się wbudowanej metodzie, a następnie zaimplementujemy własną. Zaczniemy od metody `times`, która stanowi nieznacznie ulepszoną wersję metody `loop`.

### 6.3.5. Implementowanie metody times

Metoda `times` jest metodą instancji klasy `Integer`. Oznacza to, że metodę tę możesz wywołać w przypadku liczb całkowitych. Metoda uruchamia blok kodu *n* razy dla dowolnej liczby całkowitej *n*, a na końcu działania metody wartością zwracaną jest *n*.

Po uruchomieniu przykładowego kodu z metodą `times` w sesji narzędzia irb zobaczysz zarówno dane wyjściowe, jak i wartość zwracaną:

```
>> 5.times { puts "Jest to zapisywane 5 razy!" } ←❶
Jest to zapisywane 5 razy!
Jest to zapisywane 5 razy!
Jest to zapisywane 5 razy!
Jest to zapisywane 5 razy!
Jest to zapisywane 5 razy!
=> 5                          ←❷
```

Wywołanie metody obejmuje blok kodu ❶, który jest wykonywany pięć razy. Wartość zwracana *całej* metody jest obiektem, od którego zaczęto, czyli liczbą całkowitą 5 ❷.

Działanie metody `times` dobrze ilustruje to, że przekazanie sterowania blokowi i powrót z metody to dwie różne rzeczy. Metoda może przekazywać sterowanie swojemu blokowi dowolną liczbę razy, od zera do nieskończoności (w przypadku metody `loop` jest to nieskończona liczba razy). Jednak każda metoda zwraca wynik dokładnie raz (zakładając, że nie wystąpiły żadne błędy krytyczne) po zakończeniu wykonywania wszystkiego, co miało zostać zrealizowane. Przypomina to trochę wyskok w jeździe figurowej na lodzie. Wybijasz się w powietrze, wykonujesz kilka obrotów i lądujesz. W podobny sposób wywołanie metody powoduje jednokrotne jej wykonanie i zwrócenie wyniku. Jednak tak jak w przypadku obrotów w powietrzu między tymi dwoma operacjami metoda może przekazać sterowanie z powrotem blokowi (jeśli istnieje) co najmniej raz albo wcale.

Przed zaimplementowaniem metody `times` przyjrzyjmy się innej jej funkcji. Za każdym razem, gdy metoda `times` przekazuje sterowanie swojemu blokowi, coś dostarcza. Z całą pewnością bloki kodu, tak jak metody, mogą pobierać argumenty. Podczas przekazywania sterowania metoda może dostarczać co najmniej jedną wartość.

Blok pobiera argument za pośrednictwem swoich parametrów. W przypadku metody `times` możesz podać pojedynczy argument, który zostanie powiązany z dowolną wartością, jaka jest przekazywana blokowi w każdej iteracji. Jak możesz się domyślić, wartości przekazywane przez tę metodę są liczbami całkowitymi z przedziału od 0 do $n - 1$:

```
>> 5.times {|i| puts "To jest iteracja #{i}!" }
To jest iteracja 0!
To jest iteracja 1!
To jest iteracja 2!
To jest iteracja 3!
To jest iteracja 4!
=> 5
```

Każdorazowo podczas przetwarzania, czyli przy przekazywaniu sterowania blokowi kodu przez metodę `times`, dostarcza ona następną wartość, która jest umieszczana w zmiennej `i`.

Jesteśmy gotowi do zaimplementowania metody `times` lub raczej metody `my_times`. Odpowiedni kod wygląda następująco:

```
class Integer
  def my_times
    c = 0
    until c == self
      yield(c)
      c += 1
    end
    self
  end
end
```

Aby wypróbować przykład w sesji narzędzia irb, możesz wpisać w niej poprzedni kod lub umieścić go w pliku (na przykład o nazwie *my_times.rb*), a następnie wykonać polecenie `require "./my_times.rb"` w oknie tego narzędzia. W efekcie po uruchomieniu kodu ujrzysz następujące wyniki:

```
>> 5.my_times {|i| puts "To jest iteracja #{i}!" }
To jest iteracja 0!
To jest iteracja 1!
To jest iteracja 2!
To jest iteracja 3!
To jest iteracja 4!
=> 5
```

Metoda `my_times` działa tak jak metoda `times`. W żadnej mierze implementacja metody `my_times` nie jest najbardziej zwięzłą możliwą implementacją, ale działa. Poznasz znacznie więcej technik modyfikowania kolekcji i iteracji, które możesz wykorzystać do tego, aby definicje własnych metod były bardziej zwięzłe i przejrzyste.

Skoro o tym mowa, zajmiemy się teraz metodą `each`. Jak się tutaj okaże, a jeszcze bardziej w dalszych rozdziałach, w których obszernie prezentujemy obiekty kolekcji, metoda `each` stanowi zasadniczą i intensywnie wykorzystywaną metodę.

### 6.3.6. Ważność metody each

Idea działania metody `each` jest prosta: jest ona uruchamiana dla obiektu kolekcji. Metoda `each` przekazuje każdy element kolekcji blokowi kodu (po jednym na raz). W języku Ruby występuje kilka klas kolekcji, a ponadto jeszcze więcej klas, które wystarczająco przypominają kolekcje, aby obsługiwać metodę `each`. Dwa dalsze rozdziały poświęcono kolekcjom języka Ruby. W tym miejscu na potrzeby przykładów zostanie zastosowana skromna tablica.

Oto prosta operacja z metodą `each`:

```
array = [1,2,3,4,5]
array.each {|e| puts "Blokowi przekazano właśnie element #{e}." }
```

Dane wyjściowe każdego wywołania metody `each` w sesji narzędzia irb prezentują się następująco:

```
>> array.each {|e| puts "Blokowi przekazano właśnie element #{e}." }
Blokowi przekazano właśnie element 1.
Blokowi przekazano właśnie element 2.
```

```
Blokowi przekazano właśnie element 3.
Blokowi przekazano właśnie element 4.
The block just got handed 5.
=> [1, 2, 3, 4, 5]
```

Ostatni wiersz nie należy do danych wyjściowych metody. Jest to wartość zwracana metody `each` potwierdzana przez narzędzie irb. Gdy dla metody `each` określono blok, jej wartością zwracaną jest jej odbiorca, czyli oryginalna tablica (jeśli dla metody nie podano bloku, zwraca ona enumerator; więcej na ten temat znajdziesz w rozdziale 10.). Podobnie jak metoda `times` metoda `each` nie ma robiącej wrażenie wartości zwracanej. Całe zainteresowanie nią wynika z faktu, że metoda `each` przekazuje wartości blokowi.

Aby zaimplementować metodę `my_each`, wykonamy kolejny krok związany z wierszami ulepszenia iteracji. W przypadku metody `my_loop` iteracja trwała bez końca. Z kolei dla metody `my_times` iteracja została wykonana *n* razy. W przypadku metody `my_each` liczba iteracji (liczba przekazań sterowania przez metodę) zależy od wielkości tablicy.

Niezbędny jest licznik do śledzenia bieżącego położenia w tablicy, a także utrzymanie przekazywania sterowania do momentu zakończenia przetwarzania. Wygodnym rozwiązaniem jest zapewnianie przez tablice metody `size`, która ułatwia określenie liczby iteracji („obrotów w powietrzu") wymaganych do wykonania. Jako wartości zwracanej metody użyjemy obiektu oryginalnej tablicy:

```
class Array
  def my_each
    c = 0
    until c == size
      yield(self[c]) ←——— Użycie znaków [] do uzyskania bieżącego elementu tablicy
      c += 1
    end
    self
  end
end
```

Próbne uruchomienie metody `my_each` powoduje uzyskanie żądanego wyniku:

```
>> array = [1,2,3,4,5]
>> array.my_each {|e| puts "Blokowi przekazano właśnie element #{e}." }
Blokowi przekazano właśnie element 1.
Blokowi przekazano właśnie element 2.
Blokowi przekazano właśnie element 3.
Blokowi przekazano właśnie element 4.
Blokowi przekazano właśnie element 5.
=> [1, 2, 3, 4, 5]
```

Pomyślnie zaimplementowaliśmy co najmniej prostą wersję metody `each`. W przypadku tej metody dobre jest to, że jest ona taka zwykła: całe jej działanie sprowadza się do przekazywania wartości w bloku kodu (po jednej na raz) do momentu, aż ich zabraknie. Istotną konsekwencją tego jest możliwość zbudowania *na bazie* metody `each` dowolnej liczby bardziej złożonych iteratorów, które cechują się rozbudowaną semantyką. Prezentowany przykład ponownej implementacji zakończymy metodą `map`,

którą pobieżnie opisano w punkcie 6.3.4. Zaznajamianie się z tą metodą pozwoli nam również poznać kilka dodatkowych niuansów związanych z tworzeniem i stosowaniem bloków kodu.

**Dodatkowe informacje: definiowanie metody my_each z wykorzystaniem metody my_times**

Interesującym eksperymentem jest zdefiniowanie metody `my_each` za pomocą istniejącej definicji metody `my times`. Używając metody `size`, możesz określić wymaganą liczbę iteracji i wykonać je dzięki wsparciu metody `my_times` w następujący sposób:

```
class Array
  def my_each
    size.my_times do |i|
      yield self[i]
    end
    self
  end
end
```

Zastosowanie metody `my_times` pozwala uniknąć konieczności tworzenia kodu licznika pętli w metodzie `my each`. Jest to jednak trochę przestarzałe rozwiązanie: wiele iteratorów języka Ruby tworzonych jest na bazie metody `each`, a nie w inny sposób. Jak daną definicję metody `my each` przedstawioną wcześniej w rozdziale zastosowałbyś w implementacji metody `my_times`?

W przeciwieństwie do pierwszego przykładowego eksperymentu w przypadku tego naprawdę pozostawiamy swobodę odnośnie do podjęcia decyzji o przeprowadzeniu go samemu!

### *6.3.7. Od metody each do metody map*

Podobnie jak metoda `each` metoda `map` przetwarza po jednym elemencie tablicy i przekazuje go blokowi kodu. Różnica między tymi metodami tkwi w wartości zwracanej: metoda `each` zwraca swojego odbiorcę, a metoda `map` zwraca nową tablicę. Nowa tablica zawsze ma taką samą wielkość jak oryginalna tablica, ale zamiast oryginalnych elementów nowa tablica zawiera skumulowane wartości zwracane bloku kodu z iteracji.

Poniżej przedstawiono przykład użycia metody `map`. Zauważ, że wartość zwracana zawiera nowe elementy. Nie jest to już tablica, od której rozpoczęto przetwarzanie:

```
>> names = ["Dawid", "Adam", "Nowak"]
=> ["Dawid", "Adam", "Nowak"]
>> names.map {|name| name.upcase }
=> ["DAWID", "ADAM", "NOWAK"]
```

Odwzorowanie powoduje utworzenie nowej tablicy, której każdy element odpowiada elementowi w tym samym położeniu oryginalnej tablicy, ale przetwarzanemu za pomocą bloku kodu. Elementem układanki dodawanym przez metodę `map` do prezentowanej analizy iteracji jest zagadnienie bloku kodu zwracającego wartość *do* metody, która przekazała blokowi sterowanie. Wygląda to następująco: tak jak metoda może przekazać wartość, tak też blok może zwrócić wartość. Wartość zwracana przekazywana jest jako wartość wywołania metody `yield`.

Aby zaimplementować metodę my_map, konieczne jest określenie tablicy akumulującej, do której będą wstawiane wartości zwracane kolejnych wywołań bloku kodu. Tablica akumulująca zostanie następnie zwrócona jako wynik całego wywołania metody my_map.

Zacznijmy od wstępnej, a nie ostatecznej implementacji, w której nie bazujemy na metodzie my_each, lecz tworzymy metodę my_map od podstaw. Ma to na celu dokładne zilustrowanie, czym odwzorowanie różni się od zwykłej iteracji. Później ulepszymy implementację.

Pierwsza implementacja wygląda następująco:

```
class Array
  def my_map
    c = 0
    acc = []  <------ Inicjuje tablicę akumulującą
    until c == size
      acc << yield(self[c])  <------ Przechwytuje wartość zwracaną z bloku w tablicy akumulującej
      c += 1
    end
    acc  <------ Zwraca tablicę akumulującą
  end
end
```

Z metody my_map uzyskujemy teraz te same wyniki co w przypadku metody map:

```
>> names.my_map {|name| name.upcase }
=> ["DAWID", "ADAM", "NOWAK"]
```

Podobnie jak metoda my_each metoda my_map przekazuje kolejno każdy element tablicy. W przeciwieństwie do metody my_each metoda my_map przechowuje wartość zwracaną z bloku. Metoda akumuluje odwzorowanie starych wartości na nowe w następujący sposób: nowe wartości bazują na starych wartościach przetwarzanych w obrębie bloku.

Jednak użyta implementacja metody my_map „nie dotrzymuje obietnicy" metody my_each, czyli tego, że metoda each pełni rolę zwykłego iteratora, na bazie którego mogą być tworzone bardziej złożone iteratory. Zaimplementujmy ponownie metodę map. Tym razem zostanie utworzona metoda my_map z wykorzystaniem metody my_each.

### TWORZENIE METODY MAP NA BAZIE METODY EACH

Tworzenie metody map na bazie metody each jest zadziwiająco proste:

```
class Array
  # W tym miejscu umieść definicję metody my_each
  def my_map
    acc = []
    my_each {|e| acc << yield(e) }
    acc
  end
end
```

Wykorzystujemy zwykły iterator, umożliwiając metodzie my_each sprawdzenie kolejnych elementów tablicy. Nie ma potrzeby utrzymywania jawnego licznika lub tworzenia pętli z instrukcją until. Związana z tym logika jest już zawarta w metodzie my_each. Warto skorzystać z tego faktu podczas tworzenia metody my_map.

O iteratorach można napisać znacznie więcej; w szczególności o sposobach bazujących na metodzie `each`, jakie w języku Ruby mają na celu zapewnienie wyjątkowo rozbudowanego zestawu narzędziowego złożonego z metod przetwarzających kolekcje. Zajmiemy się tym w rozdziale 10. Na razie zagłębmy się jeszcze bardziej w podstawy dotyczące iteratorów, począwszy od reguł przypisań i określania zasięgu, które zarządzają sposobem wykorzystania parametrów i zmiennych przez iteratory.

### 6.3.8. Parametry bloku i zasięg zmiennych

Wcześniej wspomniano, że parametry bloku są otoczone przez symbole potoku, a nie przez nawiasy okrągłe jak w przypadku parametrów metody. Informacje o argumentach metody możesz jednak wykorzystać do tworzenia list parametrów bloku. Czy pamiętasz metodę `args_unleashed` z rozdziału 2.?

```
def args_unleashed(a,b=1,*c,d,e)
  puts "Argumenty:"
  p a,b,c,d,e
end
```

Oto wersja metody bazująca na bloku:

```
def block_args_unleashed
  yield(1,2,3,4,5)
end
block_args_unleashed do |a,b=1,*c,d,e|
  puts "Argumenty:"
  p a,b,c,d,e
end
```

Powiązania parametrów i dane wyjściowe programu są takie same jak w przypadku oryginalnej wersji metody:

```
Argumenty:
1
2
[3]
4
5
```

A co z zasięgiem? Jak wiesz, definicja metody tworzy nowy zasięg lokalny. Bloki są trochę bardziej skomplikowane.

Zacznijmy od prostego przypadku: wewnątrz bloku odwołujesz się do zmiennej (nie do parametru bloku, a jedynie do zmiennej) o nazwie `x`. Przed utworzeniem bloku w zasięgu została już uzyskana zmienna o takiej nazwie:

```
def block_scope_demo
  x = 100
  1.times do  <------ Pojedyncza iteracja ma za zadanie utworzenie kontekstu bloku kodu
    puts x
  end
end
```

Po uruchomieniu metody (zawiera przydatną instrukcję `puts`) okaże się, że zmienna `x` w obrębie bloku jest taka sama jak istniejąca już zmienna `x`:

```
block_scope_demo  ←——— Dane wyjściowe: 100
```

A co z przypisaniem do zmiennej wewnątrz bloku? I tym razem okazuje się, że zmienna w bloku jest identyczna jak zmienna, która istniała przed jego utworzeniem. Możesz to stwierdzić przez zmodyfikowanie tej zmiennej w obrębie bloku, a następnie wyświetlenie jej wartości po zakończeniu przetwarzania bloku:

```
def block_scope_demo_2
  x = 100
  1.times do
    x = 200
  end
  puts x
end
block_scope_demo_2  ←——— Dane wyjściowe: 200
```

Inaczej mówiąc, bloki mają bezpośredni dostęp do już istniejących zmiennych (tak jak w przypadku zmiennej `x` w omawianym przykładzie). Jednak parametry bloku (nazwy zmiennych znajdujące się między symbolami potoku) zachowują się inaczej niż zmienne, które nie są parametrami. Jeśli w zasięgu istnieje zmienna o danej nazwie, a ponadto tej samej nazwy użyjesz dla jednego z parametrów bloku, takie dwie zmienne (jedna zmienna już istnieje, a druga należy do listy parametrów) *nie* są identyczne.

> **UWAGA** Choć samo w sobie jest to już ważne, fakt, że bloki współużytkują zasięg lokalny z poprzedzającym je kodem, nabierze dodatkowego znaczenia przy analizowaniu obiektów `Proc` i *domknięć* w rozdziale 14. Dowiesz się, że bloki mogą pełnić rolę zawartości obiektów funkcji anonimowych, które zachowują zmienne lokalne znajdujące się w zasięgu w czasie ich tworzenia (nawet jeśli obiekty funkcji są przekazywane między innymi zasięgami lokalnymi).

Przyjrzyj się zmiennym `x` w następującym przykładzie:

```
def block_local_parameter
  x = 100     ←——— Zewnętrzna zmienna x (przed blokiem)
  [1,2,3].each do |x|  ←——— Parametr bloku x
    puts "Parametr x to #{x}"
    x = x + 10         ←——— Przypisanie do zmiennej x wewnątrz bloku
    puts "Dokonano ponownego przypisania do zmiennej x w bloku. Teraz jest to #{x}"
  end
  puts "Zewnętrzna zmienna x to nadal #{x}"
end
```

Oto dane wyjściowe wywołania tej metody:

```
Parametr x to 1
Dokonano ponownego przypisania do zmiennej x w bloku. Teraz jest to 11
Parametr x to 2
Dokonano ponownego przypisania do zmiennej x w bloku. Teraz jest to 12
Parametr x to 3
Dokonano ponownego przypisania do zmiennej x w bloku. Teraz jest to 13
Zewnętrzna zmienna x to nadal 100
```

Zmienna `x` w obrębie bloku nie jest tym samym co zmienna `x` poza nim, ponieważ zmienna wewnętrzna jest parametrem bloku. Nawet ponowne przypisanie do zmiennej `x`

wewnątrz bloku nie powoduje nadpisania zewnętrznej zmiennej x. Takie działanie umożliwia zastosowanie dowolnej żądanej nazwy zmiennej dla parametrów bloku bez konieczności martwienia się o to, czy zmienna o takiej samej nazwie znajduje się już w zasięgu.

Czasami może być wymagane użycie zmiennej tymczasowej wewnątrz bloku, nawet wtedy, gdy nie jest to jeden z parametrów przypisywanych w momencie wywołania bloku. W tym przypadku wskazane jest, aby nie było konieczne martwienie się o przypadkowe ponowne użycie zmiennej spoza bloku. Język Ruby zapewnia specjalną notację wskazującą, że co najmniej jedna zmienna ma być lokalna względem bloku, nawet w sytuacji, gdy istnieją już zmienne o takiej samej nazwie. Notacja ta ma postać znaku średnika na liście parametrów bloku.

Pora na przykład. Zwróć uwagę na średnik na następującej liście parametrów:

```
def block_local_variable
  x = "Oryginalna zmienna x!"
  3.times do |i;x|
    x = i
    puts "Zmienna x w bloku to teraz #{x}"
  end
  puts "Zmienna x za zakończonym blokiem to #{x}"
end
block_local_variable
```

Średnik, po którym następuje zmienna x, wskazuje, że blok wymaga własnej zmiennej x niepowiązanej z żadną zmienną x, jaka mogła już zostać utworzona w zasięgu poza blokiem. W prezentowanym przykładzie ma miejsce przypisanie do zmiennej x wewnątrz bloku, ale takie przypisania nie mają wpływu na już istniejącą zmienną x. Dane wyjściowe potwierdzają, że oryginalna zmienna x jest zachowywana:

```
Zmienna x w bloku to teraz 0
Zmienna x w bloku to teraz 1
Zmienna x w bloku to teraz 2
Zmienna x za zakończonym blokiem to Oryginalna zmienna x!
```

Z całą pewnością oryginalna zmienna x została uchroniona przed zmianą.

Zmienne wyszczególnione po średniku nie są traktowane jak parametry bloku. Po wywołaniu bloku zmienne te nie są z niczym wiązane. Są one *nazwami zastrzeżonymi*, które mają zapewniać możliwość użycia ich w roli nazw zmiennych tymczasowych wewnątrz bloku bez potrzeby przeprowadzania sprawdzania pod kątem kolizji z nazwami spoza bloku.

Podsumowując, dostępne są następujące trzy podstawowe „odmiany” zmiennej bloku:

- Zmienne lokalne istniejące już w momencie tworzenia bloku.
- Parametry bloku, które zawsze są lokalne względem bloku.
- Prawdziwe zmienne lokalne względem bloku, które są wyszczególniane po średniku, a ponadto nie są przypisywane, lecz chronią dowolne zmienne o takiej samej nazwie przed zasięgiem zewnętrznym.

Gdy dysponuje się takimi narzędziami, powinno być możliwe takie skonstruowanie bloków, aby realizowały żądane działania odnośnie do zmiennych i zasięgu. Oznacza to, że nie będzie konieczne „pozbycie się" z zasięgu zewnętrznego żadnych zmiennych, w przypadku których nie jest to pożądane.

Iteratory i bloki kodu w języku Ruby umożliwiają tworzenie i używanie metod konstruowanych w celu współużytkowania własnych funkcji z ich obiektami wywołującymi. Metoda zawiera logikę i procedurę, ale w momencie wywoływania jej udostępniasz dodatkowy kod, który zapewnia logikę, a ponadto indywidualizuje konkretne tworzone wywołanie. Jest to zgrabna funkcja o nieskończonej liczbie zastosowań. W rozdziałach 10. i 11. powrócimy do iteratorów podczas szczegółowego omawiania obiektów kolekcji.

Na razie jednak przyjrzymy się kolejnemu mechanizmowi przepływu sterowania. Do tej pory działaliśmy w „scenerii" cechującej się kooperatywnością i efektywnością. Nie zawsze jednak odbywa się to w ten sposób. Jednym z najważniejszych aspektów związanych z przepływem sterowania, który wymaga zrozumienia, jest kwestia tego, co się dzieje, gdy pojawią się problemy.

## 6.4. Obsługa błędów i wyjątki

W rozdziale 1. przyjrzeliśmy się sposobowi sprawdzania kodu pod kątem błędów składni:

```
$ ruby -cw filename.rb
```

Przekazanie parametrów testu `-cw` oznacza, że interpreter języka Ruby może uruchomić program. Nie oznacza to jednak, że wszystko się powiedzie w czasie działania programu. Możesz utworzyć poprawny składniowo program, który zostanie zaakceptowany i wykonany przez interpreter. Program ten może realizować wszelkiego rodzaju niedozwolone operacje. W języku Ruby niedopuszczalne działania w czasie wykonywania kodu są obsługiwane przez *zgłoszenie wyjątku*.

### 6.4.1. Zgłaszanie wyjątków i stosowanie dla nich klauzuli rescue

*Wyjątek* to specjalny rodzaj obiektu, który jest instancją klasy `Exception` lub jej klasy potomnej. *Zgłoszenie* wyjątku oznacza zatrzymanie standardowego wykonywania programu i zajęcie się napotkanym problemem lub całkowite zakończenia działania programu.

To, która z tych operacji zostanie wykonana (zajmowanie się problemem lub przerwanie pracy programu), zależy od tego, czy podano klauzulę `rescue`. Jeśli jej nie określono, program zostanie zakończony. W przeciwnym razie sterowanie zostanie przekazane tej klauzuli.

Aby zaznajomić się z działaniem wyjątków w praktyce, spróbuj wykonać operację dzielenia przez zero:

```
$ ruby -e '1/0'
```

Interpreter języka Ruby zgłasza wyjątek:

```
-e:1:in `/': divided by 0 (ZeroDivisionError)
    from -e:1:in `<main>'
```

ZeroDivisionError to nazwa konkretnego wyjątku. Według bardziej technicznego opisu jest to nazwa klasy potomnej klasy Exception. W języku Ruby występuje cała rodzina klas wyjątków w postaci drzewa, z których wszystkie ostatecznie prowadzą do klasy Exception.

**TYPOWE WYJĄTKI**

W tabeli 6.1 zaprezentowano kilka typowych wyjątków (każdy z nich jest klasą pochodzącą od klasy Exception) wraz z częstymi przyczynami ich zgłaszania. Podano też przykład kodu, który powoduje zgłoszenie poszczególnych wyjątków.

**Tabela 6.1.** Typowe wyjątki

| Nazwa wyjątku | Typowe przyczyny | Kod powodujący zgłoszenie wyjątku |
|---|---|---|
| RuntimeError | Domyślny wyjątek zgłaszany przez metodę raise. | Raise |
| NoMethodError | Obiektowi wysłano komunikat, dla którego nie może on określić nazwy metody. Wyjątek ten jest zgłaszany przez domyślną metodę method_missing. | a = Object.new<br>a.nieznana_nazwa_metody |
| NameError | Interpreter napotkał identyfikator, dla którego nie może określić nazwy zmiennej lub metody. | a = losowy_identyfikator |
| IOError | Wyjątek jest powodowany przez odczyt zamkniętego strumienia, zapis do strumienia tylko do odczytu oraz podobne operacje. | STDIN.puts("Nie zapisuj do strumienia STDIN!") |
| Errno::błąd | Rodzina błędów powiązanych z plikowymi operacjami wejścia-wyjścia. | File.open(-12) |
| TypeError | Metoda odebrała argument, którego nie może obsłużyć. | a = 3 + "Nie można dodać łańcucha do liczby!" |
| ArgumentError | Wyjątek jest powodowany użyciem niewłaściwej liczby argumentów. | def m(x); end;<br>m(1,2,3,4,5) |

Powyższe przykłady możesz wypróbować w sesji narzędzia irb. Choć zostanie wyświetlony komunikat o błędzie, sesja nie powinna zostać zakończona. Narzędzie to dobrze radzi sobie z traktowaniem potencjalnych błędów krytycznych jak błędów, które takimi nie są. We własnych programach również możesz osiągnąć coś podobnego.

### *6.4.2. Słowo kluczowe rescue na ratunek!*

Zgłoszenie wyjątku nie musi oznaczać zakończenia pracy programu. Wyjątki mogą być obsługiwane (zajęcie się problemem i utrzymanie działania programu) z wykorzystaniem słowa kluczowego rescue. Operacja obsługi wyjątku uwzględnia blok słowa kluczowego ograniczony słowami kluczowymi begin i end. W środku bloku znajduje się klauzula rescue:

```
print "Wprowadź liczbę: "
n = gets.to_i
begin
  result = 100 / n
```

```
rescue
  puts "Podana liczba spowodowała niepowodzenie operacji. Czy to było zero???"
  exit
end
puts "100/#{n} to #{result}."
```

Po uruchomieniu tego programu i wprowadzeniu zera jako liczby operacja dzielenia (`100/n`) zgłosi wyjątek `ZeroDivisionError`. Ponieważ operacja została wykonana wewnątrz bloku `begin/end` z klauzulą `rescue`, sterowanie jest jej przekazywane. Po wyświetleniu komunikatu o błędzie program kończy działanie.

Jeśli wprowadzisz coś innego niż `0` i operacja dzielenia powiedzie się, sterowanie programu pominie instrukcję `rescue` i blok, a wykonywanie kodu zostanie następnie wznowione (przez wywołanie metody `puts`).

Technikę tę możesz ulepszyć przez dokładne wskazanie wyjątku, który ma zostać przechwycony. Zamiast stosowania ogólnego wariantu działania instrukcji `rescue`, która obsługuje każdy błąd będący klasą potomną klasy `StandardError`, instruujesz instrukcję, co ma być przez nią obsługiwane:

```
rescue ZeroDivisionError
```

Powyższa instrukcja przechwytuje pojedynczy typ wyjątku, pomijając pozostałe. Zaletą jest to, że eliminujesz ryzyko nieumyślnego uwzględnienia jakiegoś innego problemu w wyniku zbyt przesadnego stosowania klauzuli `rescue`.

Z obsługą wyjątków za pomocą klauzuli `rescue` w obrębie metody lub bloku kodu związanych jest kilka różnych funkcji, które są godne uwagi.

### UŻYCIE KLAUZULI RESCUE WEWNĄTRZ METOD I BLOKÓW KODU

Początek metody lub bloku kodu zapewnia niejawny kontekst słów kluczowych `begin/end`. A zatem jeśli w obrębie metody lub bloku kodu używasz słowa kluczowego `rescue`, nie musisz jawnie określać słowa kluczowego `begin`, przy założeniu, że klauzula `rescue` ma zarządzać całą metodą lub blokiem:

```
def open_user_file
  print "Plik do otwarcia: "
  filename = gets.chomp
  fh = File.open(filename)      ←❶
yield fh
  fh.close
  rescue                        ←❷
    puts "Nie można otworzyć pliku!"
end
```

Jeśli operacja otwierania pliku ❶ spowoduje zgłoszenie wyjątku, sterowanie przechodzi bezpośrednio do klauzuli `rescue` ❷. Słowa kluczowe `def/end` służą do ograniczenia zasięgu operacji realizowanej za pomocą tej klauzuli.

Możesz jednak wymagać trochę większej dokładności odnośnie do tego, do jakich wierszy kodu ma zastosowanie klauzula `rescue`. W poprzednim przykładzie klauzula jest wyzwalana nawet wtedy, gdy wyjątek zgłoszono z powodów, które nie mają nic wspólnego z próbą otwarcia pliku. Jeśli na przykład z dowolnego powodu wywołanie metody `gets` powoduje zgłoszenie wyjątku, klauzula `rescue` zostanie zastosowana.

Aby zapewnić większą dokładność, musisz ponownie skorzystać z jawnego bloku opakowującego słów kluczowych `begin/end`:

```
def open_user_file
  print "Plik do otwarcia: "
  filename = gets.chomp
  begin                          ←❶
    fh = File.open(filename)
  rescue                         ←❷
    puts "Nie można otworzyć pliku!"
    return                       ←❸
  end
  yield fh
  fh.close
end
```

W tej wersji kodu klauzula `rescue` zarządza jedynie tym, co trafia między słowo kluczowe `begin` ❶ i nią samą ❷. Co więcej, w obrębie klauzuli `rescue` ❸ niezbędne jest jawne określenie instrukcji `return`, ponieważ w przeciwnym razie metoda będzie w dalszym ciągu wykonywana.

Do tej pory przyjrzeliśmy się temu, jak przechwytywać wyjątki zgłaszane przez interpreter języka Ruby (poznasz więcej technik przechwytywania wyjątków). Zajmijmy się jednak teraz „drugą stroną monety", czyli sposobem zgłaszania wyjątków we własnym zakresie.

### 6.4.3. Jawne zgłaszanie wyjątków

W przypadku tradycyjnej elastyczności języka Ruby i możliwości zwięzłego tworzenia w nim kodu wyjątki nie są wyjątkiem, jak by to można wyrazić. Wyjątki możesz zgłaszać we własnym kodzie, a ponadto masz możliwość tworzenia nowych wyjątków, które mają zostać zgłoszone.

Aby zgłosić wyjątek, używasz instrukcji `raise` oraz nazwy wyjątku do zgłoszenia. Jeśli nie podasz nazwy wyjątku (a ponadto nie zgłosisz ponownie innego rodzaju wyjątku, o czym wspomniano w punkcie 6.4.4), interpreter języka Ruby zgłasza raczej ogólny wyjątek `RuntimeError`. Możliwe jest też przekazanie instrukcji `raise` drugiego argumentu, który pełni rolę łańcucha komunikatu podczas zgłaszania wyjątku:

```
def fussy_method(x)
  raise ArgumentError, "Wymagana jest liczba mniejsza niż 10" unless x < 10
end
fussy_method(20)
```

W przypadku uruchomienia z pliku o nazwie *fussy.rb* powyższy kod wyświetla następujący komunikat:

```
fussy.rb:2:in `fussy_method': Wymagana jest liczba mniejsza niż 10 (ArgumentError)
    from fussy.rb:5:in `<main>'
```

W następującym przypadku możesz również użyć instrukcji `raise`:

```
begin
  fussy_method(20)
rescue ArgumentError
```

```
  puts "Podana liczba jest niedozwolona!"
end
```

Wygodne jest to, że jeśli instrukcji `raise` zostanie przekazany komunikat jako jedyny argument, a nie jako drugi argument, co w tym przypadku oznaczałoby, że pierwszym argumentem jest klasa wyjątku, instrukcja ta zostanie poinstruowana do zgłoszenia wyjątku `RuntimeError` z wykorzystaniem udostępnionego komunikatu. Następujące dwa wiersze są równorzędne:

```
raise "Problem!"
raise RuntimeError, "Problem!"
```

W klauzulach `rescue` możliwe jest przechwycenie obiektu wyjątku w zmiennej, a następnie odpytanie jej w celu uzyskania ewentualnych przydatnych informacji.

### 6.4.4. Przechwytywanie wyjątku w klauzuli rescue

Aby do zmiennej przypisać obiekt wyjątku, używany jest specjalny operator `=>` wraz z poleceniem `rescue`. Obiekt wyjątku, tak jak dowolny obiekt, odpowiada na komunikaty (metody). W szczególności przydatne są metody `backtrace` i `message`. Metoda `backtrace` zwraca tablicę łańcuchów reprezentujących stos wywołań w momencie zgłoszenia wyjątku: nazwy metod, nazwy plików i numery wierszy, które stanowią pełny „drogowskaz" kodu wykonanego do chwili wystąpienia wyjątku. Metoda `message` zwraca łańcuch komunikatu przekazany (jeśli określono) instrukcji `raise`.

Aby sprawdzić te rozwiązania w praktyce, wcześniej zamieszczoną definicję metody `fussy_method` umieść w pliku *fussy.rb* (jeśli jeszcze tego nie zrobiono), a następnie dodaj następujący blok `begin/end`:

```
begin
  fussy_method(20)
rescue ArgumentError => e          ←❶
  puts "Podana liczba jest niedozwolona!"
  puts "Oto dane śledzenia wstecznego dla tego wyjątku:"
  puts e.backtrace                 ←❷
  puts "Oto komunikat obiektu wyjątku:"
  puts e.message                   ←❸
end
```

W klauzuli `rescue` do zmiennej `e` ❶ przypisywany jest obiekt wyjątku, a następnie żądane jest od niego wyświetlenie swoich danych śledzenia wstecznego ❷ oraz komunikatu ❸. Jeśli między metodą `fussy_method` i słowem kluczowym `begin` wstawiono pusty wiersz, to po uruchomieniu pliku *fussy.rb* zostaną uzyskane następujące dane wyjściowe (w dowolnym przypadku ujrzysz prawie takie same dane, choć mogą się różnić liczby wierszy):

```
Podana liczba jest niedozwolona!
Oto dane śledzenia wstecznego dla tego wyjątku:
fussy.rb:2:in `fussy_method'
fussy.rb:6:in `<main>'
Oto komunikat obiektu wyjątku:
Wymagana jest liczba mniejsza niż 10
```

Dane śledzenia wstecznego pokazują, że w momencie zgłoszenia wyjątku wykonywanie kodu miało miejsce w metodzie `fussy_method` w drugim wierszu pliku *fussy.rb*. Ponadto wcześniej wykonywany był kod w szóstym wierszu tego samego pliku w kontekście `<main>`, czyli inaczej mówiąc, na najwyższym poziomie programu (poza jakąkolwiek definicją klasy, modułu lub metody). Komunikat `Wymagana jest liczba mniejsza niż 10` pochodzi z wywołania metody `raise` wewnątrz metody `fussy_method`.

Klauzula `rescue` może też spowodować ponowne zgłoszenie wyjątku, który wyzwoliła.

**Co jest zgłaszane: wyjątek czy klasa wyjątku?**

Język zgłaszania wyjątku oparty jest na klasach. Oznacza to użycie kodu `raise ZeroDivi↪sionError` zamiast kodu `raise ZeroDivisionError.new`. W rzeczywistości jednak zgłaszane są instancje klas wyjątków. Składnia umożliwia zgłaszanie klasy, ponieważ wygląda to lepiej, a ponadto dokonuje abstrakcji faktu występowania operacji tworzenia instancji.

Przełącznik klasa/instancja możesz zauważyć podczas analizowania obiektu przechwytywanego w klauzuli `rescue`:

```
begin
raise ArgumentError
rescue => e
p e.class  ←——— Klasa ArgumentError
end
```

Klasą obiektu jest klasa `ArgumentError`. Sam obiekt jest instancją klasy `ArgumentError`, a nie klasą `ArgumentError`.

Nie trzeba używać konstrukcji `.new`, a kod ma możliwość ogólnego wglądu, zapewniając ilość informacji wystarczającą do zaprezentowania programiście tego, co ma miejsce, bez zbędnych szczegółów związanych z obsługą.

### PONOWNE ZGŁASZANIE WYJĄTKU

Częstym wymogiem jest ponowne zgłaszanie wyjątku. Umożliwia to jego obsługę przez następną pozycję w stosie wywołań po tym, jak obsługą wyjątku zajmował się blok klauzuli `rescue`. Na przykład może być konieczne rejestrowanie danych informacji o wyjątku, który nadal ma być traktowany przez kod wywołujący jak wyjątek.

Poniżej zaprezentowano drugą wersję bloku `begin/end` z metody `open_user_file` przedstawionej kilka przykładów wcześniej. W tej wersji założono, że istnieje metoda `logfile`, która zwraca uchwyt pliku zapisywalnego dla pliku dziennika:

```
begin
  fh = File.open(filename)
rescue => e
  logfile.puts("Użytkownik próbował otworzyć plik #{filename}, #{Time.now}")
  logfile.puts("Wyjątek: #{e.message}")
  raise
end
```

W tym przypadku celem jest przechwycenie wyjątku, utworzenie dla niego wpisu w pliku dziennika, a następnie ponowne zgłoszenie wyjątku przez wywołanie klauzuli `raise` (nawet pomimo tego, że nie podano żadnego argumentu tej klauzuli, wewnętrznie określa ona, że ma zostać ponownie zgłoszony obsługiwany wyjątek, a nie zwykły, ogólny wyjątek `RuntimeError`). Kod w programie, który jako pierwszy spowodował

wywołanie metody open_user_file, musi zatem obsłużyć wyjątek albo nie, jeśli lepszym wariantem będzie zezwolenie wyjątkowi na zatrzymanie wykonywania programu.

Innym ulepszeniem obsługi przepływu sterowania za pomocą wyjątków jest klauzula ensure, która w momencie zgłoszenia wyjątku jest wykonywana bezwarunkowo, niezależnie od tego, co jeszcze ma miejsce.

### 6.4.5. Klauzula ensure

Załóżmy, że chcesz wczytać wiersz z pliku danych i zgłosić wyjątek, gdy wiersz nie będzie zawierać określonego podłańcucha. Jeśli podłańcuch istnieje, wymagasz zwrócenia wiersza. W przeciwnym razie oczekujesz zgłoszenia wyjątku ArgumentError. Cokolwiek jednak się wydarzy, żądasz zamknięcia uchwytu pliku przed zakończeniem metody.

Oto możliwy sposób osiągnięcia tego za pomocą klauzuli ensure:

```
def line_from_file(filename, substring)
  fh = File.open(filename)
  begin
    line = fh.gets
    raise ArgumentError unless line.include?(substring)
  rescue ArgumentError
    puts "Niepoprawny wiersz!"
    raise
  ensure
    fh.close
  end
  return line
end
```

W tym przykładzie blok begin/end opakowuje wiersz wczytywany z pliku, a klauzula rescue obsługuje jedynie wyjątek ArgumentError. Oznacza to, że jeśli coś innego się nie powiedzie (np. okaże się, że plik nie istnieje), klauzula rescue nie zostanie zastosowana. Jeśli jednak wyjątek ArgumentError zostanie zgłoszony w wyniku testu dołączenia podłańcucha substring do łańcucha line, klauzula rescue zostanie wykonana.

Co więcej, klauzula ta jest wykonywana niezależnie od tego, czy wyjątek został zgłoszony, czy nie. Klauzula ensure jest powiązana ze strukturą bloku begin/end, której stanowi część, a jej wykonywanie jest bezwarunkowe. W tym przykładzie wymagane jest zapewnienie zamknięcia uchwytu pliku. Klauzula ensure zajmuje się tym, niezależnie od tego, co innego może się jeszcze wydarzyć.

> **UWAGA** Istnieje lepszy sposób otwierania pliku oparty na bloku kodu, który opakowuje operacje na plikach, a ponadto zajmuje się zamykaniem pliku. Zajmijmy się jednak na razie tylko jednym. Technika ta zostanie omówiona w rozdziale 12., w którym przyjrzymy się operacjom wejścia-wyjścia i operacjom plikowym.

Problemem utrzymującym się w przypadku metody line_from_file jest to, że Argument
↪Error to nie najlepsza nazwa dla zgłaszanego wyjątku. Najbardziej odpowiednia byłaby nazwa taka jak InvalidLineError, która nie istnieje. Na szczęście możesz tworzyć własne klasy wyjątków i nadawać im dowolne żądane nazwy.

### 6.4.6. Tworzenie własnych klas wyjątków

Nowa klasa wyjątków jest tworzona przez dziedziczenie z klasy `Exception` lub jej klasy potomnej:

```
class MyNewException < Exception
end
raise MyNewException, "Wystąpił nowy rodzaj błędu!"
```

Technika ta oferuje dwie podstawowe korzyści. Po pierwsze, umożliwiając nadawanie nowych nazw klasom wyjątków, technika realizuje funkcję automatycznego dokumentowania: w momencie zgłoszenia wyjątek `MyNewException` różni się od na przykład wyjątku `ZeroDivisionError` lub zwykłego wyjątku `RuntimeError`.

Po drugie, technika pozwala dokładnie określić operacje uwzględniające użycie klauzuli `rescue`. Po utworzeniu wyjątku `MyNewException` możesz zastosować dla niego tę klauzulę, przekazując jej nazwę wyjątku:

```
class MyNewException < Exception
end
begin
  puts "Zostanie zgłoszony wyjątek..."
  raise MyNewException
rescue MyNewException => e
  puts "Właśnie zgłoszono wyjątek: #{e}"
end
```

Oto dane wyjściowe powyższego fragmentu kodu:

```
Zostanie zgłoszony wyjątek...
Właśnie zgłoszono wyjątek: MyNewException
```

Klauzula `rescue` będzie przechwytywać tylko błędy `MyNewException`. Jeśli najpierw z dowolnego powodu zostanie zgłoszony inny wyjątek, spowoduje to zakończenie programu bez użycia klauzuli `rescue`.

Poniżej zaprezentowano wcześniej omawianą metodę `line_from_file` z zastosowanym wyjątkiem niestandardowym (uwzględniono kod, który tworzy klasę wyjątków niestandardowych). Dziedziczenie odbywa się z klasy `StandardError`, czyli superklasy klasy `RuntimeError`:

```
class InvalidLineError < StandardError
end
def line_from_file(filename, substring)
  fh = File.open(filename)
  line = fh.gets
  raise InvalidLineError unless line.include?(substring)
  return line
rescue InvalidLineError
    puts "Niepoprawny wiersz!"
    raise
  ensure
    fh.close
end
```

Tym razem w pełni określono wyjątek, który miał zostać przechwycony.

Przez samo dziedziczenie z klasy `StandardError` klasa `InvalidLineError` zapewnia konkretną nazwę wyjątku i udoskonala semantykę operacji bazującej na klauzuli `rescue`. Tworzenie klas wyjątków niestandardowych jest proste i nie wymaga dużego nakładu pracy, a ponadto może zaoferować znaczącą wartość. Sam język Ruby zawiera wiele klas wyjątków, dlatego nie wahaj się tworzyć własnych wyjątków za każdym razem, gdy uznasz, że żaden z wbudowanych wyjątków nie wyraża wystarczająco tego, czego wymagasz. I nie zapomnij o tym, że wyjątki to klasy, a klasy to stałe, dla których dzięki zagnieżdżaniu mogą być tworzone przestrzenie nazw:

```
module TextHandler
  class InvalidLineError < StandardError
  end
end
def line_from_file(filename, substring)
  fh = File.open(filename)
  line = fh.gets
  raise TextHandler::InvalidLineError unless line.include?(substring)
end
```

**Nazwa wyjątku z dobrze określoną przestrzenią nazw!**

Tworzenie w ten sposób przestrzeni nazw dla wyjątków jest „grzeczne" w tym sensie, że umożliwia innym osobom określanie przez nie żądanych nazw wyjątków bez obawy o kolizje nazw. Staje się to też koniecznością, gdy zaczynasz tworzenie czegoś więcej niż tylko bardzo niewielkiej liczby klas wyjątków.

Omawiając wyjątki i sposób ich obsługi, dotarliśmy do końca prezentacji przepływu sterowania. Jak pokazano, sterowanie może być w dość znacznym stopniu przekazywane. Jeśli jednak pamiętasz o różnych rodzajach elementów przekazywania sterowania (instrukcje warunkowe, pętle, iteratory i wyjątki), będziesz w stanie prześledzić dowolny kod Ruby, a ponadto utworzyć kod, który efektywnie wykorzysta wiele dostępnych technik powiązanych z przepływem sterowania.

## 6.5. Podsumowanie

W rozdziale zaprezentowano następujące zagadnienia:

- Instrukcje warunkowe (`if/unless` i `case/when`).
- Pętle (`loop`, `for`, `while` i `until`).
- Iteratory i bloki kodu, w tym parametry i zmienne bloku.
- Przykłady implementowania metod języka Ruby w kodzie Ruby.
- Wyjątki i ich obsługa.

W rozdziale omówiono kilka obszernych zagadnień, które połączono ze sobą, ponieważ ich wspólnym elementem jest to, że uwzględniają przepływ sterowania. Instrukcje warunkowe przekazują sterowanie na podstawie prawdziwości lub fałszu wyrażeń. Pętle powtarzają segment kodu bezwarunkowo, warunkowo lub jednokrotnie dla każdego elementu listy. Iteratory (metody przekazujące sterowanie blokowi kodu zapewnionemu wraz z wywołaniem metody) należą do najbardziej wyróżniających się funkcji języka Ruby. Dowiedziałeś się, jak utworzyć i wywołać iterator. Są to techniki, z którymi będziesz mieć często do czynienia w dalszej części książki (i nie tylko).

Wyjątki są mechanizmem języka Ruby służącym do obsługi warunków błędów *niemożliwych do usunięcia*, co jest pojęciem względnym: dla warunku błędu możesz użyć klauzuli `rescue` i kontynuować wykonywanie kodu. W takiej sytuacji wymagane jednak będzie przygotowanie zamierzonej interwencji za pośrednictwem bloku klauzuli `rescue`, a zatem zmiana i przejęcie sterowania programem w miejscu, w którym w innym razie nastąpiłoby zakończenie jego wykonywania. Możliwe jest też tworzenie własnych klas wyjątków przez dziedziczenie z wbudowanych klas wyjątków języka Ruby.

Na tym etapie zagłębimy się w tematykę wbudowanych funkcji języka Ruby, począwszy od ogólnych i wszechobecnych funkcji oraz technik, a skończywszy na konkretnych klasach i modułach. Do tej pory przedstawiono i użyto wielu wbudowanych funkcji. Pora jednak na większą systematyczność i szersze omówienie sposobu działania wbudowanych elementów języka.

# Część II

# Wbudowane klasy i moduły

W części II zajmiemy się istotą języka Ruby, czyli wbudowanymi klasami i modułami.

Spora część tego, co zrealizujesz jako programista używający języka Ruby, będzie obejmować jego wbudowane elementy. Zaprezentowano już przykłady wykorzystujące wiele spośród nich: łańcuchy, tablice, pliki itd. Język Ruby zapewnia bogaty zestaw narzędzi ze standardowymi, wbudowanymi typami danych, które mogą być stosowane, a także na bazie których można tworzyć kod.

Podczas projektowania własnych klas i modułów często stwierdzisz, że to, co jest niezbędne, przypomina istniejącą klasę języka Ruby. Jeśli na przykład tworzysz klasę `DeckOfCards`, jedną z pierwszych Twoich myśli będzie prawdopodobnie to, że talia kart bardzo przypomina tablicę. W związku z tym pomyślisz o tym, czy klasa kart powinna być podklasą klasy `Array`, a być może o tym, że każdy obiekt talii może przechowywać tablicę w zmiennej instancji i w niej umieszczać karty itd. Rzecz w tym, że klasy wbudowane języka Ruby zapewniają bazę wyjściową na potrzeby naszych własnych projektów klas i obiektów, a także oferują zestaw klas, które już same w sobie są wyjątkowo przydatne.

Na początku części II przyjrzymy się podstawowym elementom wbudowanym (rozdział 7.). Celem tego rozdziału jest zaprezentowanie asortymentu (jak to można określić) technik i narzędzi, które okażą się przydatne w treści dalszych rozdziałów.

Aby na przykład poznawać łańcuchy, tablice i tablice asocjacyjne, warto wiedzieć, w jaki sposób w języku Ruby traktowane są pojęcia *prawda* i *fałsz*, które nie są powiązane z żadną pojedynczą klasą wbudowaną, ale wymagają ogólnego zrozumienia.

Po omówieniu podstawowych elementów zajmiemy się konkretnymi klasami, lecz pogrupowanymi za pomocą ogólnych kategorii: najpierw zostaną przedstawione obiekty *skalarne* (rozdział 8.), a następnie *kolekcje* (rozdział 9.). Obiekty skalarne to obiekty niepodzielne, takie jak łańcuchy, liczby i symbole. Każdy obiekt skalarny reprezentuje jedną wartość. Obiekty te nie zawierają innych obiektów (oczywiście w łańcuchach znajdują się znaki, ale w języku Ruby nie występuje osobna klasa znaków, dlatego łańcuchy w dalszym ciągu są obiektami skalarnymi). Obiekty kolekcji zawierają inne obiekty. Podstawowe klasy kolekcji w języku Ruby to tablice i tablice asocjacyjne. W omówieniu kolekcji będą też uwzględnione *zakresy*, będące obiektami hybrydowymi, które mogą (ale nie zawsze) służyć do reprezentowania kolekcji obiektów. Na końcu przyjrzymy się zbiorom, które są implementowane w standardowej bibliotece (a nie w rdzeniu języka Ruby), ale zasługują na odstępstwo od ogólnej reguły, zgodnie z którą koncentrujemy się na samym rdzeniu języka.

Na równi z konkretnymi klasami kolekcji ważne są rozwiązania współużytkowane przez wszystkie kolekcje w języku Ruby: są to rozwiązania wbudowane w moduł `Enumerable`. Zapewnia on obiektom kolekcji informacje, jak na wiele sposobów wykonywać w swoim obrębie operacje przechodzenia i transformacji. Rozdział 10. poświęcono modułowi `Enumerable` oraz jego wpływowi na możliwości programowania w przypadku języka Ruby.

W kolejnym, 11. rozdziale części II zaprezentowano wyrażenia regularne (choć są one powiązane z łańcuchami, zasługują na osobne omówienie). Tę część książki zakończono rozdziałem 12., w którym objaśniono operacje plikowe oraz operacje wejścia-wyjścia: odczytywanie i zapisywanie plików oraz strumieni wejścia-wyjścia, a także powiązane dodatkowe zagadnienia, takie jak obsługa błędów i zapytania dotyczące statusu plików. Nie jest zaskoczeniem to, że w języku Ruby wszystkie te elementy, w tym wyrażenia regularne i strumienie wejścia-wyjścia, są traktowane jako obiekty.

Po przeczytaniu części II będziesz dysponować dużą wiedzą praktyczną o podstawowych klasach języka Ruby. Twoje horyzonty wiedzy związane z tym językiem zostaną w znacznym stopniu rozszerzone.

# 7 Wbudowane elementy podstawowe

**W tym rozdziale:**

- Konstruktory obiektów literału.
- „Lukier" składniowy.
- Metody „niebezpieczne" i/lub destrukcyjne.
- Rodzina metod konwersji `to_*`.
- Stany i obiekty boolowskie oraz obiekt `nil`.
- Techniki porównywania obiektów.
- Inspekcja możliwości obiektów w czasie działania kodu.

W dalszych rozdziałach tej części książki zostaną omówione konkretne klasy wbudowane. Zostanie przybliżone to, czym są, jakie zapewniają możliwości, a także jakie metody zawierają ich instancje. W tym rozdziale zajmiemy się niektórymi zagadnieniami, które dotyczą wielu klas wbudowanych.

Niniejszy rozdział jest jednak czymś więcej: jest to również swego rodzaju następnej generacji przewodnik po języku Ruby, czyli bardziej szczegółowa i obszerniejsza wersja rozdziału 1. Podobnie jak w jego przypadku dla rozdziału 6. ustalono dwa cele. Pierwszym celem jest umożliwienie wykorzystania określonej ilości materiału w kolejnych rozdziałach, w których w różnych miejscach będzie on się pojawiać w różnym stopniu. Drugim celem jest zaprezentowanie Czytelnikowi informacji o języku Ruby,

które na swój sposób są ważne i przydatne. W trakcie lektury rozdziału poznasz „bogactwo" tkwiące w każdym obiekcie języka, a także niektóre podsystemy składniowe i semantyczne, które sprawiają, że język jest tak interesujący i wszechstronny.

Ponieważ w rozdziale poruszanych jest wiele zagadnień, prawdopodobnie warto wcześniej dowiedzieć się, jakie one są. Oto ich podsumowanie z niewielkim komentarzem:

- *Konstruktory literałów*. Metody tworzenia konkretnych obiektów za pomocą składni, a nie przez wywołanie metody `new`.
- *„Lukier" składniowy*. Działania możliwe w języku Ruby, które zapewniają lepszy wygląd kodu.
- *Metody „niebezpieczne" i/lub destrukcyjne*. Metody trwale zmieniające swoich odbiorców oraz inne „groźne" kwestie.
- *Rodzina metod konwersji* `to_*`. Metody dokonujące konwersji obiektu w obiekt innej klasy oraz powiązane z nimi funkcje składniowe.
- *Stany i obiekty boolowskie oraz obiekt* `nil`. Dokładna analiza wartości `true` i `false`, a także powiązanych z nimi zagadnień występujących w języku Ruby.
- *Techniki porównywania obiektów*. Techniki dotyczące całego języka Ruby, zarówno domyślne, jak i oferujące możliwość dostosowywania, które są używane na potrzeby porównywania obiektów.
- *Inspekcja możliwości obiektów w czasie działania kodu*. Ważny zestaw technik służących do sprawdzania możliwości obiektu w czasie działania kodu.

Wszystkie powyższe zagadnienia uznasz za przydatne podczas analizowania i/lub tworzenia kodu Ruby w ramach lektury książki, a także już po jej przeczytaniu.

Czytając rozdział, możesz uruchomić sesję narzędzia irb. W rozdziale często wykorzystywany jest format tej sesji na potrzeby przykładów z kodem. Często możesz sprawdzać przykłady z niewielkimi różnicami, aby zorientować się, jak działa kod Ruby.

## 7.1. Konstruktory literałów języka Ruby

Język Ruby oferuje mnóstwo klas wbudowanych. W przypadku większości z nich instancja jest tworzona za pomocą metody `new`:

```
str = String.new
arr = Array.new
```

Dla niektórych klas zastosowanie tej metody nie jest możliwe. Na przykład nie możesz utworzyć nowej instancji klasy `Integer`. Przeważnie jednak możliwe jest tworzenie nowych instancji klas wbudowanych.

Ponadto wśród klas wbudowanych istnieje kilka wybranych „szczęśliwców", którzy mogą korzystać z przywileju posiadania *konstruktorów literałów*. Oznacza to, że do utworzenia nowego obiektu takiej klasy możesz użyć specjalnej notacji, a nie wywołania metody `new`.

W tabeli 7.1 zaprezentowano klasy z konstruktorami literałów. Gdy korzystasz z jednego z tych konstruktorów, w efekcie powstaje nowy obiekt (choć wprost nie

wynika to z treści tabeli, godne uwagi jest to, że w przypadku obiektów `Symbol` nie istnieje konstruktor `new`; jedynym sposobem wygenerowania obiektu `Symbol` jest zastosowanie konstruktora literałów).

**Tabela 7.1.** Klasy wbudowane języka Ruby z konstruktorami literałów

| Klasa | Konstruktor literału | Przykłady |
|---|---|---|
| `String` | Znaki cudzysłowu | `"new string"`<br>`'new string'` |
| `Symbol` | Umieszczony na początku dwukropek | `:symbol`<br>`:"symbol ze spacjami"` |
| `Array` | Nawiasy kwadratowe | `[1,2,3,4,5]` |
| `Hash` | Nawiasy klamrowe | `{"Nowy Jork" => "NY", "Oregon" => "OR"}` |
| `Range` | Dwie lub trzy kropki | `0..9 or 0...10` |
| `Regexp` | Znaki `/` | `/([a-z]+)/` |
| `Proc` (`lambda`) | Kreska, strzałka, nawiasy okrągłe i klamrowe | `->(x,y) { x * y }` |

Dość szczegółowo przyjrzymy się sporej liczbie funkcji wszystkich tych klas. Na razie zacznijmy od przyzwyczajenia się do notacji, aby możliwe było szybkie rozpoznawanie takich typów danych. Konstruktory literałów nigdy nie są jedynym sposobem tworzenia instancji w postaci obiektu danej klasy, ale są bardzo często używane.

W dalszej kolejności przyjrzymy się wybranym elementom stanowiącym „lukier” składniowy udostępniany w języku Ruby w przypadku wielu obiektów.

**Znaki konstruktorów literałów o więcej niż jednym znaczeniu**

Część notacji używanej w przypadku konstruktorów literałów w języku Ruby ma więcej niż jedno znaczenie. Wiele obiektów zawiera metodę o nazwie `[]`, która przypomina konstruktor tablicy literałów, ale nim nie jest. Jak już pokazano, choć bloki kodu mogą być ograniczone przez nawiasy klamrowe, w dalszym ciągu są blokami, a nie literałami tablicy asocjacyjnej. Tego rodzaju przeciążenie notacji jest konsekwencją skończonej liczby symboli dostępnych na klawiaturze. Zawsze możesz zidentyfikować znaczenie notacji dzięki jej kontekstowi. Istnieje naprawdę niewiele kontekstów, w przypadku których przy niewielkim doświadczeniu łatwa będzie taka identyfikacja.

## 7.2. Powracający „lukier” składniowy

Jak wiesz, czasami w języku Ruby możliwe jest zastosowanie notacji w postaci „lukru” składniowego, która zastępuje zwykłą składnię wywoływania metody `object.method(args)`. Umożliwia to wykonanie działań upiększających kod, takich jak użycie znaku plusa między dwoma liczbami jako operatora:

```
x = 1 + 2
```

zamiast dziwnie wyglądającego odpowiednika charakterystycznego dla metod:

```
x = 1.+(2)
```

Zagłębiając się bardziej w język Ruby i jego metody wbudowane, bądź świadom tego, że określone metody zawsze są w ten sposób traktowane. Konsekwencją tego jest możliwość zdefiniowania sposobu zachowania obiektów w kodzie podobnym do następującego:

```
my_object + my_other_object
```

W tym celu wystarczy zdefiniować metodę +. Proces ten został zaprezentowany w praktyce, a zwłaszcza w powiązaniu z równością przypadków i definiowaniem metody ===. Przyjrzymy się teraz jednak w szerszym zakresie tej eleganckiej technice.

### *7.2.1. Definiowanie operatorów przez określanie metod*

Jeśli dla klasy zdefiniujesz metodę +, to w celu wykonania operacji dodawania obiekty tej klasy mogą użyć „lukru" składniowego. Co więcej, nie ma czegoś takiego, jak definiowanie znaczenia takiej składni niezależnie od definiowania metody. Operator jest metodą, która po prostu prezentuje się ładniej jako operator.

Pamiętaj też o tym, że semantyka metod, takich jak +, jest całkowicie zależna od konwencji. Dla interpretera języka Ruby znak + nie oznacza dodawania. Nic (z wyjątkiem dobrej oceny sytuacji) nie powstrzyma Cię przed tworzeniem metod +, które nie mają nic wspólnego z dodawaniem:

```
obj = Object.new
def obj.+(other_obj)
  "Czy podejmowana jest próba dodania czegoś do mnie?"
end
puts obj + 100   ←——— Brak dodawania. Zwracane są tylko dane wyjściowe
```

Znak plusa w instrukcji `puts` oznacza wywołanie metody + obiektu `obj` z liczbą całkowitą `100` jako pojedynczym argumentem (ignorowanym).

Ponad „lukrem" w stylu operatora znajduje się „lukier" w postaci skrótu `x +=1` dla `x = x + 1`. I tym razem w przypadku zdefiniowania odpowiedniej metody lub metod automatycznie uzyskasz „żniwa" w postaci „lukru". Oto przykład klasy kont bankowych z metodami + i -:

```
class Account
  attr_accessor :balance
  def initialize(amount=0)
    self.balance = amount
  end
  def +(x)
    self.balance += x
  end
  def -(x)        ←——— ❶
    self.balance -= x
  end
  def to_s
    balance.to_s
  end
end
acc = Account.new(20)
acc -= 5          ←——— ❷
puts acc          ←——— Dane wyjściowe: 15
```

Definiując metodę instancji - ❶, uzyskujemy skrót -=, a ponadto taka notacja ❷ umożliwia odejmowanie od konta. Jest to prosty, ale pouczający przykład tego, że język Ruby zachęca do wykorzystania dokładnie tego samego „okablowania", jakiego sam używa do integrowania programów z bazową technologią w sposób możliwie jak najbardziej płynny.

W tabeli 7.2 zebrano metody, dla których automatycznie używany jest „lukier" składniowy.

**Tabela 7.2.** Metody z notacją wywołania, która bazuje na „lukrze" składniowym w stylu operatorów

| Kategoria | Nazwa | Przykład definicji | Przykład wywołania | Notacja z „lukrem" składniowym |
|---|---|---|---|---|
| Metoda/operatory arytmetyczne | + | def +(x) | obj.+(x) | obj + x |
| | - | def -(x) | obj.-(x) | obj - x |
| | * | def *(x) | obj.*(x) | obj * x |
| | / | def /(x) | obj./(x) | obj / x |
| | % (modulo) | def %(x) | obj.%(x) | obj % x |
| | ** (wykładnik potęgi) | def **(x) | obj.**(x) | obj ** x |
| Pobieranie/ustawianie/ dołączanie danych | [] | def [](x) | obj.[](x) | obj[x] |
| | []= | def []=(x,y) | obj.[]=(x,y) | obj[x] = y |
| | << | def <<(x) | obj.<<(x) | obj << x |
| Metoda/operatory porównywania | <=> | def <=>(x) | obj.<=>(x) | obj <=> x |
| | == | def ==(x) | obj.==(x) | obj == x |
| | > | def >(x) | obj.>(x) | obj > x |
| | > | def <(x) | obj.<(x) | obj < x |
| | >= | def >=(x) | obj.>=(x) | obj >= x |
| | <= | def <=(x) | obj.<=(x) | obj <= x |
| Operator równości przypadków | === | def ===(x) | obj.===(x) | obj === x |
| Operatory bitowe | \| (OR) | def \|(x) | obj.\|(x) | obj \| x |
| | & (AND) | def &(x) | obj.&(x) | obj & x |
| | ^ (XOR) | def ^(x) | obj.^(x) | obj ^ x |

Nie jest trudno zapamiętać, jakie metody objęte są używaniem „lukru" składniowego. Metody te należą do kilku różnych kategorii widocznych w tabeli 7.2. Kategorie służą jedynie wygodzie podczas uczenia się języka i odwoływania się do jego elementów. W języku Ruby metody nie są kategoryzowane, a za implementowanie sensownej semantyki odpowiada programista. Nazwy kategorii wskazują, w jaki sposób nazwy tych metod są używane w klasach wbudowanych języka Ruby, a także jak zwyczajowo są one najczęściej wykorzystywane przez programistów, którzy implementują je w nowych klasach.

Nie zapomnij też o operatorze przypisania warunkowego `||=`, a także o jego rzadko spotykanym kuzynie `&&=`. Oba operatory zapewniają tego samego rodzaju skrót co metody pseudooperatora, ale oparte są na operatorach `||` i `&&`, których nie możesz przesłonić.

Intensywne użycie tego rodzaju „lukru" składniowego, w przypadku którego coś *przypomina* operator, ale *jest* wywołaniem metody, mówi wiele o filozofii towarzyszącej językowi programowania Ruby. To, że możesz definiować, a nawet ponownie definiować elementy takie jak znak plus, znak minus oraz nawiasy kwadratowe, oznacza, że język Ruby oferuje naprawdę sporą elastyczność. Istnieją jednak ograniczenia dotyczące możliwości ponownego definiowania w tym języku. Nie możesz ponownie zdefiniować żadnego z konstruktorów obiektów literału: `{}` to zawsze literał tablicy asocjacyjnej (lub blok kodu, jeśli występuje w tym kontekście), a znaki `""` zawsze będą ograniczać łańcuch itd.

Istnieje jednak możliwość wielu działań. Możesz nawet zdefiniować operatory jednoargumentowe za pomocą definicji metody.

### 7.2.2. Dostosowywanie operatorów jednoargumentowych

Operatory jednoargumentowe `+` i `-` występują najczęściej jako znaki liczb (np. –1). Mogą one być jednak definiowane. Na potrzeby własnych obiektów i klas możesz określić działanie wyrażeń `+obj` i `-obj`. W tym celu definiowane są metody `+@` i `-@`.

Załóżmy, że operatory `+` i `-` mają zmieniać wielkość liter w przypadku obiektu podobnego do obiektu łańcuchowego. Oto kod prezentujący sposób definiowania odpowiedniego działania operatorów jednoargumentowych na przykładzie klasy `Banner`:

```
class Banner
  def initialize(text)
    @text = text
  end
  def to_s      <------ ❶
    @text
  end
  def +@
    @text.upcase
  end
  def -@
    @text.downcase
  end
end
```

Utwórz teraz nagłówek i modyfikuj wielkość jego liter za pomocą operatorów jednoargumentowych `+` i `-`:

```
banner = Banner.new("Zjedz w restauracji Dawida!")
puts banner   <------ Dane wyjściowe: Zjedz w restauracji Dawida!
puts +banner  <------ Dane wyjściowe: ZJEDZ W RESTAURACJI DAWIDA!
puts -banner  <------ Dane wyjściowe: zjedz w restauracji dawida!
```

Dane wyjściowe w postaci podstawowego łańcucha, którym jest niezmieniony tekst nagłówka, zapewniane są przez metodę konwersji `to_s` ❶. Metoda ta zostanie dokładnie omówiona w punkcie 7.4.1.

Definiując metodę `!`, możesz też zdefiniować operator `!` (*negacja* logiczna). Okazuje się, że definiowanie metody `!` zapewnia zarówno operator jednoargumentowy `!`, jak i słowo kluczowe `not`. Dodajmy definicję do klasy `Banner`:

```
class Banner
  def !
    reverse
  end
end
```

Sprawdźmy teraz nagłówek w postaci „zanegowanej”. Aby wyraźnie określić pierwszeństwo wyrażeń, konieczne będzie umieszczenie w nawiasach okrągłych wersji ze słowem `not` (w przeciwnym razie metoda `puts` rozpozna próbę wyświetlenia łańcucha `not`):

```
puts !banner      <------ Dane wyjściowe: !adiwaD ijcaruatser w zdejZ
puts (not banner) <------ Dane wyjściowe: !adiwaD ijcaruatser w zdejZ
```

Jak to często w jego przypadku język Ruby zapewnia oparty na metodzie obiektowy sposób dostosowywania tego, co początkowo możesz uznać za powiązane na stałe elementy składniowe, dotyczy to nawet operatorów jednoargumentowych, takich jak `!`.

Negacja z użyciem operatora jednoargumentowego nie jest jedynym wariantem wykorzystania znaku wykrzyknika w języku Ruby.

## 7.3. Metody z nazwą zakończoną wykrzyknikiem i metody „niebezpieczne”

Nazwy metod języka Ruby mogą być zakończone wykrzyknikiem (`!`). Wewnętrznie nie ma on żadnego znaczenia w tym języku. Metody o takich nazwach są wywoływane i wykonywane tak jak dowolne inne metody. Zwyczajowo jednak znak wykrzyknika identyfikuje metodę jako „niebezpieczną”, a dokładniej rzecz biorąc, jako „niebezpieczny” odpowiednik metody o takiej samej nazwie, lecz bez wykrzyknika.

Określenie *niebezpieczne* może mieć dowolne znaczenie, jakie zostanie nadane metodzie przez tworzącą ją osobę. W przypadku klas wbudowanych oznacza to zwykle, że *taka metoda, w przeciwieństwie do jej odpowiednika bez wykrzyknika w nazwie, trwale modyfikuje swojego odbiorcę*. Nie zawsze jednak tak jest: `exit!` to „niebezpieczna” alternatywa metody `exit` w tym sensie, że nie uruchamia żadnych metod finalizujących poza obrębem programu. „Niebezpieczeństwo” metody `sub!` (w łańcuchu w miejsce dopasowanego wzorca wstawia łańcuch zastępujący) po części wynika z tego, że zmienia swojego odbiorcę, a po części z faktu, że zwraca wartość `nil`, jeśli nie miała miejsca żadna zmiana, w przeciwieństwie do metody `sub`, która zawsze zwraca kopię oryginalnego łańcucha z zastosowanym zastąpieniem (lub bez niego).

Jeśli „niebezpieczeństwo” uznasz za zbyt melodramatyczne określenie, obecność wykrzyknika w nazwach metod możesz potraktować jak coś w rodzaju „alertu”. Z wyłączeniem bardzo niewielu specyficznych wyjątków każda metoda powinna występować w parze ze swym odpowiednikiem bez wykrzyknika w nazwie. Powrócimy do kwestii związanych z najlepszą strategią nadawania nazw metodom po przyjrzeniu się działaniu niektórych metod z wykrzyknikiem w nazwie.

### 7.3.1. Destrukcyjne efekty (zmieniające odbiorcę) uznawane za „niebezpieczne"

Bez wątpienia większość metod z wykrzyknikiem w nazwie, które napotkasz w rdzeniu języka Ruby, zawiera w nazwie ten znak, ponieważ są destrukcyjne: zmieniają one obiekt, w którym są wywoływane. Wywołanie metody `upcase` dla łańcucha zapewnia nowy łańcuch mający postać oryginalnego łańcucha, ale złożonego z dużych liter. Z kolei metoda `upcase!` zamienia pierwotny łańcuch w jego odpowiednik zawierający małe litery:

```
>> str = "Witaj"
=> "Witaj"
>> str.upcase
=> "WITAJ"
>> str
=> "Witaj"  ←——❶
>> str.upcase!
=> "WITAJ"
>> str
=> "WITAJ"  ←——❷
```

Sprawdzenie oryginalnego łańcucha po przekształceniu go w wersję złożoną z dużych liter pokazuje, że jest ona kopią. Oryginalny łańcuch pozostał niezmieniony ❶. Operacja wykonana przez metodę z wykrzyknikiem w nazwie zmieniła jednak zawartość samej zmiennej `str` ❷.

Podstawowe klasy języka Ruby są pełne destrukcyjnych (zmieniających odbiorcę) metod z wykrzyknikiem w nazwie powiązanych z ich niedestrukcyjnymi odpowiednikami: `sort/sort!` (tablice), `strip/strip!` (usuwanie początkowego i końcowego białego znaku w przypadku łańcuchów), `reverse/reverse!` (łańcuchy i tablice) oraz wiele innych metod. Jeśli dla obiektu wywołasz wersję metody bez wykrzyknika w nazwie, w każdym przypadku uzyskasz nowy obiekt. Jeśli z kolei wywołasz wersję metody z wykrzyknikiem w nazwie, ma miejsce działanie dotyczące tego samego obiektu, do którego wysłano komunikat.

Zawsze należy mieć świadomość tego, czy wywoływana metoda zmienia swojego odbiorcę. Żadna opcja nie jest ani dobra, ani zła. To, która opcja jest najlepsza, zależy od wykonywanej operacji. Kwestią, która faworyzuje modyfikowanie obiektów zamiast ich tworzenia, jest wydajność: tworzenie nowych obiektów (np. drugiego łańcucha różniącego się od pierwszego tylko jedną literą) jest kosztowne pod względem używanej pamięci i czasu przetwarzania. Nie ma znaczenia to, czy w grę wchodzi niewielka liczba obiektów. Gdy jednak zajmiesz się na przykład obsługą danych z dużych plików z wykorzystaniem pętli i iteratorów, tworzenie nowych obiektów może spowodować znaczne zużywanie zasobów.

Patrząc na to z innej strony, niezbędne jest zachowanie ostrożności przy lokalnym modyfikowaniu obiektów, ponieważ inne części programu mogą bazować na tym, że takie obiekty nie zostaną zmienione. Dla przykładu przyjmijmy, że istnieje baza danych imion. Nazwy są wczytywane z bazy do tablicy. W pewnym momencie musisz przetworzyć imiona w celu wyświetlenia danych wyjściowych, które w całości są złożone z dużych liter. Możesz wykonać następujący kod:

```
names.each do |name|
  capped = name.upcase
  # ... kod przetwarzający obiekt capped...
end
```

W tym przykładzie capped to nowy obiekt, czyli duplikat obiektu name złożony z dużych liter. Jeśli później przetworzysz tę samą tablicę, to w sytuacji, gdy *nie zechcesz*, aby nazwy zawierały duże litery (np. w celu ponownego zapisania nazw w bazie danych), nazwy będą mieć pierwotną postać.

Tworząc nowy łańcuch (capped) do reprezentowania wersji każdej nazwy złożonej z dużych liter, unikasz efektu ubocznego trwałego zmieniania nazw. Operacja wykonywana dla nazw osiąga swoje cele bez zmiany podstawowego stanu danych. Czasami wymagana będzie trwała zmiana obiektu, a czasami nie. Nie ma w tym nic złego, pod warunkiem że wiesz, co robisz i z jakiego powodu.

Ponadto nie przyjmuj bezpośredniej korelacji między metodami z wykrzyknikiem w nazwie i metodami destrukcyjnymi. Często pokrywają się one, ale nie są tym samym.

### 7.3.2. Destrukcyjność i „niebezpieczeństwo” zmieniają się niezależnie

W tym punkcie zamieszczono komentarz dotyczący konwencji i sprawdzonych procedur. Język Ruby nie zwraca na to uwagi. Jego interpreter bez obaw wykona metody, których nazwy są zakończone znakiem !, niezależnie od tego, czy są one „niebezpieczne”, bezpieczne, powiązane z metodą bez wykrzyknika w nazwie, pozbawione powiązania z inną metodą itd. Wartość zapisu z użyciem znaku !, który pełni rolę tokenu komunikacji między autorem metody i jej użytkownikiem, tkwi całkowicie w konwencjach. Warto dobrze zrozumieć te konwencje, a także powody, dla których mają sens.

Najlepsza rada dotycząca tego, kiedy używać nazw metod zakończonych wykrzyknikiem, to...

#### UŻYWAJ ZNAKU ! WYŁĄCZNIE W PRZYPADKU PAR METOD M/M!

Notacja nazwy metody z wykrzyknikiem powinna być stosowana tylko wtedy, gdy istnieje metoda o takiej samej nazwie bez wykrzyknika, relacja między tymi dwoma metodami jest taka, że obie realizują zasadniczo to samo działanie, a ponadto w sytuacji, gdy wersja metody z wykrzyknikiem w nazwie ma również efekty uboczne, inną wartość zwracaną bądź cechuje się odmiennym zachowaniem, które odróżnia ją od jej odpowiednika bez wykrzyknika w nazwie.

Nie używaj znaku ! tylko dlatego, że uważasz, że utworzona metoda jest niebezpieczna w bliżej nieokreślony, abstrakcyjny sposób. Wszystkie metody realizują jakieś działanie, które samo w sobie jest niebezpieczne. Znak ! to ostrzeżenie, że może mieć miejsce więcej, niż sugeruje nazwa metody. To z kolei ma sens tylko wtedy, gdy nazwa jest używana dla metody, która nie cechuje się niebezpiecznym zachowaniem.

Nie nadawaj metodzie nazwy save! tylko z tego powodu, że dokonuje zapisu w pliku. Nadaj takiej metodzie nazwę save. Jeśli istnieje inna metoda zapisująca dane w pliku, ale na przykład nie tworzy kopii zapasowej oryginalnego pliku (przy założeniu, że metoda save tak postępuje), możesz śmiało nadać jej nazwę save!.

Jeśli postanowisz utworzyć metodę zapisującą w pliku i na końcu jej nazwy umieścisz znak `!`, ponieważ obawiasz się, że ma ona zbyt duże możliwości lub jest zbyt niepewna, należy ponownie zastanowić się nad stosowanym nazewnictwem metod. Każdy doświadczony programista używający języka Ruby, który w dokumentacji znajdzie metodę `save!`, będzie zainteresowany stwierdzeniem, czym się ona różni od metody `save`. Wykrzyknik nie oznacza niczego wyizolowanego. Znak ten ma sens tylko na końcu nazwy wchodzącej w skład jednej pary nazw metod, które poza tym znakiem niczym się nie różnią.

#### NIE STAWIAJ ZNAKU RÓWNOŚCI MIĘDZY NOTACJĄ Z WYKRZYKNIKIEM A DZIAŁANIEM DESTRUKCYJNYM I ODWROTNIE

W przypadku nazwy metody z wykrzyknikiem „niebezpieczeństwo" oznacza zwykle działanie powodujące modyfikowanie obiektu lub działanie destrukcyjne. Z tego względu często można usłyszeć osoby twierdzące, że znak `!` oznacza destrukcyjność. Nawiązując do tego, naprawdę niewiele brakuje, aby zaczęto się zastanawiać, dlaczego nazwy niektórych metod destrukcyjnych nie są zakończone znakiem `!`.

Taki sposób rozumowania od samego początku jest problematyczny. Wykrzyknik nie oznacza destrukcyjności, a jedynie niebezpieczne, być może nieoczekiwane zachowanie. Jeśli używasz metody o nazwie `upcase`, a ponadto zamierzasz utworzyć jej wersję destrukcyjną, bez obaw możesz nadać jej nazwę `destructive_upcase`. Żadna reguła nie określa, że do oryginalnej nazwy musisz dodać znak `!`. Jest to tylko konwencja, ale ekspresyjna.

Nazwy metod destrukcyjnych nie zawsze są zakończone znakiem `!`. Nazwy wielu metod z wykrzyknikiem sprawiają, że *oczekuje się* zmiany odbiorcy przez metodę. Takie metody nie mają niedestrukcyjnych odpowiedników. Co oznaczałoby zastosowanie niedestrukcyjnej wersji metody `String#clear`, która usuwa wszystkie znaki z łańcucha, i powoduje, że ma on postać `""`? Jeśli łańcuch nie jest modyfikowany w miejscu, w którym występuje, dlaczego nie można by po prostu od razu zapisać łańcucha w postaci `""`? Jeśli nazwa metody bez wykrzyknika sugeruje już lokalną modyfikację lub dowolnego innego rodzaju „niebezpieczne zachowanie", nie należy jej uważać za niebezpieczną.

Prawie na pewno stwierdzisz, że tradycyjne użycie notacji ze znakiem `!` to najbardziej elegancki i logiczny wariant. Jeśli nie zgadzasz się z tymi konwencjami, najlepiej nie umieszczać wykrzyknika w nazwach metod.

Pozostawiając „zagrożenie" za sobą, zajmijmy się teraz rozwiązaniami, jakie język Ruby zapewnia w celu przekształcania jednego obiektu w drugi.

## 7.4. Metody wbudowane i niestandardowe to_* służące do konwersji

Język Ruby oferuje kilka metod wbudowanych z nazwami złożonymi z przedrostka `to_` oraz wskaźnika klasy, *z* którą wiąże obiekt przekształcająca go metoda: `to_s` (przekształca w łańcuch), `to_sym` (przekształca w symbol), `to_a` (przekształca w tablicę), `to_i` (przekształca w liczbę całkowitą) i `to_f` (przekształca w liczbę zmiennoprzecin-

kową). Nie wszystkie obiekty odpowiadają na każdą z tych metod. Jednak wiele obiektów reaguje na wiele spośród tych metod, a obowiązująca zasada jest na tyle spójna, aby zapewnić wspólne ich traktowanie.

### 7.4.1. Konwersja łańcucha: metoda to_s

Najczęściej używana metoda z rodziny `to_*` to prawdopodobnie metoda `to_s`. Każdy obiekt języka Ruby (z wyjątkiem instancji klasy `BasicObject`) odpowiada na metodę `to_s`, a tym samym ma możliwość zaprezentowania się w postaci łańcucha. Jak prezentuje poniższy fragment sesji narzędzia irb, metoda `to_s` realizuje różne działania, począwszy od zwracania jedynie własnego odbiorcy, gdy obiekt jest już łańcuchem.

```
>> "Jestem już łańcuchem!".to_s
=> "Jestem już łańcuchem!"
```

a skończywszy na zwracaniu łańcucha zawierającego reprezentację obiektu przypominającą kod:

```
>> ["jeden", "dwa", "trzy", 4, 5, 6].to_s
=> "[\"jeden\", \"dwa\", \"trzy\", 4, 5, 6]"
```

W przypadku tego kodu znak `\` umieszczony przed znakami cudzysłowu oznacza, że wewnątrz łańcucha faktycznie występuje znak cudzysłowu. Wśród działań metody `to_s` jest też zwracanie opisowego, choć zagadkowego, łańcucha z informacjami dotyczącymi obiektu:

```
>> Object.new.to_s
=> "#<Object:0x000001030389b0>"
```

Istotną kwestią związaną z metodą `to_s` jest to, że stosowana jest ona przez określone metody oraz w konkretnych kontekstach składniowych, aby zapewnić kanoniczną reprezentację łańcuchową obiektu. Na przykład metoda `puts` wywołuje metodę `to_s` dla swoich argumentów. Jeśli utworzysz własną metodę `to_s` dla klasy lub przesłonisz ją dla obiektu, metoda ta objawi się w momencie przekazania obiektu metodzie `puts`. Możesz to wyraźnie zauważyć — choć jest to trochę absurdalne — używając obiektu ogólnego:

```
>> obj = Object.new
=> #<Object:0x000001011c9ce0>  ←❶
>> puts obj
#<Object:0x000001011c9ce0>     ←❷
=> nil
>> def obj.to_s  ←❸
>>   "Jestem obiektem!"
>> end
=> :to_s  ←❹
>> puts obj  ←❺
Jestem obiektem!
=> nil
```

Domyślna reprezentacja łańcuchowa obiektu to zwykły zrzut ekranowy zawierający nazwę klasy i położenie w pamięci ❶. Dokładnie takie dane wyjściowe ujrzysz po wywołaniu metody `puts` dla obiektu ❷. Jeśli jednak zdefiniujesz niestandardową

metodę `to_s` dla obiektu ❸, kolejne wywołania metody `puts` będą odzwierciedlać nową definicję ❺. Zauważ, że dla samej definicji metody określany jest symbol `:to_s`, który reprezentuje nazwę metody ❹.

Dane wyjściowe metody `to_s` uzyskujesz również w przypadku użycia obiektu w interpolacji łańcucha:

```
>> "Mój obiekt mówi: #{obj}"
=> "Mój obiekt mówi: Jestem obiektem!"
```

Nie zapomnij też o tym, że jawnie możesz wywoływać metodę `to_s`. Nie musisz czekać, aż interpreter języka Ruby zacznie jej szukać. Jednak duży procent wywołań tej metody to zautomatyzowane wywołania realizowane w tle w imieniu metody `puts` lub mechanizmu interpolacji.

> **UWAGA** W przypadku generowania reprezentacji łańcuchowych instancji tablic wykonują one działania odstające nieznacznie od normy. Jeśli metodę `puts` wywołasz dla tablicy, uzyskasz cykliczną reprezentację opartą na wywoływaniu metody `to_s` dla każdego elementu tablicy i zwracaniu po jednym wyniku na wiersz. Jest to szczególne zachowanie. Nie odpowiada ono temu, co uzyskasz po wywołaniu metody `to_s` dla tablicy, czyli reprezentacji łańcuchowej tablicy w nawiasach kwadratowych.

Zajmując się reprezentacjami łańcuchowymi obiektów, sprawdźmy kilka powiązanych metod. Być może odchodzimy trochę od kategorii metod `to_*`, ale są to wszystkie te metody, które generują łańcuchy na bazie obiektów. Z tego też powodu uwzględnianie ich jest właściwe.

### PRZEZNACZONA DO PRZESŁANIANIA: METODA INSPECT

Każdy obiekt języka Ruby (i tym razem z wyjątkiem instancji klasy `BasicObject`) zawiera metodę `inspect`. Domyślnie, jeśli dana klasa nie przesłania metody `inspect`, łańcuch tej metody stanowi minizrzut ekranu zawierający położenie w pamięci dla obiektu:

```
>> Object.new.inspect
=> "#<Object:0x007fe24a292b68>"
```

W rzeczywistości narzędzie irb używa metody `inspect` dla każdej wyświetlanej wartości, dlatego łańcuchy tej metody dla różnych obiektów możesz zobaczyć nawet bez jawnego wywoływania metody `inspect`:

```
>> Object.new
=> #<Object:0x007f91c2a8d1e8>
>> "abc"
=> "abc"
>> [1,2,3]
=> [1, 2, 3]
>> /wyrażenie regularne/
=> /wyrażenie regularne/
```

Jeśli potrzebujesz przydatnego łańcucha metody `inspect` dla własnych klas, musisz ją jawnie zdefiniować:

```
class Person
  def initialize(name)
    @name = name
  end
def inspect
    @name
  end
end
david = Person.new("Dawid")
puts david.inspect # Dane wyjściowe: Dawid
```

Zauważ, że przesłanianie metody `to_s` i przesłanianie metody `inspect` to dwie różne rzeczy. Przed pojawieniem się języka Ruby 2 metoda `inspect` bazowała na metodzie `to_s`, dlatego przez przesłonięcie jednej z nich możliwe było przesłonięcie obu. Uległo to już zmianie.

Inna, rzadziej używana metoda `display` generuje i wyświetla reprezentację łańcuchową obiektu.

### UŻYCIE METODY DISPLAY

Z metodą `display` nie spotkasz się zbyt często. Występuje ona ostatecznie tylko raz we wszystkich plikach programów Ruby w całej bibliotece standardowej (metoda `inspect` pojawia się 160 razy). Metoda `display` jest specjalizowaną metodą danych wyjściowych.

Metoda pobiera argument, którym jest zapisywalny strumień wyjściowy w postaci obiektu wejścia-wyjścia języka Ruby. Domyślnie metoda korzysta ze standardowego strumienia wyjściowego `STDOUT`:

```
>> "Witaj".display
Witaj=> nil
```

Zauważ, że w przeciwieństwie do metody `puts` metoda `display`, podobnie jak metoda `print`, nie wstawia automatycznie znaku nowego wiersza. Z tego właśnie powodu instrukcja `=> nil` jest uruchamiana w jednym wierszu z danymi wyjściowymi.

Dane wyjściowe metody `display` możesz przekierować przez zapewnienie jako argumentu na przykład uchwytu otwartego pliku:

```
>> fh = File.open("/tmp/display.out", "w")
=> #<File:/tmp/display.out>
>> "Witaj".display(fh)  ←❶
=> nil
>> fh.close
=> nil
>> puts(File.read("/tmp/display.out"))  ←❷
Witaj
```

Łańcuch `"Witaj"` jest „wyświetlany” bezpośrednio w pliku ❶, co potwierdzamy przez wczytanie i wyświetlenie jego zawartości ❷.

Opuśćmy w tym miejscu „terytorium” łańcuchów i przyjrzyjmy się temu, jak techniki konwersji działają w przypadku klasy `Array`.

### 7.4.2. Konwersja tablic za pomocą metody to_a i operatora *

Metoda `to_a` dokonująca konwersji do postaci tablicy, jeśli zostanie zdefiniowana, zapewnia reprezentację tablicową obiektów. Jedną z najbardziej widocznych cech tej metody jest to, że automatycznie wiąże się ona z operatorem gwiazdki *. Operator realizuje działanie polegające na „odpakowaniu" swojego operandu do postaci jego składników, które są elementami reprezentacji łańcuchowej operandu.

Wcześniej zaprezentowano już użycie operatora gwiazdki na listach parametrów metody, w przypadku których oznacza ona parametr pobierający argumenty opcjonalne do tablicy. W bardziej ogólnym przypadku gwiazdka powoduje przekształcenie dowolnej tablicy lub obiektu, który odpowiada na metodę `to_a`, w równoważną listę uproszczoną.

Pojęcie *listy uproszczonej* oznacza kilka identyfikatorów lub obiektów literału oddzielonych przecinkami. W określonych kontekstach takie listy są poprawne składniowo. Na przykład listę uproszczoną możesz umieścić w nawiasach kwadratowych konstruktora tablicy literałów:

```
[1,2,3,4,5]
```

Choć jest to subtelna różnica, zapis między nawiasami kwadratowymi nie ma postaci tablicy. Jest to lista, a tablica, dzięki nawiasom kwadratowym, jest tworzona przy użyciu listy.

Gwiazdka powoduje swego rodzaju efekt usuwania nawiasów kwadratowych lub eliminowania tablicy. To, co na początku jest tablicą, staje się listą. Możesz się o tym przekonać, jeśli na bazie tablicy ze znakiem gwiazdki utworzysz tablicę:

```
>> array = [1,2,3,4,5]
=> [1, 2, 3, 4, 5]
>> [*array]
=> [1, 2, 3, 4, 5]
```

Tablica w obiekcie `array` została, jak to można określić, zdegradowana do postaci listy uproszczonej, co zawdzięczamy gwiazdce. Porównaj to z tym, co ma miejsce, gdy gwiazdka nie zostanie zastosowana:

```
>> [array]
=> [[1, 2, 3, 4, 5]]
```

W tym przypadku lista, na bazie której konstruowana jest nowa tablica, zawiera jeden element, czyli obiekt `array`. Obiekt ten nie był sprawdzany pod kątem jego elementów wewnętrznych, tak jak to miało miejsce w przykładzie z gwiazdką.

Konsekwencją tego jest możliwość użycia gwiazdki przed argumentem metody w celu przekształcenia jej z tablicy w listę. Jest to wykonywane w sytuacjach, gdy w tablicy znajdują się obiekty wymagające wysłania do metody oczekującej listy argumentów:

```
def combine_names(first_name, last_name)
  first_name + " " + last_name
end
names = ["Dawid", "Nowak"]
puts combine_names(*names)  ←——— Dane wyjściowe: Dawid Nowak
```

Jeśli nie użyjesz gwiazdki eliminującej tablicę, metodzie zostanie przekazany tylko jeden argument w postaci tablicy. W tej sytuacji metoda nie będzie „zadowolona".

Zajmijmy się liczbami.

### 7.4.3. Konwersja liczb za pomocą metod to_i i to_f

W przeciwieństwie do niektórych języków programowania, takich jak Perl, w języku Ruby nie ma miejsca automatyczne przekształcanie łańcuchów w liczby lub odwrotnie. Nie jest możliwa następująca operacja:

```
>> 1 + "2"  ←——— Komunikat o błędzie TypeError: String can't be coerced into Fixnum
```

Wynika to stąd, że interpreter języka Ruby nie potrafi dodawać do siebie łańcucha i liczby całkowitej. Jeśli zastosujesz poniższy kod, będziesz zaskoczony:

```
print "Wprowadź liczbę: "
n = gets.chomp
puts n * 100
```

Ujrzysz wersję łańcuchową liczby wyświetloną 100 razy (taki wynik informuje również o tym, że interpreter języka Ruby umożliwia mnożenie łańcucha, który jednak zawsze traktowany jest jak łańcuch, nawet jeśli składa się z cyfr). Jeśli żądana jest liczba, musisz jawnie przekształcić w nią łańcuch:

```
n = gets.to_i
```

Jak się okaże, jeśli eksperymentujesz z przekształcaniem łańcuchów w liczby całkowite (z łatwością możesz to zrealizować w sesji narzędzia irb za pomocą takich wyrażeń jak `"witaj".to_i`), wartość wykonanej za pomocą metody `to_i` konwersji łańcuchów, które nie mają żadnego sensownego odpowiednika w postaci liczb całkowitych (w tym łańcucha `"Witaj"`), zawsze wynosi `0`. Jeśli łańcuch rozpoczyna się od cyfr, ale nie jest złożony wyłącznie z nich (np. `"123witaj"`), części łańcucha pozbawione cyfr są ignorowane, a konwersja jest przeprowadzana tylko dla cyfr znajdujących się na początku łańcucha.

Konwersja do postaci liczby zmiennoprzecinkowej wykonywana przy użyciu metody `to_f` zapewnia, zgodnie z oczekiwaniami, zmiennoprzecinkowy odpowiednik dowolnej liczby całkowitej. Reguły dotyczące niezgodnych znaków są podobne do reguł obowiązujących w przypadku przekształceń łańcuchów w liczby całkowite: `"witaj".to_f` odpowiada `0.0`, natomiast po konwersji `"1.23witaj".to_f` uzyskuje się `1.23`. Jeśli metodę `to_f` wywołasz dla liczby zmiennoprzecinkowej, w rezultacie uzyskasz z powrotem tę samą liczbę. Podobnie wywołanie metody `to_i` dla liczby całkowitej powoduje zwrócenie tej liczby.

Jeśli reguły konwersji dotyczące łańcuchów wydają się być trochę niedbałe (w sytuacji, gdy chcesz, aby niepowodzeniem kończyła się konwersja łańcuchów, takich jak `"-5xyz"`, do postaci liczb całkowitych lub liczb zmiennoprzecinkowych), można skorzystać z kilku bardziej wymagających technik konwersji.

**BARDZIEJ WYMAGAJĄCE KONWERSJE ZA POMOCĄ METOD INTEGER I FLOAT**

Język Ruby zapewnia metody `Integer` i `Float` (zgadza się, choć przypominają one stałe, są metodami z nazwami pokrywającymi się z nazwami klas, do postaci których dokonują konwersji). Metody te przypominają odpowiednio metody `to_i` i `to_f`, ale są trochę bardziej wymagające: jeśli przekażesz im cokolwiek, co jest niezgodne z typem docelowym konwersji, metody zgłoszą wyjątek:

```
>> "123abc".to_i
=> 123
>> Integer("123abc")
ArgumentError: invalid value for Integer(): "123abc"
>> Float("3")
=> 3.0
>> Float("-3")
=> -3.0
>> Float("-3xyz")
ArgumentError: invalid value for Float(): "-3xyz"
```

Zauważ, że przekształcanie z liczby całkowitej w liczbę zmiennoprzecinkową jest dozwolone. Przyczyną problemu są litery.

Jeśli wymagasz ścisłości odnośnie do tego, co jest konwertowane, a co odrzucane, metody `Integer` i `Float` mogą okazać się pomocne.

**Porównanie konwersji i rzutowania typów**

Gdy wywołujesz metody takie jak `to_s`, `to_i` i `to_f`, wynikiem jest nowy obiekt (lub odbiorca, jeśli obiekt jest konwertowany do postaci własnej klasy). Nie do końca odpowiada to rzutowaniu typów w języku C oraz innych językach. Nie używasz obiektu jako łańcucha lub liczby całkowitej. Od obiektu żądasz udostępnienia drugiego obiektu, który mu odpowiada (jak to można ująć) w jednej z tych postaci.

Różnica między konwersją i rzutowaniem typów związana jest z kilkoma ważnymi aspektami „serca" języka Ruby. W pewnym sensie wszystkie obiekty nieustannie same wykonują rzutowanie typów. Każdorazowo w momencie wywoływania metody dla obiektu żądasz od niego, aby zachowywał się jak konkretny typ. Odpowiednio „typ" obiektu w rzeczywistości jest agregatem wszystkiego, co może on zrealizować w określonym momencie.

Najbliżej tradycyjnego rzutowania typów (i nie można wcale uznać tego za bardzo bliskie) język Ruby jest w przypadku metod konwersji przyjmujących role, które opisano w punkcie 7.4.4.

Wróćmy do rodziny metod konwertujących `to_*`. Oprócz prostych metod dokonujących konwersji obiektów język Ruby oferuje kilka metod `to_*`, które cechują się niewielką dodatkową „inteligencją" związaną z określaniem tego, co ma być realizowane z wykorzystaniem ich wartości zwracanej.

### *7.4.4. Metody z rodziny to_* przyjmujące role*

W pewnym sensie wbrew naturze programowania w języku Ruby jest zbytnie martwienie się o to, do jakiej klasy należy obiekt. Istotne jest jedynie to, co obiekt umożliwia, czyli jakie metody może wykonać.

Jednak w kilku sytuacjach obejmujących klasy podstawowe duża uwaga jest zwracana na klasę obiektów. Nie traktuj tego jako planu odnoszącego się do sposobu postrze-

gania obiektów przez język Ruby. Są to bardziej względy praktyczne dające możliwość „wkroczenia" do świata podstawowych obiektów w sposób, który po fazie początkowej pozwoli w mniejszym stopniu zastanawiać się nad przynależnością obiektów do klas.

### PRZYJMOWANIE ROLI ŁAŃCUCHA ZA POMOCĄ METODY TO_STR

Aby wyświetlić obiekt, możesz zdefiniować dla niego metodę `to_s` lub zastosować dowolne jej działanie, jakie zostało zapewnione przez klasę obiektu. Co jednak będzie, gdy obiekt musi *być* łańcuchem?

Odpowiedzią na to pytanie jest zdefiniowanie dla obiektu metody `to_str`. Jej reprezentacja okazuje się przydatna w momencie wywoływania podstawowej metody, która wymaga łańcucha jako swojego argumentu.

Klasycznym przykładem jest dodawanie łańcuchów. W języku Ruby możliwe jest dodanie do siebie dwóch łańcuchów w celu utworzenia trzeciego:

```
>> "Witaj, " + "przyjacielu."
=> "Witaj, przyjacielu."
```

Jeśli do łańcucha spróbujesz dodać coś, co nie jest łańcuchem, zostanie wyświetlony błąd:

```
>> "Witaj, " + 10
TypeError: no implicit conversion of Float into String
```

W tym przypadku pojawia się metoda `to_str`. Jeśli obiekt odpowiada na metodę `to_str`, jego reprezentacja tej metody będzie używana, gdy obiekt pełni rolę argumentu metody `String#+`.

Poniżej zaprezentowano przykład prostej klasy `Person`. Metoda `to_str` opakowuje metodę `name`:

```
class Person
  attr_accessor :name
  def to_str
    name
  end
end
```

Jeśli utworzysz obiekt `Person` i dodasz go do łańcucha, metoda `to_str` wprowadzi łańcuch `name`:

```
david = Person.new
david.name = "Dawid"
puts "dawid ma imię " + david + "."   ←——— Dane wyjściowe: dawid ma imię Dawid.
```

Konwersja realizowana przez metodę `to_str` jest też stosowana w przypadku argumentów metody << (dołączanie do łańcucha). Podobnie jak łańcuchy tablice dysponują metodą konwersji, która przyjmuje rolę.

### PRZYJMOWANIE ROLI TABLICY ZA POMOCĄ METODY TO_ARY

Obiekty mogą udawać tablice, jeśli zawierają metodę `to_ary`. Jeśli istnieje taka metoda, jest ona wywoływana w obiekcie w sytuacjach, gdy tablica, i tylko tablica, zostanie na przykład użyta w operacji łączenia tablic.

Oto kolejna implementacja klasy `Person`, w której rolę tablicy pełni tablica zawierająca trzy atrybuty osoby:

```
class Person
  attr_accessor :name, :age, :email
  def to_ary
    [name, age, email]
  end
end
```

Dołączanie obiektu Person do tablicy powoduje dodanie do tablicy docelowej wartości odpowiadających imieniu, wiekowi i adresowi e-mail:

```
david = Person.new
david.name = "Dawid"
david.age = 55
david.email = "dawid@gdziekolwiek"
array = []
array.concat(david)
p array      ←——— Dane wyjściowe: ["Dawid", 55, "dawid@gdziekolwiek"].
```

Podobnie jak metoda `to_str` metoda `to_ary` zapewnia obiektowi możliwość pełnienia roli obiektu konkretnej klasy podstawowej. Jak to zwykle bywa w przypadku języka Ruby, od Ciebie zależy praktyczne wykorzystywanie konwencji. Możliwe jest na przykład utworzenie metody `to_ary`, która realizuje coś innego niż zwracanie tablicy. Jednak prawie na pewno uzyskasz komunikat o błędzie przy próbie użycia jej, ponieważ w przypadku metody `to_ary` interpreter języka Ruby oczekuje tablicy. Jeśli więc zamierzasz skorzystać z metod `to_*` przyjmujących role, pamiętaj o przestrzeganiu reguł języka Ruby.

Zajmiemy się teraz zagadnieniem stanów i obiektów boolowskich w języku Ruby, które zostało już poruszone, ale zasługuje na szersze omówienie.

## 7.5. Stany i obiekty boolowskie oraz obiekt nil

Dla każdego wyrażenia w języku Ruby określany jest obiekt, a każdy obiekt ma wartość boolowską *prawda* lub *fałsz*. Co więcej, `true` i `false` to obiekty. Zagadnienie to nie jest tak zawiłe, na jakie wygląda. Jeśli `true` i `false` nie byłyby obiektami, następujące czyste wyrażenie boolowskie:

```
100 > 80
```

nie miałoby żadnego obiektu, który zostałby *dla niego* określony (ponieważ > to metoda, musi ona zwrócić obiekt).

W wielu sytuacjach, w których chcesz otrzymać wartość prawda lub fałsz (np. w przypadku instrukcji `if` lub porównania dwóch liczb), nie jest konieczne bezpośrednie modyfikowanie tych obiektów specjalnych. W takich sytuacjach prawdę i fałsz możesz traktować jako *stany*, a nie jako obiekty.

Przyjrzymy się obiektom `true` i `false` zarówno jako stanom, jak i obiektom specjalnym, uwzględniając także obiekt specjalny `nil`.

### 7.5.1. Obiekty true i false jako stany

W sensie logicznym lub boolowskim każde wyrażenie w języku Ruby jest prawdziwe lub fałszywe. Najlepszym sposobem na zrozumienie tego jest posługiwanie się rozumowaniem typowym dla instrukcji warunkowych. Dla każdego wyrażenia *e* w języku Ruby możesz wykonać następującą instrukcję:

```
if e
```

Instrukcja będzie zrozumiała dla interpretera języka Ruby.

W przypadku wielu wyrażeń test warunkowy to ostateczność. Pouczające może być jednak wypróbowanie go dla różnych wyrażeń, co prezentuje listing 7.1.

**Listing 7.1. Testowanie wartości boolowskiej wyrażeń za pomocą konstrukcji if**

```
if (class MyClass; end)                    ←❶
  puts "Pusta definicja klasy jest prawdziwa!"
else
  puts "Pusta definicja klasy nie jest prawdziwa!"
end
if (class MyClass; 1; end)                 ←❷
  puts "Definicja klasy z liczbą 1 jest prawdziwa!"
else
  puts "Definicja klasy z liczbą 1 nie jest prawdziwa!"
end
if (def m; return false; end)              ←❸
  puts "Definicja metody jest prawdziwa!"
else
  puts "Definicja metody nie jest prawdziwa!"
end
if "string"                                ←❹
  puts "Łańcuchy okazują się być prawdziwe!"
else
  puts "Łańcuchy nie okazują się być prawdziwe!"
end
if 100 > 50                                ←❺
  puts "100 jest większe niż 50!"
else
  puts "100 nie jest większe niż 50!"
end
```

Oto dane wyjściowe kodu z listingu 7.1 (z pominięciem ostrzeżenia dotyczącego użycia literału łańcuchowego w instrukcji warunkowej):

```
Pusta definicja klasy nie jest prawdziwa!
Definicja klasy z liczbą 1 jest prawdziwa!
Definicja metody jest prawdziwa!
Łańcuchy okazują się być prawdziwe!
100 jest większe niż 50!
```

Jak widać, puste definicje klasy ❶ nie są prawdziwe. Dla niepustych definicji klasy określana jest taka sama wartość jak ostatnia wartość, jaką one zawierają ❷ (w tym przykładzie jest to liczba 1). Definicje metody są prawdziwe ❸ (jeśli nawet wywołanie metody spowodowałoby zwrócenie wartości fałszywej). Łańcuchy są prawdziwe ❹ (nie przejmuj się ostrzeżeniem dotyczącym użycia literału łańcuchowego w instrukcji

warunkowej). Liczba 100 jest większa niż 50 ❺. Tej prostej techniki opartej na instrukcji `if` możesz użyć do eksploracji wartości boolowskiej dowolnego wyrażenia języka Ruby.

Przykłady z instrukcją `if` pokazują, że każde wyrażenie w języku Ruby jest prawdziwe lub fałszywe w tym sensie, że przechodzi lub nie przechodzi testu z instrukcją `if`. Jednak w przykładach tych nie jest widoczne, co jest określane dla wyrażeń. Właśnie to jest sprawdzane przez test z instrukcją `if`, który określa wartość wyrażenia (np. `class MyClass; end`), a następnie kontynuuje działanie na podstawie tego, czy prawdziwa jest wartość uzyskana w wyniku operacji określania.

Aby dowiedzieć się, jakie wartości są zwracane przez wyrażenia, dla których sprawdzano wartość prawda, możesz skorzystać z sesji narzędzia irb w następujący sposób:

```
>> class MyClass; end          ←❶
=> nil
>> class MyClass; 1; end       ←❷
=> 1
>> def m; return false; end    ←❸
=> :m
>> "łańcuch literał!"          ←❹
=> "łańcuch literał!"
>> 100 > 50                    ←❺
=> true
```

Dla pustej definicji klasy ❶ określany jest obiekt `nil`, który jest obiektem specjalnym (omówiono go w punkcie 7.5.3). Na razie o obiekcie tym musisz wiedzieć tylko tyle, że ma wartość boolowską fałsz (co możesz stwierdzić na podstawie zachowania klauzul `if` z listingu 7.1, które były z tym powiązane).

Dla definicji klasy zawierającej liczbę 1 ❷ określana jest wartość 1, ponieważ dla każdego bloku definicji klasy określane jest ostatnie znajdujące się w nim wyrażenie lub obiekt `nil` (jeśli blok jest pusty).

Dla definicji metody określany jest symbol `:m` ❸, który reprezentuje nazwę właśnie zdefiniowanej metody.

Dla łańcucha `literał` ❹ określany jest on sam. Jest to obiekt literału, który podczas określania dla niego wartości nie musi być przetwarzany do jakiejś innej postaci. Wartością tego obiektu jako wyrażenia jest on sam.

Dla wyrażenia porównania `100 > 50` ❺ określana jest wartość `true`. Nie jest to tylko coś, co ma wartość boolowską prawda, lecz obiekt `true`. Ma on wartość boolowską prawda. Wraz z obiektem `false` obiekt `true` pełni jednak szczególną rolę w obszarze prawdy i fałszu, a także decyduje o tym, jak prawda i fałsz są reprezentowane w języku Ruby.

### *7.5.2. Obiekty true i false*

Obiekty boolowskie `true` i `false` są obiektami specjalnymi, z których każdy jest wyłącznie instancją klasy specjalnie dla niego utworzonej. Są to odpowiednio klasy `TrueClass` i `FalseClass`. W następujący sposób możesz zażądać od obiektów `true` i `false` przekazania nazw ich klas:

```
puts true.class      ←——— Dane wyjściowe: TrueClass
puts false.class     ←——— Dane wyjściowe: FalseClass
```

Terminy `true` i `false` to słowa kluczowe. Nie możesz używać ich jako nazw zmiennych ani metod. Są one zarezerwowane do wyłącznego użytku w języku Ruby.

Obiekty `true` i `false` możesz przekazywać, przypisywać do zmiennych, a także sprawdzać tak jak dowolny inny obiekt. Oto sesja narzędzia irb, w ramach której sprawdzono możliwości obiektu `true` jako obiektu języka Ruby:

```
>> a = true
=> true
>> a = 1 unless a
=> nil
>> a
=> true
>> b = a
=> true
```

Czasami `true` i `false` będą używane jako argumenty metod. Jeśli na przykład klasa ma pokazać swoje metody instancji z wyłączeniem zdefiniowanych w klasach nadrzędnych, w żądaniu możesz podać argument `false`:

```
>> String.instance_methods(false)
```

Problem związany z argumentami boolowskimi polega na tym, że bardzo trudno zapamiętać, do czego one służą. Są one raczej tajemnicze. Oznacza to, że najlepiej unikać ich we własnym kodzie, chyba że wystąpi sytuacja, w której bardzo wyraźnie można odróżnić obiekt `true` lub `false`.

Podsumujmy kwestię prawdy i fałszu w języku Ruby, porównując ze sobą stany i wartości boolowskie.

### PRAWDA I FAŁSZ: PORÓWNANIE STANÓW Z WARTOŚCIAMI

Jak już wiesz, w sensie boolowskim każde wyrażenie języka Ruby jest prawdziwe lub fałszywe (co wskazuje test z instrukcją `if`). Istnieją również obiekty o nazwach `true` i `false`. Takie podwójne stosowanie terminologii powiązanej z prawdą i fałszem stanowi czasami źródło nieporozumień: gdy stwierdzasz, że coś jest prawdziwe, nie zawsze jasne jest, czy rozumiesz przez to prawdę boolowską, czy obiekt `true`.

Pamiętaj o tym, że każde wyrażenie ma wartość boolowską, w tym wyrażenia `true` i `false`. Złożone może wydać się następujące stwierdzenie: „Obiekt `true` jest prawdziwy". Jednak taki dodatkowy krok umożliwia spójne funkcjonowanie modelu.

Bazując na tej uwadze, a także na niektórych przypadkach zaprezentowanych w tabeli 7.1 w trochę odmiennej postaci, w tabeli 7.3 pokazano odwzorowanie wybranych przykładowych wyrażeń zarówno na wynik operacji określania dla nich wartości, jak i na ich wartość boolowską.

W szczególności zauważ, że łańcuchy puste (dotyczy to także pustych tablic i tablic asocjacyjnych) mają wartość boolowską prawda. `false` i `nil` to jedyne obiekty, które mają wartość boolowską fałsz.

Przejdźmy do obiektu `nil`: pora bliżej przyjrzeć się temu unikatowemu obiektowi.

**Tabela 7.3.** Odwzorowanie przykładowych wyrażeń na wyniki operacji określania dla nich wartości oraz na wartości boolowskie

| Wyrażenie | Obiekt określany dla wyrażenia | Wartość boolowska wyrażenia |
|---|---|---|
| `1` | `1` | Prawda |
| `0` | `0` | Prawda |
| `1+1` | `2` | Prawda |
| `true` | `true` | Prawda |
| `false` | `false` | Prawda |
| `nil` | `nil` | Fałsz |
| `"łańcuch"` | `"łańcuch"` | Prawda |
| `""` | `""` | Prawda |
| `puts "łańcuch"` | `nil` | Fałsz |
| `100 > 50` | `true` | Prawda |
| `x = 10` | `10` | Prawda |
| `def x; end` | `:x` | Prawda |
| `class C; end` | `nil` | Fałsz |
| `class C; 1; end` | `1` | Prawda |

### 7.5.3. Obiekt specjalny nil

Obiekt specjalny `nil` to rzeczywiście obiekt (jest jedyną instancją klasy o nazwie `NilClass`). W praktyce jest to jednak też coś w rodzaju czegoś innego niż obiekt. Wartością boolowską obiektu `nil` jest fałsz, ale jest to dopiero początek jego nieobiektowości.

Obiekt `nil` oznacza brak czegokolwiek. Możesz to zaobserwować w postaci graficznej podczas sprawdzania na przykład wartości zmiennej instancji, która nie została zainicjowana:

```
puts @x
```

Polecenie to wyświetla obiekt `nil` (gdy spróbujesz użyć polecenia dla zmiennej lokalnej, zostanie wygenerowany błąd; zmienne lokalne nie są automatycznie inicjowane za pomocą żadnej wartości, nawet obiektu `nil`). `nil` jest też wartością domyślną nieistniejących elementów kontenera i obiektów kolekcji. Jeśli na przykład utworzysz tablicę z trzema elementami, a następnie spróbujesz uzyskać dostęp do dziesiątego elementu (na pozycji 9. indeksu, ponieważ indeksowanie tablicy rozpoczyna się od zera), okaże się, że wynikiem jest obiekt `nil`:

```
>> ["jeden","dwa","trzy"][9]
=> nil
```

Czasami obiekt `nil` jest trudny do zrozumienia. Choć obiekt `nil` odnosi się do braku i nieobecności elementu, sam istnieje i odpowiada na wywołania metod tak jak inne obiekty:

```
>> nil.to_s
=> ""
>> nil.to_i
```

```
=> 0
>> nil.object_id
=> 8
```

Wynikiem konwersji obiektu `nil` za pomocą metody `to_s` jest łańcuch pusty (`""`). Reprezentacją liczbową obiektu `nil` jest zero. Identyfikator tego obiektu to 8 (obiekt `nil` nie jest w żaden szczególny sposób powiązany z tą liczbą, która wydaje się być po prostu przydzielona do niego jako jego identyfikator).

Stwierdzenie, że obiekt `nil` jest pusty, nie jest dokładne, ponieważ wskazywałoby to, że obiekt ma właściwości i wielkość, tak jak liczba lub kolekcja, a tak nie ma być. Próba zrozumienia obiektu `nil` może sprowadzić na „cierniste" terytorium natury filozoficznej. `nil` możesz traktować jak istniejący obiekt wyposażony w zestaw niezbędnych metod, który jednak ma za zadanie reprezentowanie braku oraz stanu nieznanego.

Wracając do punktu wyjścia, pamiętaj o tym, że obiekt `nil` ma wartość boolowską fałsz. `nil` i `false` to jedyne obiekty z taką wartością. Nie są to jedyne tego rodzaju *wyrażenia*. Wyrażenie `100 < 50` ma wartość boolowską fałsz, gdyż jest dla niego określany obiekt `false`. `nil` i `false` są jednak jedynymi *obiektami* w języku Ruby z wartością boolowską fałsz. Wszystkie inne obiekty tego języka (np. liczby, łańcuchy, instancje klasy `MyCoolClass`) mają wartość boolowską prawda. Bezpośrednio testowane wszystkie z powodzeniem przechodzą test z użyciem instrukcji `if`.

Wartości boolowskie i testowanie zapewniają łącznik z następnym zagadnieniem, czyli porównaniami obiektów. Przyjrzymy się testom obejmującym dwa obiekty oraz sposoby określania, czy są one równe, a jeśli nie, czy mogą być uszeregowane jako większe/mniejsze z ustaleniem kryteriów bazowych.

## 7.6. Porównanie dwóch obiektów

Obiekty języka Ruby są tworzone w taki sposób, że umożliwiają za pomocą dowolnej spośród kilku metod porównywanie ich z innymi obiektami pod kątem równości i/lub uporządkowania. Testy równości są najczęściej stosowanymi testami porównującymi, dlatego zaczniemy od nich. Później przyjrzymy się wbudowanemu modułowi języka Ruby o nazwie `Comparable`, który pozwala w szybki sposób przekazać używanym klasom i obiektom informacje o operacjach porównywania. W tym celu moduł jest stosowany przez wiele wbudowanych klas języka Ruby.

### 7.6.1. Testy równości

Wewnątrz klasy `Object` wszystkie metody testu równości realizują to samo: informują o tym, czy dwa obiekty są dokładnie takie same. Oto przykład użycia takich metod:

```
>> a = Object.new
=> #<Object:0x00000101258af8>
>> b = Object.new
=> #<Object:0x00000101251d70>
>> a == a
=> true
>> a == b
=> false
>> a != b
```

```
=> true
>> a.eql?(a)
=> true
>> a.eql?(b)
=> false
>> a.equal?(a)
=> true
>> a.equal?(b)
=> false
```

Wszystkie trzy metody pozytywnego testu równości (==, `eql?` i `equal?`) zapewniają w przykładach identyczne wyniki: w przypadku testu porównującego `a` z `a` wynikiem jest prawda. Z kolei dla testu porównującego `a` z `b` wynikiem jest fałsz (metoda `!=` negatywnego testu równości lub testu nierówności stanowi odwrotność metody ==; okazuje się, że jeśli zdefiniujesz ==, obiekty automatycznie uzyskają metodę `!=`). Istnieje mnóstwo sposobów na określenie, że `a` jest równe `a`, lecz nie `b`.

Nie ma jednak większego sensu w stosowaniu trzech testów realizujących to samo. W klasach innych niż `Object` metody == i/lub `eql?` są zwykle ponownie definiowane w celu wykonywania konkretnej pracy dla różnych obiektów. Metoda `eql?` jest zwykle niewykorzystywana, dlatego zawsze możesz użyć jej do sprawdzenia, czy dwa obiekty są dokładnie tym samym obiektem.

Poniżej zamieszczono przykład z łańcuchami. Zauważ, że prawda występuje dla nich w przypadku metod == i `eql?`, a fałsz dla metody `equal?`:

```
>> string1 = "tekst"
=> "tekst"
>> string2 = "tekst"
=> "tekst"
>> string1 == string2
=> true
>> string1.eql?(string2)
=> true
>> string1.equal?(string2)
=> false
```

Co więcej, język Ruby zapewnia zestaw narzędzi służących do porównywania obiektów. Porównania nie zawsze dotyczą jedynie równości. Przyjrzymy się teraz temu, jak testy równości i ich ponowne definicje wpasowują się w ogólny obraz porównań.

### 7.6.2. Porównania i moduł Comparable

== to najczęściej ponownie definiowana metoda testu równości, którą spotkasz najczęściej. Metoda jest częścią większej rodziny metod testu równości, a ponadto należy do rodziny metod porównujących, które obejmują następujące metody: ==, `!=`, >, <, >= i <=.

Nie każda klasa obiektu wymaga wszystkich tych metod lub powinna je mieć (trudno wyobrazić sobie, co miałby oznaczać fakt, że jeden obiekt `Rower` jest równy drugiemu lub większy; przekładnię zębatą?). Jednak w przypadku klas wymagających pełnej funkcjonalności związanej z porównywaniem język Ruby zapewnia wygodny sposób, który na to pozwala. Aby obiekty klasy `MyClass` dysponowały pełnym zestawem metod porównujących, musisz jedynie wykonać następujące kroki:

1. Dołącz moduł o nazwie `Comparable` (wchodzi w skład języka Ruby) do klasy `MyClass`.
2. Zdefiniuj metodę porównującą o nazwie `<=>` jako metodę instancji w klasie `MyClass`.

Metoda porównująca `<=>` (zwykle określana mianem *operatora* lub *metody „statku kosmicznego”*) stanowi sedno sprawy. W obrębie tej metody definiujesz to, co uważasz za *mniejsze niż*, *równe* i *większe niż*. Gdy to nastąpi, interpreter języka Ruby będzie dysponować wszystkim, co jest niezbędne do zapewnienia odpowiednich metod porównujących.

Dla przykładu załóżmy, że przyjmujesz oferty dotyczące wykonania zadania, a ich śledzenie może ułatwić użycie skryptu Ruby. Uznajesz, że przydatna może być możliwość porównania dwóch dowolnych obiektów `Bid` na podstawie atrybutu `estimate` z wykorzystaniem prostych operatorów porównania, takich jak `>` i `<`. *Większe niż* oznacza żądanie większej kwoty, a *mniejsze niż* — mniejszej kwoty.

Listing 7.2 prezentuje pierwszą prostą wersję klasy `Bid`.

**Listing 7.2. Przykład klasy dołączającej moduł Comparable**

```
class Bid
  include Comparable
  attr_accessor :estimate
  def <=>(other_bid)                    ←❶
    if self.estimate < other_bid.estimate
      -1
    elsif self.estimate > other_bid.estimate
      1
    else
      0
    end
  end
end
```

Metoda „statku kosmicznego” ❶ złożona jest z kaskady instrukcji `if/elsif/else`. Zależnie od tego, jaka gałąź jest wykonywana, metoda zwraca liczbę ujemną (zgodnie z konwencją `-1`), liczbę dodatnią (zgodnie z konwencją `1`) lub `0`. Te trzy wartości zwracane są predefiniowanymi sygnałami dla interpretera języka Ruby. Metoda `<=>` musi zwrócić jedną z tych trzech wartości każdorazowo, gdy zostanie wywołana. Wartości te zawsze oznaczają odpowiednio: mniejsze niż, równe i większe niż.

Możesz skrócić tę metodę. Oszacowania ofert są liczbami zmiennoprzecinkowymi lub liczbami całkowitymi (drugi wariant dotyczy sytuacji, w której nieistotna jest część groszowa kwoty lub przechowywane kwoty są wyrażone w groszach, a nie w złotych). W języku Ruby możliwe jest porównywanie ze sobą liczb, w tym liczb całkowitych ze zmiennoprzecinkowymi. A zatem metoda `<=>` klasy `Bid` może bazować na istniejących metodach `<=>` klas `Integer` i `Float`. Oto przykład:

```
def <=>(other_bid)
  self.estimate <=> other_bid.estimate
end
```

W tej wersji metody „statku kosmicznego" asekurujemy się. Oznacza to, że aby dowiedzieć się, jak wypada porównanie ze sobą dwóch ofert, trzeba poruszyć kwestię wartości szacunkowych dwóch ofert, a następnie takim porównaniem posłużyć się jako bazą do porównywania ofert.

Korzyścią wynikającą ze zdefiniowania operatora „statku kosmicznego" i uwzględnienia modułu `Comparable` jest możliwość późniejszego użycia całego zestawu metod porównujących dla par obiektów. W przytoczonym przykładzie oferta `bid1` wygrywa przetarg, ponieważ ma cenę niższą od (jest to identyfikowane przez metodę <) ceny oferty `bid2`:

```
>> bid1 = Bid.new
=> #<Bid:0x000001011d5d60>
>> bid2 = Bid.new
=> #<Bid:0x000001011d4320>
>> bid1.estimate = 100
=> 100
>> bid2.estimate = 105
=> 105
>> bid1 < bid2
=> true
```

Metoda < (wraz z metodami >, >=, <=, ==, `!=` i `between?`) jest definiowana na warunkach metody <=> w obrębie modułu `Comparable` (`b.between?(a,c)` określa, czy `b > a`, a także czy `b < c`).

Wszystkie klasy numeryczne języka Ruby obejmują moduł `Comparable`, a ponadto zawierają definicję metody <=>. To samo dotyczy klasy `String`. Łańcuchy mogą być porównywane z wykorzystaniem pełnego zestawu możliwości oferowanych przez metody i operatory tego modułu. Moduł `Comparable` to przydatne narzędzie, które zasadniczo w zamian za jedną definicję metody zapewnia wiele funkcji.

Zajmiemy się teraz zagadnieniem inspekcji obiektów w czasie wykonywania kodu. Aby być w zgodzie z „duchem" tego rozdziału, przyjrzymy się technikom wystarczającym do tego, aby zachęcić do przeczytania większości materiału z reszty książki. Miej jednak świadomość tego, że w ostatnim rozdziale 15. ponownie zajmiemy się między innymi inspekcją obiektów w czasie wykonywania kodu. A zatem niniejsze omówienie możesz uznać za pierwsze, lecz nie ostatnie, które będzie dotyczyć tego zagadnienia.

## 7.7. Inspekcja możliwości obiektów

Terminy *inspekcja* i *introspekcja* odnoszą się do różnych sposobów umożliwiających uzyskanie od obiektów języka Ruby informacji o nich w czasie ich istnienia. Większość tego, o czym wcześniej wspomniano na temat wymuszania udostępniania przez obiekty swoich reprezentacji łańcuchowych, może zostać opisana jako inspekcja. W tym podrozdziale zajmiemy się innym rodzajem introspekcji dotyczącej wykonywania kodu, a mianowicie technikami uzyskiwania od obiektów informacji o wykonywanych przez nie metodach.

Jak zostanie to przeprowadzone, zależy od obiektu, a także od tego, czego dokładnie szukasz. Każdy obiekt może przekazać informacje o metodach, jakie mogą być

w nim wywoływane, a przynajmniej w momencie skierowania do niego odpowiedniego zapytania. Ponadto obiekty klas i modułów umożliwiają rozdzielenie metod zapewnianych obiektom, które z nich korzystają (w postaci instancji lub przez włączenie modułu).

### 7.7.1. Wyświetlanie listy metod obiektu

Najprostszym i najczęstszym przypadkiem jest określanie, jakie komunikaty są rozpoznawane przez obiekt, czyli jakie metody mogą być w nim wywołane. Język Ruby oferuje zwykle prosty sposób, który to umożliwia. W sesji narzędzia irb wprowadź następujący wiersz:

```
>> "Jestem obiektem String".methods
```

W efekcie zostanie wyświetlona duża tablica nazw metod. Pożądane będzie przynajmniej posortowanie tych nazw, aby możliwe było znalezienie szukanej nazwy:

```
>> "Jestem obiektem String".methods.sort
```

Metoda `methods` działa również z obiektami klas i modułów. Pamiętaj jednak, że metoda ta informuje o tym, jaki odpowiada obiekt (klasa lub moduł), a nie instancje klasy lub obiektów, które korzystają z modułu. Na przykład wpisanie w sesji narzędzia irb następującego wiersza:

```
>> String.methods.sort
```

powoduje wyświetlenie listy metod, na jakie odpowiada obiekt `String` klasy `Class`. Jeśli na tej liście widoczna jest metoda, wiadomo, że może zostać wysłana bezpośrednio do obiektu `String`.

Metody wyświetlone po wywołaniu metody `methods` w obiekcie obejmują jego metody pojedynczego obiektu (utworzone wyłącznie dla tego obiektu), a także wszystkie metody, jakie obiekt może wywołać dzięki dołączeniu co najmniej jednego modułu w dowolnym jego elemencie nadrzędnym. Wszystkie te metody są prezentowane jako równorzędne: lista metod upraszcza ścieżkę wyszukiwania metod, a ponadto informuje jedynie o tym, jakie metody są znane obiektowi, niezależnie od miejsca ich zdefiniowania.

Możesz to zweryfikować, korzystając z sesji narzędzia irb. Poniżej zaprezentowano przykład, w którym metoda pojedynczego obiektu jest dodawana do łańcucha. Jeśli na końcu dołączysz wywołanie metody `str.methods.sort`, okaże się, że metoda `shout` jest teraz wśród metod łańcucha:

```
>> str = "Zwykły, poczciwy łańcuch"
=> "Zwykły, poczciwy łańcuch"
>> def str.shout
>>   self.upcase + "!!!"
>> end
=> nil
>> str.shout
=> "ZWYKŁY, POCZCIWY ŁAŃCUCH!!!"
>> str.methods.sort
```

Wygodnym rozwiązaniem jest zapytanie tylko o metody pojedynczego obiektu:

```
>> str.singleton_methods
=> [:shout]
```

Jeśli dołączysz moduł do klasy za pomocą instrukcji `include`, instancje tej klasy będą o sobie informować jako mające możliwość wywoływania metod instancji z poziomu tego modułu. Co interesujące, ten sam wynik uzyskasz nawet wtedy, gdy moduł dołączysz po tym, jak instancja będzie już istnieć. Rozpocznij nową sesję narzędzia irb (w celu wyczyszczenia pamięci z danych poprzedniego przykładu) i sprawdź poniższy kod. Zamiast wyświetlać wszystkie metody, zostanie użytych kilka bardziej uporządkowanych technik w celu stwierdzenia, czy obiekt `str` ma metodę `shout`:

```
>> str = "Kolejny zwykły, poczciwy łańcuch."
=> "Kolejny zwykły, poczciwy łańcuch."
>> module StringExtras
>>   def shout
>>     self.upcase + "!!!"
>>   end
>> end
=> :shout
>> class String
>>   include StringExtras  <------ Zamienia łańcuchy w łańcuchy z wykrzyknikami
>> end
=> String
>> str.methods.include?(:shout)
=> true
```

Dołączenie modułu ma wpływ na już istniejące łańcuchy, ponieważ w momencie zażądania utworzenia łańcucha z wykrzyknikami obiekt `str` szuka metody `shout` w swojej ścieżce wyszukiwania metod i znajduje ją w module. Dla obiektu łańcucha naprawdę obojętne jest, kiedy i jak moduł został umieszczony w ścieżce wyszukiwania.

Dowolny obiekt może poinformować o znanych sobie metodach. Ponadto obiekty klas i modułów mogą zapewnić informacje na temat udostępnianych metod.

### *7.7.2. Odpytywanie obiektów klas i modułów*

Metoda `instance_methods` to jedna z metod znajdujących się na liście generowanej przez polecenie `String.methods.sort` wpisane w sesji narzędzia irb. Metoda ta informuje o wszystkich metodach instancji zapewnionych instancjom klasy `String`:

```
>> String.instance_methods.sort
```

Wynikowa lista jest taka sama jak lista metod (zaprezentowana przez metodę `methods`) dla dowolnego danego łańcucha (jeśli nie dodano do niego metod pojedynczego obiektu).

Możliwe jest utworzenie podobnego zapytania dotyczącego modułu:

```
>> Enumerable.instance_methods.sort
```

Oprócz zwykłych list metod i list metod instancji w języku Ruby zapewniana jest określona liczba działań, które ułatwiają tworzenie bardziej szczegółowych zapytań.

### 7.7.3. Listy filtrowanych i wybranych metod

Czasem wymagane będzie wyświetlenie metod instancji zdefiniowanych w konkretnej klasie bez konieczności zajmowania się metodami zawartymi w każdym obiekcie. Wiesz już przecież, że używany obiekt ma *te* metody. Metody instancji klasy, z pominięciem metod jej klas nadrzędnych, możesz sprawdzić za pomocą trochę tajemniczej techniki, którą zaprezentowano wcześniej. Polega ona na zapewnieniu argumentu `false` metodzie `instance_methods`:

```
String.instance_methods(false).sort
```

Dzięki temu ujrzysz znacznie mniej metod, ponieważ przeszukiwana jest lista metod wyłącznie zdefiniowanych w klasie `String`. Działanie nie obejmuje metod zdefiniowanych w dowolnej z klas lub modułów nadrzędnych klasy `String`. Takie rozwiązanie zapewnia ograniczony „obraz" metod dostępnych dla obiektów łańcucha, ale przydaje się podczas bardziej szczegółowego określania, jak i gdzie umiejscowiono definicje metod powiązane z danym obiektem.

Inne metody generujące listę metod obejmują następujące (metoda `singleton_methods` została już wcześniej zaprezentowana):

- `obj.private_methods`,
- `obj.public_methods`,
- `obj.protected_methods`,
- `obj.singleton_methods`.

Ponadto klasy i moduły umożliwiają sprawdzenie ich metod instancji:

- `Class.private_instance_methods`,
- `MyClass.protected_instance_methods`,
- `MyClass.public_instance_methods`.

Ostatnia z wymienionych metod (`public_instance_methods`) jest synonimem metody `instance_methods`.

Mechanizmy sprawdzania metod obiektów są rozbudowane. Jak zawsze trzeba jasno określić, czym jest odpytywany obiekt (a dokładniej rzecz biorąc, czy jest to obiekt klasy/modułu, czy „zwykły" obiekt), a także jakie informacje są od niego żądane.

Dotarliśmy do końca naszej sesji wprowadzającej, „pakietu przetrwania" lub przewodnika wprowadzającego do języka Ruby... Jak by tego nie nazwać (można nawet użyć terminu „rozdział 7."!), zaprezentowany materiał odpowiednio przygotowuje nas do bliższego przyjrzenia się kilku ważnym klasom podstawowym, którym poświęcono kilka następnych rozdziałów.

## 7.8. Podsumowanie

W rozdziale zaprezentowano następujące zagadnienia:

- Konstruktory literałów języka Ruby.
- Przekształcanie metod w operatory z wykorzystaniem „lukru" składniowego.
- Metody „destrukcyjne" i metody z wykrzyknikiem w nazwie.

- Metody konwersji (metoda `to_s` i powiązane z nią).
- Metody `inspect` i `display`.
- Wartości i obiekty boolowskie.
- Obiekt specjalny `nil`.
- Porównywanie obiektów i moduł `Comparable`.
- Sprawdzanie metod obiektu.

W rozdziale omówiono kilka zagadnień powiązanych z wieloma klasami i modułami wbudowanymi. Przedstawiono konstruktory literałów języka Ruby, które zapewniają bardziej zwięzłą alternatywę wywoływania metody `new` w przypadku określonych klas wbudowanych. Wyjaśniono również, w jaki sposób w języku Ruby zapewniany jest „lukier" składniowy dla konkretnych nazw metod, w tym dużej liczby metod o nazwach odpowiadających operatorom arytmetycznym.

Przyjrzeliśmy się znaczeniu metod zmieniających własnych odbiorców, które są tworzone przez wiele metod wbudowanych (wiele tych metod to metody ze znakiem `!` na końcu nazwy). Sprawdziliśmy też metody `to_*`, czyli metody wbudowane służące do przeprowadzania konwersji między klasami podstawowymi.

Przedstawiono również kilka ważnych kwestii i technik powiązanych z wartościami boolowskimi (prawda/fałsz) oraz porównywaniem obiektów. Dowiedziałeś się, że każdy obiekt w języku Ruby ma wartość boolowską i że występują w nim boolowskie obiekty specjalne (`true` i `false`), które reprezentują te wartości w ich najprostszej postaci. Trzeci obiekt specjalny `nil` reprezentuje stan niezdefiniowania lub braku. Omówiono także techniki porównywania obiektów za pomocą standardowego operatora porównań (`<=>`) oraz modułu `Comparable`.

Na końcu przyjrzeliśmy się sposobom zażądania od obiektów języka Ruby przekazania informacji o metodach, na jakie odpowiadają. Jest to odmiana techniki introspekcji, która może ułatwić zrozumienie tego, co ma miejsce w danym miejscu programu. Obszerniej introspekcją zajmiemy się w rozdziale 15.

Materiał zamieszczony w tym rozdziale dobrze przygotuje do tego, co znajdziesz w reszcie książki, i nie tylko tam. Czytając takie stwierdzenia jak: „Metoda ta ma alternatywę w postaci metody z wykrzyknikiem w nazwie", będziesz wiedzieć, co to oznacza. Gdy w dokumentacji znajdziesz informację o tym, że określony argument metody ma domyślną wartość `nil`, będzie to zrozumiałe. To, że zaznajomiłeś się z tymi powracającymi zagadnieniami, ułatwi szybsze powtórzenie związanych z nimi informacji w kolejnych rozdziałach poświęconych klasom i modułom wbudowanym języka Ruby oraz skoncentrowanie się na dalszym poznawaniu języka.

# 8 Łańcuchy, symbole i inne obiekty skalarne

**W tym rozdziale:**

- Tworzenie i modyfikowanie obiektów łańcuchów.
- Metody transformacji łańcuchów.
- Semantyka symboli.
- Porównywanie łańcuchów i symboli.
- Liczby całkowite i zmiennoprzecinkowe.
- Obiekty czasu i dat.

Termin *skalar* oznacza *jednowymiarowość*. W rozdziale odnosi się on do obiektów reprezentujących pojedyncze wartości, w przeciwieństwie do obiektów kolekcji lub kontenerów, które przechowują wiele wartości. Istnieje kilka „odcieni szarości": na przykład łańcuchy — oprócz tego, że stanowią pojedyncze jednostki tekstu, mogą być postrzegane jako kolekcje znaków. W pewnym stopniu skalar jest pojęciem względnym. Jednak w ramach pierwszego dobrego przybliżenia klasy omówione w rozdziale możesz postrzegać jako klasy jednowymiarowych obiektów o konkretnej wielkości, które kontrastują z obiektami kolekcji prezentowanymi w następnym rozdziale.

Obiekty wbudowane, którym przyjrzymy się w tym rozdziale, obejmują następujące:

- Łańcuchy, które w języku Ruby pełnią rolę standardowego sposobu obsługi danych tekstowych o dowolnej długości.
- Symbole, które (między innymi) są kolejnym sposobem reprezentowania tekstu w języku Ruby.

- Liczby całkowite.
- Liczby zmiennoprzecinkowe.
- Obiekty `Time`, `Date` i `DateTime`.

Wszystkie te obiekty, które pod innymi względami różnią się raczej zasadniczo, są skalarami. Są to jednowymiarowe obiekty niepełniące roli kontenerów, które nie zawierają w sobie żadnych dodatkowych obiektów, tak jak ma to miejsce w przypadku tablic. Nie oznacza to, że skalary nie są złożone i mają prostą semantykę. Jak się okaże, jest inaczej.

## 8.1. Zastosowanie łańcuchów

Język Ruby zapewnia dwie klasy wbudowane, które oferują wszystkie funkcje związane z reprezentacją i modyfikowaniem tekstu: `String` i `Symbol`. Choć łańcuchy i symbole znacznie się od siebie różnią, są na tyle podobne w zakresie wspólnych możliwości reprezentowania tekstu, że zasługują na omówienie w tym samym rozdziale. Zaczniemy od łańcuchów, które są standardowym sposobem reprezentowania treści tekstowej o dowolnej zawartości i długości. Łańcuchy zaprezentowano już w wielu kontekstach. W tym miejscu dokładniej przyjrzymy się ich semantyce i możliwościom. Najpierw dowiesz się, jak tworzyć łańcuchy, a następnie zostanie omówionych kilka sposobów modyfikowania łańcuchów, odpytywania ich (na przykład w celu uzyskania informacji o długości), porównywania ze sobą oraz transformowania (z małych liter do dużych itd.). Przeanalizujemy też dodatkowo szczegóły procesu przekształcania łańcuchów za pomocą metody `to_i` i powiązanych z nią metod.

### 8.1.1. Notacja łańcuchów

*Literał łańcuchowy* jest zwykle zawarty w znakach cudzysłowu:

```
"To jest łańcuch."
```

Mogą być również użyte apostrofy:

```
'To jest łańcuch.'
```

Zachowanie łańcucha ujętego w apostrofy jest jednak w niektórych sytuacjach inne niż łańcucha umieszczonego w cudzysłowie. Główną różnicą jest to, że interpolacja łańcuchów nie działa w przypadku łańcuchów w apostrofach. Aby zauważyć różnicę, sprawdź następujące dwa wiersze kodu:

```
puts "Dwa plus dwa wynosi #{2 + 2}."
puts 'Dwa plus dwa wynosi #{2 + 2}.'
```

Jak się okaże, po wklejeniu tych wierszy do sesji narzędzia irb uzyskasz następujące dwa bardzo różne wyniki:

```
Dwa plus dwa wynosi 4.
Dwa plus dwa wynosi #{2 + 2}.
```

Apostrofy wyłączają mechanizm interpolacji #{...}. Jeśli jest on wymagany, nie możesz użyć apostrofów. W odwrotnej sytuacji w razie potrzeby za pomocą *znaków zmiany znaczenia* \ możesz wyłączyć mechanizm interpolacji łańcuchów w łańcuchu ujętym w cudzysłów:

```
puts "Interpolacja z użyciem znaków zmiany znaczenia: \"\#{2 + 2}\"."
```

Łańcuchy ujęte w apostrofy i cudzysłów zachowują się odmiennie również w odniesieniu do konieczności zmiany znaczenia określonych znaków. Poniższe instrukcje dokumentują i demonstrują różnice. Przyjrzyj się dokładnie temu, jakie łańcuchy umieszczono w apostrofach, a jakie w cudzysłowie, a także sposobowi użycia znaku \:

```
puts "Znaki \\ wymagają zmiany znaczenia w cudzysłowie."
puts 'Nie możesz po prostu wpisać raz znaku \ w łańcuchu ujętym w apostrofy.'
puts "Niezależnie jednak od tego, czy zastosowano apostrofy, czy cudzysłów,..."
puts "...niezbędna jest zmiana znaczenia symbolu apostrofu lub cudzysłowu, np. \"."
puts 'Dotyczy to też znaku \' w łańcuchach ujętych w apostrofy.'
puts 'Znak \ razem z literą n wygląda jak ciąg \n wstawiony między apostrofy.'
puts "Oba znaki oznaczają jednak znak nowego wiersza\nw łańcuchu ujętym w cudzysłów."
puts 'To samo dotyczy ciągu \t, który okazuje się ciągiem \t z apostrofami,...'
puts "...ale wstawia znak tabulacji:\t wewnątrz cudzysłowu."
puts "Możliwa jest zmiana znaczenia znaku \ w celu uzyskania ciągów \\n i \\t z cudzysłowem."
```

Poniżej przedstawiono dane wyjściowe w postaci takiego zestawu cytatów. Choć wyniki nie są wiersz po wierszu identyczne z kodem, możesz stwierdzić, dlaczego tak się dzieje, jeśli przyjrzysz się instrukcji, która zwraca znak nowego wiersza:

```
Znaki \ wymagają zmiany znaczenia w cudzysłowie.
Nie możesz po prostu wpisać raz znaku \ w łańcuchu ujętym w apostrofy.
Niezależnie jednak od tego, czy zastosowano apostrofy, czy cudzysłów,...
...niezbędna jest zmiana znaczenia symbolu apostrofu lub cudzysłowu, np. ".
Dotyczy to też znaku ' w łańcuchach ujętych w apostrofy.
Znak \ razem z literą n wygląda jak ciąg \n wstawiony między apostrofy.
Oba znaki oznaczają jednak znak nowego wiersza
w łańcuchu ujętym w cudzysłów.
To samo dotyczy ciągu \t, który okazuje się ciągiem \t z apostrofami,...
...ale wstawia znak tabulacji:                                    wewnątrz cudzysłowu.
Możliwa jest zmiana znaczenia lewego ukośnika w celu uzyskania ciągów \n i \t z cudzysłowem.
```

W dalszej części rozdziału zostaną zaprezentowane inne przypadki interpolacji łańcuchów i zmiany znaczenia znaków. Tymczasem na razie najlepszym sposobem na zaznajomienie się z tymi operacjami jest poeksperymentowanie z łańcuchami w sesji narzędzia irb.

W języku Ruby oprócz apostrofów i cudzysłowów oferowanych jest kilka sposobów tworzenia łańcuchów.

### INNE MECHANIZMY STOSOWANIA CUDZYSŁOWÓW

Alternatywne mechanizmy stosowania cudzysłowów przyjmują postać `%znak{text}`, w przypadku której *znak* to jeden z kilku znaków specjalnych, natomiast nawiasy klamrowe zastępują wybrany ogranicznik. Przykładem jednego z takich mechanizmów jest ciąg `%q`, który generuje następujący łańcuch ujęty w apostrofy:

```
puts %q{W przypadku użycia ciągu %q nie jest wymagana zmiana znaczenia apostrofów.}
```

Jak pokazuje powyższe przykładowe zdanie, ponieważ znak cudzysłowu nie jest używany w roli znaku apostrofu, wewnątrz łańcucha nie jest wymagana zmiana znaczenia znaku cudzysłowu.

Dostępny jest również ciąg `%Q{}` generujący łańcuch ujęty w cudzysłów, a także zwykły ciąg `%{}` (znak procentu i ogranicznik), który także tworzy łańcuch umieszczony w cudzysłowie. Oczywiście nie jest konieczna zmiana znaczenia znaku cudzysłowu wewnątrz łańcuchów, które są reprezentowane z zastosowaniem jednej z tych notacji.

Ogranicznikiem dla notacji ze znakiem `%` może być właściwie cokolwiek, co chcesz, pod warunkiem że ogranicznik otwierający jest zgodny z domykającym. *Zgodność* oznacza w tym przypadku tworzenie pary nawiasów (klamrowe, okrągłe lub kwadratowe) złożonych z lewego i prawego lub pary w postaci dwóch takich samych znaków. A zatem wszystkie następujące ograniczniki są dozwolone:

```
%q-Łańcuch-
%Q/Kolejny łańcuch/
%[Jeszcze jeden łańcuch]
```

W roli ograniczników nie możesz użyć znaków alfanumerycznych, jeśli jednak zależy Ci na ukryciu czegoś, możesz zastosować spację. Trudno ją zobaczyć w przykładzie, dlatego poniższy kod w całości umieszczono w nawiasach kwadratowych, które nie powinny być wpisywane, jeśli zdecydujesz się na sprawdzenie przykładu w sesji narzędzia irb lub w pliku programu Ruby:

```
[%q Witaj! ]
```

Pomijając to, że przykład ze spacjami jako ogranicznikami jest zabawny (choć pouczający), przywodzi na myśl, co się stanie, gdy ogranicznik zostanie użyty wewnątrz łańcucha (ze względu na to, że w wielu łańcuchach znajdują się spacje). Jeśli ogranicznik jest pojedynczym znakiem, konieczna jest zmiana jego znaczenia:

```
[%q Witaj,\ kolego! ]
%q-Lepiej zmień znaczenie znaku \- wewnątrz tego łańcucha!-
```

W przypadku użycia dopasowanych nawiasów, gdy interpreter języka Ruby napotka lewy nawias wewnątrz łańcucha, przyjmuje, że nawias ten jest częścią łańcucha, po czym szuka pasującego do niego prawego nawiasu. Aby dołączyć niedopasowany nawias tego samego typu co nawiasy używane w roli ograniczników, niezbędna jest zmiana jego znaczenia:

```
%Q[Mogę tutaj umieścić nawiasy [] bez zmiany ich znaczenia.]
%q(Konieczna jest zmiana znaczenia znaku \(, jeśli zostanie tutaj sam użyty.)
%Q(To samo dotyczy znaku \).)
```

Każdy z mechanizmów stosowania cudzysłowów z ciągiem `%znak` generuje łańcuch ujęty w apostrofy lub cudzysłów. Takie rozróżnienie dotyczy wszystkich łańcuchów. Każdy łańcuch zalicza się do jednego z tych typów, niezależnie od tego, jakiej użyjesz notacji. Dotyczy to również notacji, której teraz się przyjrzymy. Mowa o składni dokumentów miejscowych.

**Narzędzie irb nie radzi sobie dobrze z częścią takiej składni**

Narzędzie irb dysponuje własnym analizatorem kodu Ruby, który musi borykać się z tym, że w trakcie analizowania jednego wiersza nie ma możliwości stwierdzenia, jaki będzie następny wiersz. W efekcie narzędzie irb realizuje swoje działania trochę inaczej niż interpreter języka Ruby. W przypadku mechanizmów cudzysłowów może się okazać, że w narzędziu irb zmiana znaczenia niedopasowanych nawiasów kwadratowych oraz innych powoduje dziwne wyniki. Ogólnie rzecz biorąc, lepszym rozwiązaniem będzie umieszczenie tych przykładów w poleceniu `ruby -e 'puts %q[ Przykład: \[ ]'` o formacie wiersza poleceń oraz podobnych do tego polecenia.

### DOKUMENTY MIEJSCOWE

*Dokument miejscowy* (ang. *here-doc*) to łańcuch, zazwyczaj złożony z wielu wierszy, który często przyjmuje postać szablonu lub zestawu wierszy danych. Termin „miejscowy" wynika stąd, że taki łańcuch jest fizycznie obecny w pliku programu, a nie jest wczytywany z osobnego pliku tekstowego.

Dokumenty miejscowe są tworzone za pomocą operatora <<, co zaprezentowano na przykładzie fragmentu następującej sesji narzędzia irb:

```
>> text = <<EOM
To jest pierwszy wiersz tekstu.
To jest drugi wiersz.
Gotowe.
EOM
=> "To jest pierwszy wiersz tekstu.\nTo jest drugi wiersz.\n Gotowe.\n"
```

Wyrażenie `<<EOM` oznacza *wystąpienie po nim tekstu, lecz bez uwzględniania następnego wystąpienia łańcucha* `EOM`. Ogranicznikiem może być dowolny łańcuch. `EOM` (*End of Message* — z ang. koniec komunikatu) to często wybierany wariant łańcucha. Łańcuch musi być wyrównany do lewej strony, a ponadto w wierszu, w którym występuje, nie może znajdować się nic innego. Wymóg wyrównania do lewej strony możesz wyeliminować przez umieszczenie łącznika przed operatorem <<:

```
>> text = <<-EOM
Łańcuch EOM nie wymaga wyrównania do lewej strony!
       EOM
=> "Łańcuch EOM nie wymaga wyrównania do lewej strony!\n"
```

Ogranicznik `EOM`, który powoduje zakończenie wczytywania tego przykładowego dokumentu miejscowego (tym razem jest on złożony z tylko jednego wiersza), znajduje się w środku wiersza.

Domyślnie dokumenty miejscowe są wczytywane jako łańcuchy ujęte w cudzysłów. Oznacza to, że mogą uwzględniać interpolację łańcuchów i użycie znaków zmiany znaczenia, takich jak `\n` i `\t`. Aby zastosować dokument miejscowy ujęty w apostrofy, ogranicznik domykający umieść w apostrofach w momencie rozpoczynania tworzenia dokumentu. Aby różnica ta była bardziej widoczna, w następującym przykładzie uwzględniono instrukcję `puts` dokumentu miejscowego:

```
>> text = <<-'EOM'
Ujęto w apostrofy!
Zwróć uwagę na literał \n.
```

```
A także na literał #{2+2}.
EOM
=> "Ujęto w apostrofy!\nZwróć uwagę na literał \\n.\nA także na literał \#{2+2}.\n"
>> puts text
Ujęto w apostrofy!
Zwróć uwagę na literał \n.
A także na literał #{2+2}.
```

Wyrażenie <<EOM (lub jego odpowiednik) nie musi być ostatnim elementem w swoim wierszu. Gdziekolwiek się ono znajdzie, pełni rolę elementu zastępczego dokumentu miejscowego, który ma się pojawić. Oto wyrażenie <<EOM przekształcane w liczbę całkowitą i mnożone przez 10:

```
a = <<EOM.to_i * 10
5
EOM
puts a
```

**Dane wyjściowe: 50**

Dokumentu miejscowego możesz nawet użyć w konstruktorze obiektu literału. Oto przykład, w którym łańcuch umieszczany jest w tablicy, co powoduje utworzenie łańcucha w postaci dokumentu miejscowego:

```
array = [1,2,3, <<EOM, 4]
To jest dokument miejscowy!
Staje się on tablicą array[3].
EOM
p array
```

Dane wyjściowe są następujące:

```
[1, 2, 3, "To jest dokument miejscowy!\nStaje się on tablicą array[3].\n", 4]
```

Notacji <<EOM możesz użyć jako argumentu metody. Argument staje się dokumentem miejscowym następującym po wierszu, w którym występuje wywołanie metody. Może to być przydatne, gdy chcesz uniknąć umieszczania zbyt dużej ilości tekstu na liście argumentów:

```
do_something_with_args(a, b, <<EOM)
http://bardzo_dlugi_adres_url_lub_inny_tekst_najlepiej_umiescic_we_wlasnym_wierszu
EOM
```

Gdy już umiesz tworzyć łańcuchy, musisz wiedzieć, jakie są ich możliwości. Są one spore. Przyjrzymy się im szczegółowo, zaczynając od podstaw dotyczących modyfikowania łańcuchów.

### *8.1.2. Podstawowe modyfikacje łańcuchów*

W tym kontekście słowo *podstawowe* oznacza modyfikowanie obiektu na najniższych poziomach przez wykonywanie następujących operacji: pobieranie i ustawianie podłańcuchów oraz łączenie łańcuchów ze sobą. Z perspektywy języka Ruby techniki te nie są w żadnym stopniu bardziej podstawowe niż te, które pojawią się w dalszej części omówienia łańcuchów. Jednak pod względem znaczeniowym techniki te są bliższe „materii” łańcuchów, jak to można określić.

### POBIERANIE I USTAWIANIE PODŁAŃCUCHÓW

Aby pobrać *n*-ty znak łańcucha, użyj operatora/metody `[]`, zapewniając mu indeks (bazujący na zerze jako pozycji początkowej) żądanego znaku. Liczby ujemne są indeksowane od końca łańcucha:

```
>> string = "Ruby to fajny język."
=> "Ruby to fajny język."
>> string[5]
=> "t"
>> string[-12]
=> "f"
```

Jeśli podasz drugi argument całkowitoliczbowy *m*, uzyskasz podłańcuch *m* znaków, począwszy od określonego indeksu:

```
>> string[5,8]
=> "to fajny"
```

Możliwe jest też podanie jako argumentu pojedynczego obiektu *zakresu*. Obszerniej zakresom przyjrzymy się w dalszej części rozdziału. Na razie zakres *n..m* można traktować jako wszystkie wartości między wartościami *n* i *m* z nimi włącznie (lub z wyłączeniem wartości *m*, jeśli zamiast dwóch kropek użyjesz trzech). W zakresie mogą się znajdować liczby ujemne, które są liczone w kierunku od końca do początku łańcucha. Jednak drugi indeks zawsze musi być bliżej końca łańcucha niż pierwszy indeks. Logika indeksu obejmuje wyłącznie przetwarzanie od lewej do prawej strony:

```
>> string[7..14]
=> " fajny j"
>> string[-12..-3]
=> "fajny języ"
>> string[-12..20]
=> "fajny język."
>> string[15..-1]
=> "ęzyk."
```

Możesz też uzyskiwać podłańcuchy za pomocą jawnego ich wyszukiwania. Jeśli podłańcuch zostanie znaleziony, jest zwracany. W przeciwnym razie wartością zwracaną jest `nil`.

```
>> string["fajny język"]
=> "fajny język"
>> string["bardzo fajny język"]
=> nil
```

Istnieje również możliwość wyszukiwania dopasowania dla wzorca przy użyciu metody `[]` z wyrażeniem regularnym. Wewnątrz tej metody znajdują się argumenty, dlatego realizuje ona dowolne zaprogramowane działania:

```
>> string[/f[aj ]+/]
=> "faj"
```

Wyrażeniami regularnymi zajmiemy się osobno w rozdziale 11. W tym miejscu zaprezentowano je w celu ogólnego pokazania możliwości związanych z takim sposobem szukania podłańcuchów.

Metoda `[]` jest też dostępna pod nazwą `slice`. Co więcej, wersja metody `slice` o nazwie `slice!`, która zmienia odbiorcę, trwale usuwa znaki z łańcucha:

```
>> string.slice!("fajny ")
=> "fajny "
>> string
=> "Ruby to język."
```

W celu ustawienia nowej wartości dla części łańcucha użyj metody `[]=`. Pobiera ona tego samego rodzaju argumenty indeksowania co metoda `[]`, ale zmienia wartości na podane. W ramach testowania powyższego krótkiego łańcucha poniżej zamieszczono kilka przykładów ustawiania podłańcuchów wraz ze sprawdzeniem każdorazowo zmodyfikowanego łańcucha:

```
>> string = "Ruby to fajny język."
=> "Ruby to fajny język."
>> string["fajny"] = "znakomity"
=> "znakomity"
>> string
=> "Ruby to znakomity język."
>> string[-1] = "!"
=> "!"
>> string
=> "Ruby to znakomity język!"
>> string[-9..-1] = "coś wartego opanowania!"
=> "coś wartego opanowania!"
>> string
=> "Ruby to coś wartego opanowania!"
```

A zatem liczby całkowite, zakresy, łańcuchy i wyrażenia regularne mogą pełnić rolę specyfikatorów indeksu lub podłańcuchów. Jeśli spróbujesz ustawić część łańcucha, która nie istnieje, podając zbyt dużą lub zbyt niską wartość indeksu liczbowego albo łańcuch lub wyrażenie regularne niezgodne z łańcuchem, zostanie wygenerowany błąd krytyczny.

Oprócz modyfikowania poszczególnych łańcuchów możesz też łączyć je ze sobą.

### ŁĄCZENIE ŁAŃCUCHÓW

Istnieje kilka technik łączenia łańcuchów. Różnią się one tym, czy drugi łańcuch jest trwale dodawany do pierwszego, albo tym, czy nowy, trzeci łańcuch jest tworzony z dwóch pierwszych, czyli inaczej mówiąc, czy operacja powoduje zmianę odbiorcy.

Aby utworzyć nowy łańcuch złożony z co najmniej dwóch łańcuchów, możesz użyć metody/operatora `+` w celu jednoczesnego zastosowania oryginalnych łańcuchów. Oto zapis sesji narzędzia irb w przypadku dodawania do siebie łańcuchów:

```
>> "a" + "b"
=> "ab"
>> "a" + "b" + "c"
=> "abc"
```

Łańcuch uzyskiwany z metody `+` zawsze jest nowym łańcuchem. Możesz to sprawdzić, przypisując łańcuch do zmiennej, używając jej w operacji z metodą `+` oraz określając wartość zmiennej po wykonaniu tej operacji:

```
>> str = "Witaj, "
=> "Witaj, "
>> str + "kolego."
=> "Witaj, kolego."  ←❶
>> str
=> "Witaj, "         ←❷
```

Dla wyrażenia `str + "kolego."` (jest to „lukier" składniowy dla wywołania metody `str.+("kolego")`) określany jest nowy łańcuch `"Witaj, kolego."` ❶, ale łańcuch `str` pozostaje bez zmian ❷.

Aby trwale dodać (dołączyć) drugi łańcuch do istniejącego łańcucha, użyj metody <<, która również zawiera formę pseudooperatora „lukru" składniowego:

```
>> str = "Witaj, "
=> "Witaj, "
>> str << "kolego."
=> "Witaj, kolego."
>> str
=> "Witaj, kolego."  ←❶
```

W tym przykładzie do oryginalnego łańcucha `str` dołączono nowy łańcuch, co możesz stwierdzić przez określenie wartości zmiennej `str` znajdującej się na końcu powyższego kodu ❶.

Interpolacja łańcuchów to między innymi kolejny sposób łączenia łańcuchów. Choć możliwości tej operacji zostały już wcześniej zaprezentowane, przyjrzyjmy się kilku szczegółom związanym ze sposobem jej przebiegania.

### ŁĄCZENIE ŁAŃCUCHÓW ZA POMOCĄ INTERPOLACJI

W swoim najprostszym wariancie interpolacja łańcuchów powoduje umieszczenie jednego istniejącego łańcucha w innym:

```
>> str = "Witaj, "
=> "Witaj, "
>> "#{str} kolego."
=> "Witaj, kolego."
```

Wynikiem jest nowy łańcuch: `"Witaj, kolego."`. Warto jednak mieć świadomość tego, że interpolacja może uwzględniać dowolne wyrażenie języka Ruby:

```
>> "Suma wynosi #{2 + 2}."
=> "Suma wynosi 4."
```

Kod w nawiasach klamrowych może być dowolny (tutaj nawiasy klamrowe są niezbędne, nie tak jak w przypadku ciągu `%q{}`, dla którego możesz wybrać własny ogranicznik). Nietypowym podejściem będzie znaczne komplikowanie kodu, gdyż spowoduje to zmniejszenie przejrzystości struktury i czytelności programu. Jednak w języku Ruby możesz zastosować dowolny interpolowany kod, dla którego interpreter uprzejmie umieści w łańcuchu reprezentację łańcuchową wartości podanej w kodzie:

```
>> "Mam na imię #{class Person
                  attr_accessor :name
                end
                d = Person.new
                d.name = "Dawid"
```

```
                d.name
                }."
=> " Mam na imię Dawid."
```

Naprawdę, ale to *naprawdę* nie chcesz tego robić, ale ważne jest uświadomienie sobie, że możesz dokonać interpolacji dowolnego żądanego kodu. Interpreter języka Ruby cierpliwie oczekuje na uruchomienie całego kodu, a następnie pobiera jego końcową wartość (w tym przypadku `d.name`, ponieważ jest to ostatnie wyrażenie wewnątrz bloku interpolacji) i dokonuje jej interpolacji.

Istnieje znacznie przyjemniejszy sposób osiągnięcia czegoś podobnego. Interpreter języka Ruby przeprowadza interpolację przez wywołanie metody `to_s` w obiekcie, dla którego określana jest wartość kodu interpolacji. Możesz to wykorzystać do usprawnienia procesu tworzenia łańcuchów, definiując odpowiednio własne metody `to_s`:

```
>> class Person
>>   attr_accessor :name
>>   def to_s
>>     name
>>   end
>> end
=> :to_s
>> david = Person.new
=> #<Person:0x00000101a73cb0>
>> david.name = "Dawid"
=> "Dawid"
>> "Witaj, #{david}!"
=> "Witaj, Dawid!"
```

Obiekt `david` pełni rolę interpolowanego kodu i generuje wynik swojej operacji z wykorzystaniem metody `to_s`, którą zdefiniowano jako metodę opakowującą metodę pobierającą `name`. Użycie metody `to_s` zapewnia wygodny sposób sterowania zachowaniem obiektów w interpolowanych łańcuchach. Pamiętaj jednak, że w powyższym przykładzie możesz też użyć konstrukcji `david.name`. A zatem jeśli metoda `to_s` klasy może zostać zastosowana w szerszym zakresie niż w przypadku bardzo specyficznego wariantu interpolacji, zwykle możesz ją dostosować.

Po utworzeniu i ewentualnie zmodyfikowaniu łańcucha możesz zażądać od niego przekazania dość znacznej ilości informacji o nim. Przyjrzymy się teraz sposobom odpytywania łańcuchów.

### *8.1.3. Odpytywanie łańcuchów*

Istnieje kilka wariantów zapytań dotyczących łańcuchów. Niektóre z nich zapewniają odpowiedź boolowską (prawda lub fałsz), a niektóre udostępniają coś w rodzaju raportu dotyczącego bieżącego statusu łańcucha. Bazując na tym, zorganizujemy nasze omówienie metod odpytywania łańcuchów.

#### ZAPYTANIA BOOLOWSKIE DOTYCZĄCE ŁAŃCUCHÓW

Za pomocą metody `include?` od łańcucha możesz zażądać informacji o tym, czy zawiera dany podłańcuch. Dla użytego wcześniej łańcucha (`"Ruby to fajny język."`) zapytania włączenia mogą mieć następującą postać:

```
>> string.include?("Ruby")
=> true
>> string.include?("Angielski")
=> false
```

Metody `start_with?` i `end_with?` umożliwiają sprawdzenie, czy odpowiednio łańcuch zaczyna się lub kończy danym łańcuchem:

```
>> string.start_with?("Ruby")
=> true
>> string.end_with?("!!!")
=> false
```

Za pomocą metody `empty?` możliwe jest też sprawdzenie braku danej treści, czyli tego, czy w ogóle istnieją dowolne znaki:

```
>> string.empty?
=> false
>> "".empty?
=> true
```

Pamiętaj o tym, że białe znaki są liczone jako znaki. Łańcuch `" "` nie jest pusty.

### ZAPYTANIA DOTYCZĄCE TREŚCI

Działanie metod `size` i `length` (są to synonimy tej samej metody) jest zgodne z tym, co sugerują ich nazwy:

```
>> string.size
=> 24
```

Aby dowiedzieć się, ile razy w łańcuchu występuje dana litera lub zestaw liter, użyj metody `count`. W celu określenia liczby wystąpień jednej litery podaj ją jako argument. W przypadku łańcucha `"Ruby to fajny język."` litera `y` występuje trzykrotnie:

```
>> string.count("y")
=> 3
```

W celu ustalenia liczby liter zakresu istniejących w łańcuchu możesz użyć zakresu z łącznikiem w środku:

```
>> string.count("g-m")
=> 3
```

W specyfikacjach znaków rozróżniana jest ich wielkość:

```
>> string.count("A-Z")
=> 1
```

Możliwe jest też podanie ręcznie wpisanego zestawu znaków, których liczba ma zostać ustalona:

```
>> string.count("aey. ")  <------ Trzy litery oraz kropka i spacja
=> 10
```

Aby zanegować wyszukiwanie, czyli określić liczbę znaków, które są niezgodne z podanymi znakami, na początku specyfikacji umieść znak wstawienia ^:

```
>> string.count("^aey. ")
=> 14
>> string.count("^g-m")
=> 19
```

Jeśli jesteś zaznajomiony z wyrażeniami regularnymi, rozpoznasz technikę wykorzystującą znak ^ jako bliskiego „kuzyna" negacji klasy znaków wyrażeń regularnych. Składnie specyfikacji mogą być łączone. Możliwe jest nawet podanie więcej niż jednego argumentu:

```
>> string.count("ag-m")
=> 8
>> string.count("ag-m", "^l")
=> 6
```

**Określanie liczby wystąpień litery a oraz liter z zakresu g – m z wyłączeniem litery l**

Innym sposobem odpytywania łańcuchów odnośnie do ich treści jest zastosowanie metody `index`. Stanowi ona wariant odwrotności operacji polegającej na użyciu metody `[]` z indeksem liczbowym. Zamiast wyszukiwać podłańcuch w konkretnej pozycji indeksu, metoda `index` zwraca pozycję indeksu, na której występuje dany podłańcuch. Zwracane jest pierwsze wystąpienie od lewej strony. Aby uzyskać pierwsze wystąpienie od prawej strony, użyj metody `rindex`:

```
>> string.index("fajny")
=> 8
>> string.index("y")
=> 3
>> string.rindex("y")
=> 17
```

Choć łańcuchy składają się ze znaków, w języku Ruby nie ma osobnej klasy znaków. Dzięki metodzie `ord` w przypadku łańcuchów jednoznakowych możliwe jest określenie ich kodu porządkowego:

```
>> "a".ord
=> 97
```

W przypadku użycia metody `ord` dla dłuższego łańcucha uzyskasz kod pierwszego znaku:

```
>> "abc".ord
=> 97
```

Odwrotna operacja jest dostępna dla liczb całkowitych w postaci metody `chr`:

```
>> 97.chr
=> "a"
```

Zażądanie liczby, która nie odpowiada żadnemu znakowi, dla jego odpowiednika zwracanego przez metodę `chr`, spowoduje wygenerowanie błędu krytycznego.

Oprócz udostępniania informacji o sobie łańcuchy mogą same porównywać się z innymi łańcuchami w celu przeprowadzenia sprawdzenia pod kątem ich równości lub uporządkowania.

### 8.1.4. Porównywanie łańcuchów i określanie ich kolejności

Klasa `String` dołącza moduł `Comparable` i definiuje metodę `<=>`. A zatem łańcuchy są odpowiednią opcją w przypadku porównań opartych na kolejności kodów znaków (w formacie ASCII lub innym):

```
>> "a" <=> "b"
=> -1
>> "b" > "a"
=> true
>> "a" > "A"
=> true
>> "." > ","
=> true
```

Pamiętaj, że metoda/operator „statku kosmicznego" zwraca wartość `-1`, jeśli większy jest obiekt po prawej stronie, wartość `1`, gdy większy jest obiekt po lewej stronie, oraz wartość `0` w przypadku równości dwóch obiektów. W pierwszym przypadku w poprzedniej sekwencji metoda zwraca wartość `-1`, ponieważ łańcuch `"b"` jest większy od łańcucha `"a"`. Jednak łańcuch `"a"` jest większy od łańcucha `"A"`, ponieważ kolejność jest ustalana za pomocą wartości znaku, a wartości znaków dla łańcuchów `"a"` i `"A"` wynoszą odpowiednio 97 i 65 (dotyczy to domyślnego kodowania w języku Ruby w formacie UTF-8). Podobnie łańcuch `"."` jest większy od łańcucha `","`, gdyż wartość dla znaku kropki to 46, a dla przecinka to 44 (w punkcie 8.1.7 zamieszczono więcej informacji na temat kodowania).

Podobnie do obiektów języka Ruby w ogólności łańcuchy oferują kilka metod służących do sprawdzania równości.

#### PORÓWNYWANIE DWÓCH ŁAŃCUCHÓW POD KĄTEM RÓWNOŚCI

Metoda == to najczęściej używana metoda porównująca łańcuchy, która sprawdza zawartość łańcuchów pod kątem równości:

```
>> "string" == "string"
=> true
>> "string" == "dom"
=> false
```

Dwa obiekty literału `"string"` to różne obiekty, które jednak mają taką samą zawartość. Oznacza to, że pomyślnie przechodzą test metody ==. Łańcuch `"dom"` ma inną zawartość, dlatego na podstawie testu metody == nie jest uważany za równy łańcuchowi `"string"`.

Inna metoda testująca równość `String#eql?` sprawdza dwa łańcuchy pod kątem identycznej zawartości. W praktyce metoda ta zwraca zwykle ten sam wynik co metoda == (choć istnieją subtelne różnice w implementacjach tych dwóch metod, możesz korzystać z dowolnej z nich; stwierdzisz, że metoda == jest powszechniejsza). Trzecia metoda `String#equal?` zachowuje się przeważnie podobnie do metody `equal?`, czyli sprawdza, czy dwa łańcuchy są tym samym obiektem, albo jeśli już o tym mowa, czy łańcuch i dowolny inny obiekt są identycznym obiektem:

```
>> "a" == "a"
=> true
>> "a".equal?("a")
=> false
>> "a".equal?(100)
=> false
```

Pierwszy test kończy się pomyślnie, ponieważ dwa łańcuchy mają taką samą zawartość. Drugi test kończy się niepowodzeniem, gdyż pierwszy łańcuch nie jest tym samym obiektem co drugi łańcuch (oczywiście żaden łańcuch nie jest takim samym obiektem jak liczba całkowita `100`, dlatego ten test również nie kończy się pomyślnie). Jest to odpowiednie miejsce na przypomnienie tego, że łańcuchy, które na pierwszy rzut oka wydają się identyczne, dla interpretera języka Ruby są obiektami z różnymi tożsamościami.

W następnych dwóch punktach zostaną kolejno zaprezentowane operacje *konwersji* i *transformacji* łańcuchów. Różnica między nimi polega na tym, że transformacja wiąże się z zastosowaniem dla zawartości łańcucha określonego rodzaju algorytmu lub procedury, natomiast konwersja oznacza uzyskiwanie na podstawie łańcucha drugiego, niepowiązanego obiektu, który zwykle nie jest nawet łańcuchem.

### 8.1.5. Transformacja łańcuchów

Nieoficjalnie transformacje łańcuchów w języku Ruby należą do następujących trzech kategorii: transformacje wielkości znaków, transformacje formatowania i transformacje zawartości. Kolejno przyjrzymy się każdej z nich.

#### TRANSFORMACJE WIELKOŚCI ZNAKÓW

Łańcuchy umożliwiają zwiększanie, zmniejszanie i zastępowanie wielkości ich znaków. Wszystkie metody zmieniające wielkość znaków mają odpowiedniki z wykrzyknikiem w nazwie, które modyfikują odbiorcę:

```
>> string = "Dawid A. Nowak"
=> "Dawid A. Nowak"
>> string.upcase
=> "DAWID A. NOWAK"
>> string.downcase
=> "dawid a. nowak"
>> string.swapcase
=> "dAWID a. nOWAK"
```

Dostępne jest też przydatne ulepszenie, które pozwala rozpocząć łańcuch od dużej litery:

```
>> string = "dawid"
=> "dawid"
>> string.capitalize
=> "Dawid"
```

Podobnie jak inne metody dokonujące transformacji wielkości znaków metoda `capitalize` dysponuje swoim odpowiednikiem z wykrzyknikiem w nazwie, czyli metodą `capitalize!`.

W przypadku formatu łańcucha możliwych jest kilka transformacji, z których większość ma na celu ułatwienie zapewnienia lepszego wyglądu łańcuchów.

### TRANSFORMACJE FORMATOWANIA

Mówiąc wprost, transformacje formatowania stanowią podzbiór transformacji zawartości. Jeśli nie zmieniła się sekwencja znaków reprezentowanych przez łańcuch, transformacja nie będzie znacząca. W ramach podpunktu poświęconego formatowaniu zostaną pogrupowane transformacje, których głównym zadaniem jest poprawienie jakości prezentowania łańcuchów.

Metody `rjust` i `ljust` zwiększają wielkość łańcucha do wartości podanej w pierwszym argumencie, uzupełniając łańcuch w razie potrzeby znakami spacji:

```
>> string = "Dawid A. Nowak"
=> "Dawid A. Nowak"
>> string.rjust(25)
=> "           Dawid A. Nowak"
>> string.ljust(25)
=> "Dawid A. Nowak           "
```

Jeśli podasz drugi argument, zostanie użyty jako uzupełnienie. Argument ten może zawierać więcej niż jeden znak:

```
>> string.rjust(25,'.')
=> "..........Dawid A. Nowak"
>> string.rjust(25,'><')
=> "><><><><><Dawid A. Nowak"
```

Wzorzec uzupełniania jest powtarzany tyle razy, ile razy się zmieści, z obcięciem w razie potrzeby ostatniego wstawionego znaku.

Aby dopełnić sferę wyrównywania, udostępniono metodę `center`, która zachowuje się podobnie do metod `rjust` i `ljust`, z tym że znaki łańcucha umieszcza w środku:

```
>> "Środek".center(20, "*")
=> "*****Środek*****"
```

Miejsca uzupełniania o nieparzystej liczbie są wyświetlane z przewagą po prawej stronie:

```
>> "Środek".center(21, "*")
=> "*****Środek******"
```

Możliwe jest też upiększenie łańcuchów przez usunięcie z ich jednej lub obu stron białych znaków za pomocą metod `strip`, `lstrip` i `rstrip`:

```
>> string = "   Dawid A. Nowak   "
=> "   Dawid A. Nowak   "
>> string.strip
=> "Dawid A. Nowak"
>> string.lstrip
=> "Dawid A. Nowak   "
>> string.rstrip
=> "   Dawid A. Nowak"
```

Wszystkie trzy metody usuwające znaki łańcucha mają swoje wersje metod ze znakiem `!` w nazwie, które trwale modyfikują łańcuch.

### TRANSFORMACJE ZAWARTOŚCI

Przyjrzymy się kilku, choć nie wszystkim sposobom transformacji łańcucha przez zmianę jego zawartości.

Obie metody `chop` i `chomp` służą do usuwania znaków z końców łańcuchów, ale realizują to w odmienny sposób. Podstawowa różnica polega na tym, że metoda `chop` usuwa znak bezwarunkowo, natomiast metoda `chomp` usuwa docelowy podłańcuch, jeśli znajdzie go na końcu łańcucha. Domyślnie docelowym podłańcuchem tej metody jest znak nowego wiersza `\n`. Podłańcuch ten możesz nadpisać, zapewniając metodzie `chomp` argument:

```
>> "Dawid A. Nowak".chop
=> "Dawid A. Nowa"
>> "Dawid A. Nowak\n".chomp
=> "Dawid A. Nowak"
>> "Dawid A. Nowak".chomp('ak')
=> "Dawid A. Nowa"
```

Jak dotąd spośród metod `chop` i `chomp` druga z nich najczęściej była używana do usuwania znaków nowego wiersza z końców łańcuchów, które zwykle trafiają do programu w postaci wierszy pliku lub danych wprowadzonych za pomocą klawiatury.

Obie metody mają odpowiedniki metod z wykrzyknikiem w nazwie, które zmieniają łańcuch.

Reprezentantem bardziej radykalnego wariantu usuwania znaków jest metoda `clear`, która usuwa wszystkie znaki łańcucha, powodując, że jest on pusty:

```
>> string = "Dawid A. Nowak"
=> "Dawid A. Nowak"
>> string.clear
=> ""
>> string
=> ""
```

`String#clear` to znakomity przykład metody, która zmienia swojego odbiorcę, choć jej nazwa nie jest zakończona znakiem `!`. Nazwa metody `clear` jasno wskazuje na to, że coś się dzieje z łańcuchem. Nie miałoby sensu użycie metody `clear`, która nie wprowadziłaby w łańcuchu żadnych zmian. Byłby to jedynie rozwlekły sposób wyrażenia łańcucha pustego (`""`).

Aby zastąpić wszystkie znaki bez konieczności pozbawiania łańcucha zawartości, możesz użyć metody `replace`, która pobiera argument w postaci łańcucha i jego zawartością zastępuje bieżącą zawartość łańcucha:

```
>> string = "(nazwa zostanie podana później)"
=> "(nazwa zostanie podana później)"
>> string.replace("Dawid A. Nowak")
=> "Dawid A. Nowak"
```

Jak w przypadku metody `clear` metoda `replace` trwale zmienia łańcuch, co i tym razem jest sugerowane przez jej nazwę.

Za pomocą metody `delete` możesz określić konkretne znaki do usunięcia z łańcucha. Argumenty tej metody podlegają tym samym regułom co argumenty metody `count` (zajrzyj do punktu 8.1.3.).

```
>> "Dawid A. Nowak".delete("abo")
=> "Dwid A. Nwk"
>> "Dawid A. Nowak".delete("^abo")
=> "aoa"
>> "Dawid A. Nowak".delete("a-e","^c")
=> "Dwi A. Nowk"
```

Inną specjalizowaną transformację łańcuchów zapewnia metoda `crypt`, która przeprowadza szyfrowanie DES (*Data Encryption Standard*) łańcucha podobnie do funkcji biblioteki `crypt(3)` systemu Unix. Pojedynczy argument metody `crypt` ma postać dwuznakowego łańcucha klucza dodatkowego:

```
>> "Dawid A. Nowak".crypt("34")
=> "34RFy4B2u5sJ6"
```

Pamiętaj o przeczytaniu informacji dotyczących solidności dowolnej używanej techniki szyfrowania, w tym metody `crypt`.

Inkrementacja łańcuchów to ostatnia technika transformacji, jakiej się przyjrzymy. Za pomocą metody `succ` (dostępna jest też pod nazwą `next`) możesz uzyskać następny łańcuch o najwyższej liczbie porządkowej. Uporządkowanie łańcuchów jest tak określane, aby miało sens, nawet kosztem ścisłej kolejności kodów znaków: `"a"` następuje po `"`"` (znak odwrotnego apostrofu) tak jak w formacie ASCII, ale po `"z"` występuje `"aa"`, a nie `"{"`. Inkrementacja na wzór prędkościomierza jest kontynuowana dla całego alfabetu:

```
>> "a".succ
=> "b"
>> "abc".succ
=> "abd"
>> "azz".succ
=> "baa"
```

Możliwość inkrementowania łańcuchów okazuje się przydatna w sytuacjach, w których wymagane są unikatowe łańcuchy generowane wsadowo (być może do zastosowania jako nazwy plików).

Jak już pokazano, łańcuchy (podobnie jak inne obiekty) mogą się same konwertować za pomocą metod z rodziny `to_*`. W dalszej kolejności przyjrzymy się jeszcze dokładniej konwersji łańcuchów.

### 8.1.6. Konwersje łańcuchów

Metoda `to_i` zaprezentowana w poprzednim rozdziale jest jedną z metod konwersji dostępnych w przypadku łańcuchów. Metoda ta oferuje dodatkową funkcję: jeśli zapewnisz metodzie argument w postaci liczby dodatniej z zakresu 2 – 36, konwertowany łańcuch jest interpretowany jako reprezentujący liczbę w bazie, która odpowiada argumentowi.

Aby na przykład zinterpretować liczbę 100 jako liczbę o bazie 17, możesz użyć następującego kodu:

```
>> "100".to_i(17)
=> 289
```

Dane wyjściowe to odpowiednik dziesiętny liczby 100 dla bazy 17.

Bazy 8 i 16 są uważane za szczególne przypadki z dedykowanymi metodami, dlatego nie ma potrzeby korzystania z metody `to_i`. Są to odpowiednio metody `oct` i `hex`:

```
>> "100".oct
=> 64
>> "100".hex
=> 256
```

Inne metody konwersji dostępne w przypadku łańcuchów obejmują metodę `to_f` (konwersja do liczby zmiennoprzecinkowej), metodę `to_s` (konwersja do łańcucha; metoda zwraca swojego odbiorcę) oraz metodę `to_sym` lub `intern` (przekształca łańcuch w obiekt `Symbol`). Żadna z tych metod nie ma w sobie nic szczególnie zaskakującego:

```
>> "1,2345".to_f
=> 1,2345
>> "Witaj".to_s
=> "Witaj"
>> "abcde".to_sym
=> :abcde
>> "1,2345i jakieś słowa".to_f
=> 1,2345
>> "tylko kilka słów".to_i
=> 0
```

Każdy łańcuch składa się z sekwencji bajtów odwzorowywanych na znaki. To, *jak* dokładnie są one odwzorowywane na znaki (ile bajtów tworzy znak, a także jakie są to znaki), jest kwestią *kodowania*, któremu przyjrzymy się teraz pobieżnie.

### 8.1.7. Kodowanie łańcuchów — krótkie wprowadzenie

Zagadnienie kodowania znaków jest interesujące, ale obszerne. Dostępnych jest wiele typów kodowania. Ponadto daleko jest od osiągnięcia ogólnoświatowego konsensusu co do najlepszego z nich. W wersji 1.9 języka Ruby dodano do łańcuchów sporą liczbę funkcji i elementów inteligencji związanych z kodowaniem. W języku Ruby 2 wielką zmianą było zastosowanie formatu UTF-8 zamiast formatu US-ASCII w roli domyślnego kodowania skryptów Ruby. Kodowanie w języku Ruby w dalszym ciągu stanowi temat dyskusji i prac projektowych. Nie mamy zamiaru rozwijać obszerniej tego zagadnienia w tym miejscu, ale będziemy mieć je na uwadze, a ponadto przyjrzymy się niektórym ważnym technikom powiązanym z kodowaniem.

#### USTAWIANIE KODOWANIA PLIKU ŹRÓDŁOWEGO

Najpierw należy wspomnieć, że kod źródłowy używa określonego kodowania. Domyślnie pliki źródłowe Ruby korzystają z kodowania UTF-8. Możesz to stwierdzić przez zażądanie od interpretera języka Ruby wyświetlenia wartości `__ENCODING__`. Następujący wiersz umieść w pliku i uruchom go:

```
puts __ENCODING__   <------ Dane wyjściowe: UTF-8
```

Wiersz ten musi zostać wstawiony do pliku, ponieważ możesz uzyskać różne wyniki w przypadku uruchomienia polecenia bezpośrednio z poziomu wiersza poleceń. Powodem różnic jest to, że uruchomienie kodu Ruby bez podanej nazwy pliku powoduje

pobranie kodowania z bieżącego ustawienia regionalnego. Możesz to sprawdzić, obserwując efekt wykonania tego samego polecenia ze zmienną środowiskową `LANG` ustawioną na inną wartość:

```
LANG=en_US.iso885915 ruby -e 'puts __ENCODING__'
```
← **Dane wyjściowe: US-ASCII**

W celu zmiany kodowania pliku źródłowego musisz użyć *„magicznego" komentarza* na początku pliku. Komentarz przyjmuje następującą postać:

```
# encoding: kodowanie
```

Wartość *`kodowanie`* to identyfikator kodowania. Aby na przykład zakodować plik źródłowy w formacie US-ASCII, następujący wiersz umieść na początku pliku:

```
# encoding: ASCII
```

Czasami wiersz ten określany jest mianem „magicznego" komentarza.

Oprócz pliku źródłowego możesz też zastosować zapytanie i ustawić kodowania poszczególnych łańcuchów.

### KODOWANIE POSZCZEGÓLNYCH ŁAŃCUCHÓW

Łańcuchy poinformują o swoim kodowaniu:

```
>> str = "Łańcuch testowy"
=> "Łańcuch testowy"
>> str.encoding
=> #<Encoding:UTF-8>
```

Łańcuch możesz zakodować przy użyciu innego kodowania, pod warunkiem że dozwolona (zależy to od możliwości obsługi nowego kodowania przez łańcuch) jest konwersja z oryginalnego kodowania do nowego (*transkodowanie*):

```
>> str.encode("US-ASCII")
=> "Łańcuch testowy"
```

W razie potrzeby możesz wymusić kodowanie za pomocą metody `force_encoding`, która pomija tabelę „dozwolonych" kodowań i koduje bezwarunkowo bajty łańcucha przy użyciu określonego kodowania.

Wersja metody `encode` z wykrzyknikiem w nazwie trwale zmienia kodowanie łańcucha:

```
>> str.encode!("US-ASCII")
=> "Łańcuch testowy"
>> str.encoding
=> #<Encoding:US-ASCII>
```

Na kodowanie łańcucha ma też wpływ obecność określonych znaków w łańcuchu i/lub poprawianie go przy użyciu konkretnych znaków. Dowolne znaki łańcucha mogą być reprezentowane przez sekwencję zmiany znaczenia `\x` (w przypadku dwucyfrowej liczby szesnastkowej reprezentującej bajt) lub `\u` (występuje po kodzie UTF-8 i wstawia odpowiedni znak).

Efekt kodowania łańcucha zależy od znaku. W przypadku kodowania US-ASCII dodanie znaku o zmienionym znaczeniu z zakresu 0 – 127 (0x00-0x7F w zapisie szesnastkowym) nie powoduje zmiany kodowania. Jeśli znak jest w zakresie 128 – 255

(0xA0-0xFF), kodowanie jest przełączane na format UTF-8. Dodanie znaku UTF-8 z zakresu 0x0000-0x007F nie wpływa na kodowanie łańcucha ASCII. Kody UTF-8 większe od wartości 0x007F sprawiają, że kodowanie łańcucha zmienia się na kodowanie UTF-8. Oto przykład:

```
>> str = "Łańcuch testowy"
=> "Łańcuch testowy"
>> str.encode!("US-ASCII")
>> str << ". Symbol euro: \u20AC"  ←❶
=> "Łańcuch testowy. Symbol euro: €"
>> str.encoding
=> #<Encoding:UTF-8>
```

Sekwencja zmiany znaczenia `\u` ❶ umożliwia wstawienie dowolnego znaku UTF-8, niezależnie od tego, czy zostanie wprowadzony bezpośrednio, czy nie.

Choć zagadnienie kodowania znaków i łańcuchów obejmuje znacznie więcej informacji, te, które zostały do tej pory zaprezentowane, są wystarczające do tego, aby być zaznajomionym z rodzajami dostępnych operacji. To, w jakim stopniu poznasz ostatecznie kodowanie, będzie zależeć od Twoich potrzeb jako programisty używającego języka Ruby. I w tym przypadku bądź świadom tego, że kodowanie będzie raczej w centrum szczególnie gorącej dyskusji oraz prac rozwojowych związanych z językiem Ruby (i nie tylko).

W tym miejscu zakończymy omówienie metod łańcuchowych i przejdziemy do klasy wykazującej silne podobieństwa do klasy `String`, ale też cechującej się kilkoma interesującymi różnicami. Mowa o klasie `Symbol`.

## 8.2. Symbole i ich zastosowanie

*Symbole* są instancjami klasy wbudowanej `Symbol` języka Ruby. Zawierają one konstruktor literałów w postaci umieszczonego na początku dwukropka. Literał symbolu zawsze możesz rozpoznać (i odróżnić go od łańcucha, nazwy zmiennej, nazwy metody lub czegokolwiek innego) dzięki następującemu tokenowi:

```
:a
:book
:"W tym miejscu wyjaśniono, jak utworzyć symbol zawierający spacje."
```

Symbol możesz również utworzyć w sposób programistyczny, wywołując metodę `to_sym` (znana też jako synonim `intern`) dla łańcucha, co zostało pokazane w poprzednim punkcie:

```
>> "a".to_sym
=> :a
>> "Przekształcanie łańcucha w symbol za pomocą metody intern....".intern
=> :"Przekształcanie łańcucha w symbol za pomocą metody intern...."
```

Zwróć uwagę na charakterystyczny — umieszczany na początku — dwukropek w wynikach określania wartości zwracanych przez narzędzie irb.

Z łatwością możesz przekształcić symbol w łańcuch:

```
>> :a.to_s
=> "a"
```

Jest to jednak zaledwie początek. Symbole różnią się od łańcuchów na różne ważne sposoby. Przyjrzyjmy się symbolom na ich własnych warunkach, a następnie powróćmy do omówienia bazującego na porównaniu symboli i łańcuchów.

### 8.2.1. Główne cechy charakterystyczne symboli

Symbole stanowią „ciężki orzech do zgryzienia" dla wielu osób uczących się języka Ruby. Nie do końca przypominają cokolwiek innego, a ponadto nie odpowiadają dokładnie typom danych, z którymi większość osób spotkała się wcześniej. Pod pewnymi względami symbole są raczej podobne do łańcuchów, ale jednocześnie mają wiele wspólnego z liczbami całkowitymi. Zdecydowanie warto przyjrzeć się bliżej ich głównym cechom charakterystycznym, czyli niezmienności i unikalności.

#### NIEZMIENNOŚĆ

Symbole są niezmienne. Nie ma miejsca taka operacja jak dołączanie znaków do symbolu. Gdy symbol już istnieje, pozostaje niezmienny. Nigdy nie napotkasz kodu `:abc << :d` lub czegoś w tym rodzaju.

Nie oznacza to, że nie istnieje symbol `:abcd`. Występuje on, ale jest całkowicie innym symbolem niż `:abc`. Nie możesz zmienić samego symbolu `:abc`. Podobnie jak liczba całkowita symbol nie może być modyfikowany. Aby odwołać się do liczby `5`, nie możesz zmodyfikować obiektu `4` przez dodanie do niego liczby `1`. Liczbę tę możesz dodać do liczby `4`, stosując wywołanie `4.+(1)` (lub `4 + 1`), ale nie możesz spowodować, że obiekt `4` stanie się obiektem `5`. Choć możesz użyć symbolu jako wskazówki dla interpretera języka Ruby, aby wygenerował inny symbol, nie masz możliwości zmiany danego symbolu.

#### UNIKALNOŚĆ

Symbole są unikatowe. Każdorazowo po napotkaniu symbolu `:abc` widoczna jest reprezentacja tego samego obiektu. I w tym przypadku pod tym względem symbole bardziej przypominają liczby całkowite niż łańcuchy. Gdy w dwóch miejscach ujrzysz notację `"abc"`, masz do czynienia z reprezentacjami dwóch różnych obiektów łańcuchowych. Konstruktor łańcucha literału `""` tworzy nowy łańcuch. Jednak symbol `:abc` zawsze jest tym samym obiektem `Symbol`, tak jak `100` zawsze jest identycznym obiektem.

Aby zauważyć różnicę między łańcuchami i symbolami dotyczącą unikalności, odpytaj obiekty odnośnie do ich identyfikatora `object_id`, który jest unikatowy dla każdego oddzielnego obiektu:

```
>> "abc".object_id
=> 2707250
>> "abc".object_id
=> 2704780
>> :abc.object_id
=> 160488
>> :abc.object_id
=> 160488
```

Notacja `"abc"` tworzy każdorazowo nowy łańcuch, co możesz stwierdzić na podstawie tego, że każdy taki łańcuch ma inny identyfikator obiektu. Jednak notacja `:abc` zawsze

reprezentuje ten sam obiekt. Symbol `:abc` identyfikuje siebie za pomocą tego samego identyfikatora liczbowego, niezależnie od tego, ile razy zostanie utworzone dotyczące go zapytanie.

Ponieważ symbole są unikatowe, nie ma potrzeby tworzenia dla nich konstruktora. W języku Ruby nie ma metody `Symbol#new`. Tworzenie symbolu nie zapewnia żadnych większych możliwości niż tworzenie liczby całkowitej. W obu przypadkach możesz jedynie się do tych obiektów odwołać.

Termin *symbol* ma szerokie konotacje. Można odnieść wrażenie, że symbol może odnosić się do dowolnego identyfikatora lub tokenu. Ważne jest zrozumienie relacji między obiektami symboli i symbolami w bardziej ogólnym znaczeniu.

### 8.2.2. Symbole i identyfikatory

Następujący kod dołącza jeden obiekt `Symbol` (`:x`) oraz identyfikator jednej zmiennej lokalnej (`s`):

```
s = :x
```

Częste jest jednak odwoływanie się do zmiennej `s` jako symbolu. *Jest* to symbol w tym sensie, że reprezentuje coś innego niż samego siebie. Okazuje się, że jednym z możliwych powodów niejasności związanych z klasą `Symbol` i obiektami symboli jest to, że obiekty te *nie* reprezentują czegoś innego niż samych siebie. W pewnym sensie nazwa zmiennej jest bardziej „symboliczna" niż sam symbol.

Istnieje związek między obiektami symboli i identyfikatorami symbolicznymi. Wewnętrznie w języku Ruby symbole są używane do śledzenia wszystkich nazw utworzonych dla zmiennych, metod i stałych. Aby wyświetlić ich listę, użyj metody klasy `Symbol.all_symbols`. Bądź świadom tego, że lista zawiera mnóstwo pozycji. Oto „wierzchołek góry lodowej":

```
>> Symbol.all_symbols
=> [:inspect, :intern, :object_id, :const_missing, :method_missing,
:method_added, :singleton_method_added, :method_removed,
:singleton_method_removed,
```

Lista jest kontynuowana i liczy ponad 3000 symboli.

Podczas przypisywania wartości do zmiennej lub stałej albo tworzenia klasy lub metody wybrany identyfikator trafia do wewnętrznej tabeli symboli języka Ruby. Techniki sprawdzające tablice pozwalają uzyskać dowód, który to potwierdza:

```
>> Symbol.all_symbols.size
=> 3118
>> abc = 1
=> 1
>> Symbol.all_symbols.size
=> 3119
>> Symbol.all_symbols.grep(/abc/)
=> [:abc]
```

**Czy użyć metody `grep` zamiast metody `include`? (Przeczytaj zamieszczoną poniżej uwagę)**

Na podstawie operacji określania wielkości tablicy zwracanej przez metodę `all_symbols` możesz stwierdzić, że zwiększa się o 1 po utworzeniu przypisania do zmiennej `abc`.

Ponadto symbol `:abc` jest teraz obecny w tablicy, co potwierdziła operacja wykonana przez metodę `grep`.

**Testy dołączenia symboli zawsze są prawdziwe**

Metoda `grep` zapewnia oparty na wyrażeniu regularnym sposób wyszukiwania w tablicy zgodnych elementów. Dlaczego po prostu nie można by użyć poniższego wiersza?

```
>> Symbol.all_symbols.include?(:abc)
```

Wynika to z tego, że wynik wykonania tego kodu zawsze będzie prawdziwy! Samo umieszczenie symbolu `:abc` w metodzie testowej `include?` powoduje wstawienie symbolu do tabeli symboli. Z tego powodu test kończy się pomyślnie nawet wtedy, gdy wcześniej nie utworzono żadnego przypisania do identyfikatora `abc`.

Tabela symboli jest tylko taką tabelą, a nie tabelą obiektów. Jeśli użyjesz identyfikatora w więcej niż jednym celu (na przykład w roli zmiennej lokalnej, a także jako nazwy metody), odpowiadający mu symbol w dalszym ciągu tylko raz będzie występować w tabeli symboli:

```
>> def abc; end  ←——— Stosuje ponownie identyfikator abc
=> :abc  ←——— Definicje metod zwracają swoje nazwy jako symbole
>> Symbol.all_symbols.size
=> 3119  ←——— Ta sama wielkość; symbol :abc występuje w tablic tylko raz
```

Ponieważ interpreter języka Ruby śledzi informacje, jakie powinny mu być znane odnośnie do symboli, może je szybko wyszukiwać. Dołączenie symbolu do tabeli symboli nie zapewnia żadnych informacji o jego przeznaczeniu.

Możesz również zauważyć, że po przypisaniu symbolu do zmiennej jest on dodawany do tabeli:

```
>> abc = :my_symbol
=> :my_symbol
>> Symbol.all_symbols.size
=> 3020
>> Symbol.all_symbols.grep(/my_symbol/)
=> [:my_symbol]
```

W tabeli są umieszczane nie tylko symbole zgodne z nazwami zmiennych i metod. Do tabeli dodawane są wszystkie symbole, jakie interpreter języka Ruby napotka gdziekolwiek w programie. To, że symbol `:my_symbol` jest zapisywany w tabeli symboli po użyciu go przez programistę, oznacza, że przy następnym skorzystaniu z niego interpreter będzie w stanie szybko go odnaleźć. W przeciwieństwie do symbolu odpowiadającego identyfikatorowi, do którego przypisano bardziej złożony obiekt, taki jak łańcuch lub tablica, symbol `:my_symbol` używany wyłącznie w takiej roli nie wymaga żadnego dodatkowego wyszukiwania. Jest on po prostu samym sobą, czyli symbolem `:my_symbol`.

Język Ruby umożliwia programiście zastosowanie tego samego mechanizmu przechowywania symboli, który jest używany przez interpreter do śledzenia identyfikatorów. Po prostu nie śledzisz identyfikatorów. Symboli używasz do własnych celów.

W dalszym ciągu korzystasz jednak z udostępniania w języku Ruby na potrzeby działań programistycznych całego mechanizmu symboli.

Jakie działania są możliwe w przypadku symboli?

### 8.2.3. Symbole w praktyce

Choć symbole mają kilka zastosowań, najczęściej można je zaliczyć do jednej z następujących dwóch kategorii: argumenty metody i klucze tablicy asocjacyjnej.

#### SYMBOLE JAKO ARGUMENTY METODY

Kilka podstawowych metod języka Ruby może pobrać symbole jako argumenty. Wiele takich metod ma również możliwość akceptowania łańcuchów. Dotychczas zaprezentowano kilka przykładów metod z rodziny `attr_*`:

```
attr_accessor :name
attr_reader :age
```

Metoda `send`, która wysyła komunikat obiektowi bez znaku kropki, może pobrać symbol:

```
"abc".send(:upcase)
```

Jeśli z góry znana jest cała nazwa metody, zwykle nie ma potrzeby stosowania metody `send`. W tym miejscu jednak jest mowa o możliwości pobierania symbolu przez tę metodę, co dotyczy nawet sytuacji, w której symbol jest przechowywany w zmiennej, a nie wprowadzany i/lub określany dynamicznie podczas działania programu.

Jednocześnie większość metod akceptujących symbole może też pobierać łańcuchy. W poprzednim przykładzie z metodą `send` symbol `:upcase` możesz zastąpić łańcuchem `"upcase"`. Po zmianie kod nadal będzie działać. Różnica polega na tym, że zapewniając symbol `:upcase`, eliminujesz konieczność dokonywania przez interpreter języka Ruby wewnętrznej translacji łańcucha `upcase` do postaci symbolu podczas określania położenia metody.

Zdarzają się też przypadki przesady. Sporadycznie napotkasz kod podobny do następującego:

```
dany_obiekt.send(nazwa_metody.to_sym)
```

W trakcie przekazywania argumentu metodzie `send` wykonywany jest dodatkowy krok (konwersja realizowana przez metodę `to_sym`). Nie ma potrzeby robienia tego, chyba że wywoływana metoda może obsługiwać tylko symbole. Jeśli metoda obsługuje łańcuchy, a ponadto istnieje łańcuch, przekaż go jej. Jeżeli wymagana jest konwersja, niech zostanie zrealizowana przez metodę.

Następna rzecz to symbole w roli kluczy tablicy asocjacyjnej. Tablicami asocjacyjnymi będziemy się obszerniej zajmować w rozdziale 9., ale użycie symboli jako ich kluczy jest wyjątkowo powszechne i warte krótkiego omówienia już w tym miejscu.

#### SYMBOLE JAKO KLUCZE TABLICY ASOCJACYJNEJ

*Tablica asocjacyjna* to struktura danych z kluczem: wartości są w niej umieszczane przez przypisanie ich do klucza. W celu pobrania wartości należy zapewnić odwołanie do klucza. W języku Ruby nie ma żadnych ograniczeń dotyczących kluczy tablicy

**Rozważ zezwolenie na użycie symboli lub łańcuchów jako argumentów metody**

Gdy tworzysz metodę pobierającą argument, który może jednak być łańcuchem lub symbolem, często warto zezwolić na nie oba. Nie jest to niezbędne w sytuacjach, w których zajmujesz się dowolnymi łańcuchami generowanymi przez użytkownika, albo wtedy, gdy używany jest tekst wczytywany z pliku. W tych przypadkach łańcuchy te nie będą mieć postaci symbolu. Jeśli jednak korzystasz z metody, która na przykład oczekuje nazwy metody lub być może wartości z ograniczonej tabeli znaczników lub etykiet, zalecane jest zezwolenie na łańcuchy lub symbole. Oznacza to zrezygnowanie z wszelkich działań względem obiektu, które wymagają od niego bycia łańcuchem lub symbolem, a ponadto spowodują błąd, jeśli obiekt nie będzie właściwego typu. Za pomocą wywołania metody `to_sym` (lub metody `to_s`, gdy wymagane jest znormalizowanie łańcuchów) możesz dokonać normalizacji argumentu, aby niezależnie od tego, co zostanie przekazane metodzie, było zgodne z operacjami niezbędnymi do wykonania.

asocjacyjnej. W roli takiego klucza możesz zastosować tablicę, klasę, inną tablicę asocjacyjną, łańcuch lub dowolny żądany obiekt. Jednak w większości przypadków prawdopodobnie użyjesz łańcuchów lub symboli.

Oto kod tworzący tablicę asocjacyjną z symbolami jako kluczami, po czym następuje pobranie jednej z wartości:

```
>> d_hash = { :name => "Dawid", :age => 55 }
=> {:name=>"Dawid", :age=>55}
>> d_hash[:age]
=> 55
```

Oto podobna tablica asocjacyjna z kluczami w postaci łańcuchów:

```
>> d_hash = { "name" => "Dawid", "age" => 55 }
=> {"name"=>"Dawid", "age"=>55}
>> d_hash["name"]
=> "Dawid"
```

Nie ma nic złego w stosowaniu łańcuchów jako kluczy tablicy asocjacyjnej, a zwłaszcza wtedy, gdy istnieje już kolekcja łańcuchów i wymagane jest uwzględnienie ich w tablicy asocjacyjnej. W przypadku kluczy tablicy asocjacyjnej symbole oferują jednak kilka korzyści.

Po pierwsze, interpreter języka Ruby może szybciej przetwarzać symbole, dlatego jeśli wykonywanych jest wiele operacji wyszukiwania w tablicy asocjacyjnej, zaoszczędzisz trochę czasu. Nie zauważysz różnicy przy przetwarzaniu jedynie niewielkiej ilości danych. Jeśli jednak niezbędne są zmiany poprawiające efektywność, klucze tablicy asocjacyjnej w postaci symboli to raczej dobry pomysł.

Po drugie, symbole dobrze prezentują się jako klucze tablicy asocjacyjnej. Oczywiście określenie „dobrze prezentują się" nie jest właściwością o charakterze technicznym, a opinia na temat tego, co dobrze wygląda, zmienia się w szerokim zakresie. Symbole przypominają jednak coś w rodzaju niezmiennej etykiety, która jest odpowiednia w sytuacjach, w których klucze tablicy asocjacyjnej są pomyślane jako identyfikatory statyczne (np. `:name` i `:age`). Z kolei łańcuchy cechują się elastycznością właściwą do reprezentowania wartości arbitralnych (np. czyjeś imię). Być może jest to przypadek rzutowania podstawy technicznej dwóch obiektów (łańcuchy podlegają

zmianom, a symbole nie) na „płaszczyźnie estetycznej". Choć może to być prawdą, programiści używający języka Ruby w roli kluczy tablicy asocjacyjnej w większym stopniu stosują symbole niż łańcuchy.

Trzecim powodem używania jako kluczy tablicy asocjacyjnej częściej symboli niż łańcuchów jest to, że w języku Ruby możliwa jest specjalna postać reprezentacji symboli w miejscu takiego klucza — w ich przypadku dwukropek znajduje się po nazwie symbolu, a nie przed nią, a ponadto usuwana jest strzałka separatora tablicy asocjacyjnej. Inaczej mówiąc, następujący wiersz:

```
hash = { :name => "Dawid", :age => 55 }
```

Może też zostać tak zapisany:

```
hash = { name: "Dawid", age: 55 }
```

Jak to często bywa, interpreter języka Ruby pozwala na zapisanie kodu w prosty i uporządkowany sposób. Oczywiście jeśli preferujesz wersję ze standardową notacją symboli i strzałkami tablicy asocjacyjnej, w dalszym ciągu możesz korzystać z takiej postaci.

Do tej pory celowo omawiano symbole głównie pod kątem tego, czym różnią się od łańcuchów. Zauważyłeś jednak, że łańcuchy regularnie pojawiają się w treści książki, niezależnie od tego, jak bardzo próbujemy rozdzielić symbole i łańcuchy. Warto skoncentrować się na symbolach, ale w tym miejscu rozszerzmy zakres omówienia, przyglądając się kilku specyficznym kwestiom związanym z porównaniem symboli i łańcuchów.

### 8.2.4. Porównanie łańcuchów i symboli

W kolejnych wersjach języka Ruby symbole coraz bardziej stawały się podobne do łańcuchów. Nie oznacza to, że symbole utraciły swoje istotne cechy. W dalszym ciągu są niezmienne i unikatowe. Prezentują jednak interfejs, który w znacznie większym stopniu niż dotychczas przypomina łańcuchy.

W ramach ogólnej demonstracji zmian poniżej podano dwie listy metod.

Pierwsza lista pochodzi z wersji 1.8.6 języka Ruby:

```
>> Symbol.instance_methods(false).sort
=> ["===", "id2name", "inspect", "to_i", "to_int", "to_s", "to_sym"]
```

Druga lista jest z wersji 2 języka Ruby:

```
>> Symbol.instance_methods(false).sort
=> [:<=>, :==, :===, :=~, :[], :capitalize, :casecmp, :downcase, :empty?,
:encoding, :id2name, :inspect, :intern, :length, :match, :next, :size,
:slice, :succ, :swapcase, :to_proc, :to_s, :to_sym, :upcase]
```

W czasie oddzielającym pojawienie się obu wersji języka symbolom zapewniono wiele nowych możliwości działań, z których większość wywodzi się z domeny właściwej łańcuchom. Zauważ jednak, że nie istnieją wersje różnych metod inkrementacji i metod zmieniających wielkość znaków, które w nazwie mają znak wykrzyknika. W przypadku łańcuchów metoda `upcase!` powoduje *natychmiastową i automatyczną zmianę znaków na duże*. Z kolei symbole są niezmienne. Symbol `:a` może pokazać symbol `:A`, ale nie może nim być.

Ogólnie rzecz biorąc, semantyka metod symboli podobnych do łańcuchów jest taka sama jak ich odpowiedników powiązanych z łańcuchami, z uwzględnieniem inkrementacji:

```
>> sym = :dawid
=> :dawid
>> sym.upcase
=> :DAWID
>> sym.succ
=> :dawid
>> sym[2]
=> "w"          ←❶
>> sym.casecmp(:dawid)
=> 0
```

Zauważ, że indeksowanie symbolu zwraca podłańcuch ❶, a nie symbol. Z punktu widzenia programisty symbole potwierdzają to, że są reprezentacjami tekstu, oferując kilka sposobów modyfikowania swojej zawartości. Nie jest to jednak prawdziwa zawartość. Symbol `:dawid` nie zawiera słowa „dawid" w takim samym stopniu, jak liczba `100` nie zawiera słowa „100". Jest to kwestia interfejsu oraz zbieżności (jak zwykle w stylu języka Ruby) teorii dotyczącej obiektów i funkcjonalności programowania.

Zasadniczo symbole bardziej przypominają liczby całkowite niż łańcuchy (tabela symboli jest właściwie tablicą asocjacyjną opartą na liczbach całkowitych). Wśród wspólnych cech symboli i liczb całkowitych są nie tylko niezmienność i unikalność, ale też bezpośredniość: zmienna, z którą powiązany jest symbol, zapewnia jego rzeczywistą wartość, a nie odwołanie do niego. Jeśli zastanawiasz się nad tym, jak dokładnie działają symbole, lub z jakiego powodu istnieją zarówno łańcuchy, jak i symbole, gdy wydają się duplikować w zakresie reprezentowania tekstu, potraktuj symbole jak przypominające liczby całkowite obiekty przybierające postać znaków. Brzmi to dziwnie, ale wiele wyjaśnia.

Cechy symboli upodabniające je do liczb całkowitych zapewniają też płynne przejście do zagadnienia obiektów liczbowych.

## 8.3. Obiekty liczbowe

W języku Ruby liczby są obiektami. Możesz do nich wysyłać komunikaty tak jak do dowolnego obiektu:

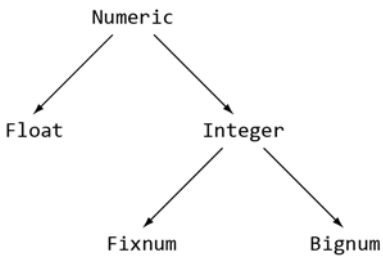
```
n = 99.6
m = n.round
puts m   ←❶
x = 12
if x.zero?
  puts "x jest zerem"
else
  puts "x nie jest zerem"  ←❷
end
puts "Znak ASCII odpowiadający liczbie 97 tos #{97.chr}"  ←❸
```

Jak się okaże, po uruchomieniu powyższego kodu liczby zmiennoprzecinkowe będą potrafiły same się zaokrąglić ❶ (w górę lub w dół). Generalnie liczby są w stanie określić ❷, czy są zerem. Liczby całkowite mają możliwość przekształcenia się w znak odpowiadający ich wartości ASCII ❸.

Liczby są obiektami, dlatego mają klasy, które tworzą całe drzewo rodziny klas.

### 8.3.1. Klasy liczbowe

Kilka klas tworzy „krajobraz" liczb. Na rysunku 8.1 pokazano trochę uproszczony widok (nie uwzględniono dołączanych modułów) tych klas, który ilustruje relacje dziedziczenia występujące między nimi.




**Rysunek 8.1.** Hierarchia klas liczbowych

`Numeric` to klasa położona na samej górze hierarchii klas liczbowych. Wszystkie inne klasy pochodzą od tej klasy. Pierwsza gałąź w drzewie znajduje się między liczbami zmiennoprzecinkowymi i liczbami całkowitymi, czyli między klasami `Float` i `Integer`. Liczby całkowite dzielą się na dwie klasy: `Fixnum` i `Bignum`. Jak możesz się domyślić, klasa `Bignum` reprezentuje duże liczby całkowite. Gdy używasz liczby całkowitej lub obliczasz liczbę całkowitą, która jest na tyle duża, aby należeć raczej do klasy `Bignum` niż do klasy `Fixnum`, w języku Ruby konwersja odbywa się automatycznie. Nie musisz się o nic martwić.

### 8.3.2. Wykonywanie operacji arytmetycznych

Przeważnie liczby w języku Ruby zachowują się tak, jak można tego oczekiwać na podstawie reguł arytmetyki oraz typowych konwencji dotyczących notacji arytmetycznych. Mało ciekawe przykłady z tabeli 8.1 powinny być przekonujące.

**Tabela 8.1.** Typowe wyrażenia arytmetyczne i wyniki operacji określania dla nich wartości

| Wyrażenie | Wynik | Komentarze |
|---|---|---|
| `1 + 1` | 2 | Dodawanie |
| `10/5` | 2 | Dzielenie liczb całkowitych |
| `16/5` | 3 | Dzielenie liczb całkowitych (bez automatycznej konwersji liczb zmiennoprzecinkowych) |
| `10/3.3` | 3.3333333333 | Dzielenie liczb zmiennoprzecinkowych |
| `1.2 + 3.4` | 4.6 | Dodawanie liczb zmiennoprzecinkowych |
| `-12 - -7` | -5 | Odejmowanie |
| `10 % 3` | 1 | Modulo (reszta z dzielenia) |

Zauważ, że podczas dzielenia liczb całkowitych wynikiem zawsze jest liczba całkowita. Aby wykonać dzielenie liczb zmiennoprzecinkowych, musisz je zapewnić interpreterowi języka Ruby (nawet wtedy, gdy cała operacja polega na dodawaniu .0 do końca liczby całkowitej).

W języku Ruby możliwe jest też modyfikowanie liczb, których bazą nie jest baza dziesiętna. Szesnastkowe liczby całkowite są identyfikowane przez umieszczony na ich początku przedrostek `0x`. Oto kilka operacji określania wartości w narzędziu irb dla wyrażeń z szesnastkowymi liczbami całkowitymi:

```
>> 0x12
=> 18
>> 0x12 + 12 ←——❶
=> 30
```

Druga liczba 12 w ostatnim wyrażeniu ❶ jest liczbą dziesiętną. Przedrostek `0x` ma zastosowanie tylko do liczb, w których się pojawia.

Liczby całkowite rozpoczynające się od zera są interpretowane jako *ósemkowe* (z bazą w postaci liczby 8):

```
>> 012
=> 10
>> 012 + 12
=> 22
>> 012 + 0x12
=> 28
```

Jak pokazano w punkcie 8.1.6, możesz też użyć łańcuchowej metody `to_i` w celu przekształcenia do postaci dziesiętnej liczb o dowolnej bazie. Aby przeprowadzić taką konwersję, jako argument metody `to_i` musisz podać bazę *źródłową* konwersji. Łańcuch jest następnie interpretowany jako liczba całkowita o podanej bazie, a całe wyrażenie zwraca odpowiednik dziesiętny. Możliwe jest zastosowanie dowolnej bazy z zakresu od 2 do 36 włącznie. Oto kilka przykładów:

```
>> "10".to_i(17)
=> 17
>> "12345".to_i(13)
=> 33519
>> "ruby".to_i(35)
=> 1194794
```

Pamiętaj o tym, że większość operatorów arytmetycznych obecnych w języku Ruby to *metody*. Nie wyglądają one na metody z powodu zapewnianego im w tym języku „lukru" składniowego, który upodabnia je do operatorów. Są to jednak metody, które mogą być wywoływane jak metody:

```
>> 1.+(1)
=> 2
>> 12./(3)
=> 4
>> -12.-(-7)
=> -5
```

W praktyce nikt nie tworzy w ten sposób operacji arytmetycznych. Zawsze napotkasz odpowiedniki w postaci „lukru" składniowego (`1 + 1` itd.). Zaznajomienie się z przykładami formy wywołania metody stanowi jednak dobry sposób na przypomnienie sobie tego, że operatory są metodami, a także tego, że jeśli na przykład we własnej klasie zdefiniujesz metodę o nazwie `+`, możesz użyć „lukru" składniowego operatora (jeżeli natrafisz na dziwnie zachowujące się operatory arytmetyczne, może to wynikać z tego, że ktoś zmodyfikował definicję ich metod bazowych).

Zajmiemy się teraz następną i ostatnią kategorią obiektów skalarnych omawianych w rozdziale. Mowa o obiektach czasu i daty.

## 8.4. Czas i daty

Język Ruby oferuje wiele sposobów modyfikowania czasu i dat. Okazuje się, że zadziwiający może być zakres i różnorodność klas reprezentujących czas i/lub datę, a także metod klas i metod instancji udostępnianych przez te klasy. To samo dotyczy różnych sposobów reprezentowania samych siebie przez instancje różnych klas. Czy chcesz wiedzieć, jaką datę miałby dzień 24 kwietnia 1705 r. w angielskim kalendarzu, który został zreformowany w 1752 r.? Jeśli tak, załaduj pakiet `date`, a następnie użyj następującego żądania:

```
>> require 'date'
=> true
>> Date.parse("April 24 1705").england.strftime("%B %d %Y")
=> "April 13 1705"
```

W ramach mniej egzotycznych zastosowań możesz przeprowadzić kilka przydatnych i wygodnych modyfikacji dotyczących obiektów czasu i daty.

Czas i daty są modyfikowane za pośrednictwem trzech klas: `Time`, `Date` i `DateTime` (dla ułatwienia instancje wszystkich tych klas mogą być zbiorczo określane mianem *obiektów czasu/dat*). Aby w pełni wykorzystać ich zalety, w programie lub sesji narzędzia irb musisz uwzględnić biblioteki `date` i `time` lub jedną z nich:

```
require 'date'
require 'time'
```

W tym przypadku pierwszy wiersz zapewnia klasy `Date` i `DateTime`, a drugi wiersz kodu rozszerza klasę `Time` (okazuje się, że jeśli nawet nie użyjesz wiersza `require 'date'`, możliwe będzie ujrzenie klasy `Date`, która jednak nie będzie jeszcze oferować żadnych możliwości działań). W przyszłości wszystkie dostępne funkcje daty i czasu mogą zostać połączone w jednej bibliotece i domyślnie udostępnione programom. Na razie jednak w celu uzyskania pełnej funkcjonalności niezbędne jest stosowanie instrukcji `require`.

W dalszej kolejności omówimy duży zestaw operacji związanych z czasem i datą. Nie będą to wszystkie operacje, lecz większość powszechnie używanych, które są wystarczające do zapewnienia podstaw do dalszego poszerzania wiedzy. Dokładniej rzecz biorąc, przyjrzymy się temu, jak tworzyć instancję obiektów daty/czasu, w jaki sposób je odpytywać, a także jak przekształcać je z jednego formatu w drugi.

### 8.4.1. Tworzenie instancji obiektów daty/czasu

To, w jaki dokładnie sposób tworzona jest instancja obiektu daty/czasu, zależy od użytego obiektu. Przyjrzymy się kolejno klasom `Date`, `Time` i `DateTime`.

#### TWORZENIE OBIEKTÓW DATY

Konstruktor `Date.today` pozwala uzyskać dzisiejszą datę:

```
>> Date.today
=> #<Date: 2013-11-02 ((2456599j,0s,0n),+0s,2299161j)
```

Uruchamiając metodę `to_s` dla daty lub stosując metodę `puts` dla tego konstruktora, możesz otrzymać prostszy łańcuch:

```
>> puts Date.today
2013-11-02
```

Możliwe jest również utworzenie obiektów daty za pomocą metody `Date.new` (dostępna też jako metoda `Date.civil`). Przekaż metodzie rok, miesiąc i dzień:

```
>> puts Date.new(1959,2,1)
1959-02-01
```

Jeśli nie podano miesiąca i dnia (lub tylko dnia), domyślnie zostanie dla nich użyta wartość 1. Jeżeli nie określono żadnych argumentów, domyślnie dla roku zostanie użyta wartość –4712, która prawdopodobnie nie należy do najbardziej przydatnych.

Możesz także utworzyć nową datę przy użyciu konstruktora `parse`, który oczekuje łańcucha reprezentującego datę:

```
>> puts Date.parse("2003/6/9")
2003-06-09
```

← **Przyjmowana jest kolejność rok/miesiąc/dzień**

Gdy podajesz liczbę jedno- lub dwucyfrową, w języku Ruby domyślnie wiek jest automatycznie uzupełniany. W przypadku liczby 69 lub większej dodawane jest przesunięcie 1900. Gdy liczba jest z przedziału od 0 do 68, przesunięciem jest 2000 (takie rozróżnienie ma związek z początkiem „epoki" w systemie Unix ustalonym na rok 1970).

```
>> puts Date.parse("03/6/9")
2003-06-09
>> puts Date.parse("33/6/9")
2033-06-09
>> puts Date.parse("77/6/9")
1977-06-09
```

Metoda `Date.parse` stara się nadać sens dowolnym przekazanym jej danym. Naprawdę świetnie sobie z tym radzi:

```
>> puts Date.parse("November 2 2013")
2013-11-02
>> puts Date.parse("Nov 2 2013")
2013-11-02
>> puts Date.parse("2 Nov 2013")
2013-11-02
>> puts Date.parse("2013/11/2")
2013-11-02
```

Metody `jd` i `commercial` umożliwiają tworzenie obiektów `Date` odpowiednio dla dat opartych na kalendarzu juliańskim i kalendarzu komercyjnym (w jego przypadku dni tygodnia są liczone, począwszy od poniedziałku, a nie od niedzieli). Możliwe jest też skanowanie za pomocą metody `strptime` łańcucha przy użyciu specyfikacji formatu, co powoduje wygenerowanie obiektu `Date`. Te techniki konstruktorów są bardziej specjalistyczne niż inne. Nie będą tutaj szczegółowo prezentowane. Jeśli jednak masz wymagania, które cechują się podobnym stopniem specjalizacji, klasa `Date` może je spełnić.

Podobnie jak klasa `Date` klasa `Time` ma wiele konstruktorów.

### TWORZENIE OBIEKTÓW CZASU

Obiekt czasu możesz utworzyć za pomocą dowolnego z kilku następujących konstruktorów: `new` (inna nazwa to `now`), `at`, `local` (inna nazwa to `mktime`) i `parse`. Taki nadmiar konstruktorów, choć początkowo może wydać się przesadny, zapewnia różne funkcje, z których każda jest przydatna. Oto kilka przykładów zaprezentowanych w sesji narzędzia irb:

```
>> Time.new
=> 2013-11-02 12:16:21 +0000                ❶
>> Time.at(100000000)
=> 1973-03-03 09:46:40 +0000                ❷
>> Time.mktime(2007,10,3,14,3,6)
=> 2007-10-03 14:03:06 +0100                ❸
>> require 'time'
=> true                                     ❹
>> Time.parse("March 22, 1985, 10:35 PM")
=> 1985-03-22 22:35:00 +0000                ❺
```

Metoda `Time.new` (lub `Time.now`) tworzy obiekt czasu reprezentujący bieżący czas ❶. Metoda `Time.at(sekundy)` zapewnia obiekt czasu dla reprezentowanej przez argument `sekundy` ❷ liczby sekund, jakie upłynęły od początku „epoki" systemu Unix (północ 1 stycznia 1970 r. czasu GMT). Metoda `Time.mktime` (lub `Time.local`) oczekuje argumentów roku, miesiąca, dnia, godziny, minut i sekund. Nie musisz podawać ich wszystkich. Gdy pominiesz argumenty po prawej stronie, metoda `Time.mktime` uzupełni je rozsądnymi wartościami domyślnymi (1 dla miesiąca i dnia, a 0 dla godziny, minut i sekund) ❸.

Aby użyć metody `Time.parse`, musisz załadować bibliotekę `time` ❹. Gdy to nastąpi, metoda ta, podobnie jak metoda `Date.parse` ❺, stara się nadać przekazanym argumentom jak największy sens.

### TWORZENIE OBIEKTÓW DATY I CZASU

Choć `DateTime` to podklasa klasy `Date`, jej konstruktory trochę się różnią z powodu zastosowanych operacji przesłaniania. Najczęściej spotykane konstruktory to `new` (dostępny również pod nazwą `civil`), `now` i `parse`:

```
>> puts DateTime.new(2009, 1, 2, 3, 4, 5)
2009-01-02T03:04:05+00:00
=> nil
>> puts DateTime.now
```

```
2013-11-03T04:44:52-08:00
=> nil
>> puts DateTime.parse("October 23, 1973, 10:34 AM")
1973-10-23T10:34:00+00:00
```

Klasa `DateTime` wyróżnia się też specjalizowanymi konstruktorami `jd` (data oparta na kalendarzu juliańskim), `commercial` i `strptime`, o których wcześniej wspomniano w odniesieniu do klasy `Date`.

Zajmijmy się teraz różnymi sposobami odpytywania obiektów daty i czasu.

### 8.4.2. Metody odpytywania obiektów daty i czasu

Przeważnie obiekty daty i czasu oferują metody do tworzenia zapytań, jakich można oczekiwać. Obiekty czasu mogą być odpytywane odnośnie do ich roku, miesiąca, dnia, godziny, minut i sekund. To samo dotyczy obiektów daty i czasu. W przypadku obiektów daty możliwe jest tworzenie zapytań dotyczących roku, miesiąca i dnia:

```
>> dt = DateTime.now
=> #<DateTime: 2014-02-21T06:33:38-05:00
((2456710j,41618s,552942000n),-18000s,2299161j)>
>> dt.year
=> 2014
>> dt.hour
=> 6
>> dt.minute
=> 33
>> dt.second
=> 38
>> t = Time.now
=> 2014-02-21 06:33:50 -0500
>> t.month
=> 2
>> t.sec
=> 50
>> d = Date.today
=> #<Date: 2014-02-21 ((2456710j,0s,0n),+0s,2299161j)>
>> d.day
=> 21
```

Zauważ, że obiekty daty i czasu mają metodę `second`, jak również metodę `sec`. Obiekty czasu dysponują jedynie metodą `sec`.

Niektóre wygodne metody powiązane z dniami tygodnia działają w jednakowy sposób w przypadku wszystkich trzech klas. Za ich pomocą możesz określić, czy dana data lub czas reprezentuje konkretny dzień tygodnia:

```
>> d.monday?
=> false
>> dt.friday?
=> true
```

Inne dostępne zapytania obejmują zapytania boolowskie odnoszące się do roku przestępnego (metoda `leap?`) i czasu letniego (metoda `dst?` tylko w przypadku obiektów czasu).

Jak pokazano, reprezentacje łańcuchów dla obiektów daty/czasu różnią się znacząco zależnie od tego, czego dokładnie zażądano, a także tego, jaka spośród trzech klas jest używana. W praktyce domyślne reprezentacje łańcuchów nie są często stosowane. Obiekty są zwykle formatowane za pomocą metod do tego przeznaczonych.

### 8.4.3. Metody formatujące datę i czas

Wszystkie obiekty daty/czasu oferują metodę `strftime`, która umożliwia w elastyczny sposób formatowanie ich pól za pomocą łańcuchów formatu. Pod względem działania metoda ta przypomina bibliotekę `strftime(3)` systemu Unix:

```
>> t = Time.now
=> 2014-02-21 06:37:59 -0500
>> t.strftime("%m-%d-%y")
=> "02-21-14"
```

W przykładzie użyto specyfikatorów formatu `%m` (miesiąc jako liczba dwucyfrowa), `%d` (dzień w postaci liczby dwucyfrowej) i `%Y` (rok w postaci liczby czterocyfrowej). Łączniki między polami są odtwarzane w danych wyjściowych dokładnie w takiej samej postaci.

W tabeli 8.2 pokazano kilka przydatnych specyfikatorów formatu metody `strftime`.

**Tabela 8.2.** Typowe specyfikatory formatu daty i czasu

| Specyfikator | Opis |
|---|---|
| `%Y` | Rok (cztery cyfry) |
| `%y` | Rok (ostatnie dwie cyfry) |
| `%b, %B` | Miesiąc w postaci krótkiej i długiej |
| `%m` | Miesiąc (liczba) |
| `%d` | Dzień miesiąca (uzupełniany zerami od lewej strony) |
| `%e` | Dzień miesiąca (uzupełniany spacjami od lewej strony) |
| `%a, %A` | Nazwa dnia w postaci krótkiej i długiej |
| `%H, %I` | Godzina (zegar 24-godzinny), godzina (zegar 12-godzinny) |
| `%M` | Minuty |
| `%S` | Sekundy |
| `%c` | Odpowiednik `"%a %b %d %H:%M:%S %Y"` |
| `%x` | Odpowiednik `"%m/%d/%y"` |

**OSTRZEŻENIE** Specyfikatory `%c` i `%x`, które obejmują wygodne kombinacje innych specyfikatorów, mogą się różnić w przypadku poszczególnych ustawień regionalnych. Na przykład w niektórych systemach w formacie `%x` dzień jest umieszczany przed miesiącem. Jest to dobre, ponieważ oznacza, że w takich formatach na stałe nie jest zdefiniowany żaden konkretny wariant ustawień regionalnych. Trzeba jednak mieć tego świadomość, aby nie liczyć na określone zachowanie, które nie zawsze może wystąpić. W razie wątpliwości możesz użyć łańcucha formatu złożonego z mniejszych specyfikatorów.

Oto kilka dodatkowych przykładów zastosowania specyfikatorów formatu:

```
>> t.strftime("Dziś jest %x")
=> "Dziś jest 11/03/13"
>> t.strftime("Inaczej znany jako %d-%b-%y")
=> "Inaczej znany jako 03-Nov-13"
>> t.strftime("A nawet dzień %e %B, %Y.")
=> "A nawet dzień 3 November, 2013."
>> t.strftime("Jest godzina %H:%m.")
=> "Jest godzina 04:11."
```

Oprócz narzędzi zapewnianych przez metodę `strftime` klasy `Date` i `DateTime` oferują zestaw wstępnie zdefiniowanych formatów danych wyjściowych na potrzeby specjalizowanych przypadków, takich jak zgodność ze standardem RFC 2822 (poczta elektroniczna) i format HTTP określony w standardzie RFC 2616:

```
>> Date.today.rfc2822
=> "Sun, 3 Nov 2013 00:00:00 +0000"
>> DateTime.now.httpdate
=> "Sun, 03 Nov 2013 12:49:48 GMT"
```

W ten lub inny sposób możesz nadać czasom i datom żądany wygląd. Obiekty daty i czasu pozwalają też na przeprowadzanie różnego rodzaju konwersji z jednej klasy obiektu do drugiej.

### 8.4.4. Metody konwersji daty i czasu

Wszystkie klasy daty i czasu umożliwiają wzajemną konwersję. Oznacza to, że klasa `Time` ma metody `to_date` i `to_datetime`, klasa `Date` zawiera metody `to_time` i `to_datetime`, a klasa `DateTime` oferuje metody `to_time` i `to_date`. W każdym przypadku, w którym klasa docelowa zawiera więcej informacji niż klasa źródłowa, dla brakujących pól ustawiana jest wartość 0 (zasadniczo reprezentuje ona północ), ponieważ wszystkie trzy klasy dysponują informacjami o datach, ale tylko dwie z nich zapewniają informacje o czasie.

Używając określonych metod i operatorów realizujących operacje arytmetyczne związane z czasem, możesz też „poruszać się” w obrębie kalendarza.

#### OPERACJE ARYTMETYCZNE ZWIĄZANE Z DATAMI I CZASEM

Obiekty czasu pozwalają na dodawanie i odejmowanie od nich sekund, co powoduje zwrócenie nowego obiektu czasu:

```
>> t = Time.now
=> 2013-11-03 04:50:49 -0800
>> t - 20
=> 2013-11-03 04:50:29 -0800
>> t + 20
=> 2013-11-03 04:51:09 -0800
```

Obiekty daty oraz obiekty daty i czasu interpretują znaki `+` i `-` jako operacje związane z dniami. Obiekty te umożliwiają przeprowadzanie konwersji dotyczących miesięcy z wykorzystaniem operatorów `<<` i `>>`:

```
>> dt = DateTime.now
=> #<DateTime: 2013-11-03T04:51:05-08:00 ... >
>> puts dt + 100
2014-02-11T04:51:05-08:00
>> puts dt >> 3
2014-02-03T04:51:05-08:00
>> puts dt << 10
2013-01-03T04:51:05-08:00
```

Możliwe jest również przejście do przodu o jedną jednostkę daty za pomocą metody `next` (inna jej nazwa to `succ`). Cała rodzina metod `next_unit` i `prev_unit` pozwala na przechodzenie do tyłu i do przodu o podaną liczbę dni, miesięcy lub lat:

```
>> d = Date.today
=> #<Date: 2013-11-03 ((2456600j,0s,0n),+0s,2299161j)>
>> puts d.next
2013-11-04
>> puts d.next_year
2014-11-03
>> puts d.next_month(3)
2014-02-03
>> puts d.prev_day(10)
2013-10-24
```

Co więcej, obiekty daty oraz obiekty daty i czasu dają możliwość iteracji dla zakresu dat za pomocą metod `upto` i `downto`, z których każda pobiera obiekt czasu, obiekt daty lub obiekt daty i czasu. Oto przykład użycia metody `upto`:

```
>> d = Date.today
=> #<Date: 2013-11-03 ((2456600j,0s,0n),+0s,2299161j)>
>> next_week = d + 7
=> #<Date: 2013-11-10 ((2456607j,0s,0n),+0s,2299161j)>
>> d.upto(next_week) {|date| puts "#{date} to #{date.strftime("%A")}" }
2013-11-03 to Sunday
2013-11-04 to Monday
2013-11-05 to Tuesday
2013-11-06 to Wednesday
2013-11-07 to Thursday
2013-11-08 to Friday
2013-11-09 to Saturday
2013-11-10 to Sunday
```

Klasy daty i czasu oferują znacznie więcej od tego, co tutaj zaprezentowano. Omówione możliwości są jednak najczęściej wykorzystywane, a ponadto z całym prawdopodobieństwem najbardziej przydatne. Nie zapomnij, że zawsze możesz użyć narzędzia wiersza poleceń `ri` do uzyskania informacji o metodach! Jeśli na przykład wypróbujesz polecenia `ri Date`, otrzymasz informacje o klasie, a także o liście dostępnych metod instancji i metod klasy, dla których możesz osobno uruchomić narzędzie `ri`.

Dotarliśmy do końca omówienia obiektów skalarnych w języku Ruby. W rozdziale 9. przyjrzymy się obiektom kolekcji i kontenerów.

## 8.5. Podsumowanie

W rozdziale przedstawiono następujące zagadnienia:

- Tworzenie i modyfikowanie łańcuchów.
- Metody działania symboli.
- Obiekty numeryczne, w tym liczby zmiennoprzecinkowe i liczby całkowite.
- Obiekty daty, obiekty czasu oraz obiekty daty i czasu, a także sposób odpytywania i modyfikowania ich.

Zostały omówione podstawy najbardziej typowych i najważniejszych obiektów skalarnych w języku Ruby. Część zagadnień uwzględniała konsolidację kwestii poruszonych wcześniej w książce, a część pojawiła się w rozdziale po raz pierwszy. Przyjrzeliśmy się też temu, jak niektóre klasy obiektów skalarnych są ze sobą powiązane. Łańcuchy i symbole reprezentują tekst. Choć są różnymi typami obiektów, przeprowadzanie konwersji między nimi jest proste i częste. Interakcja występuje również między liczbami i łańcuchami. Na przykład w przeciwieństwie do języka Perl konwersje między tymi obiektami nie odbywają się automatycznie, ale w języku Ruby są zapewniane metody dokonujące konwersji łańcucha do postaci obiektu numerycznego i odwrotnie. Są też udostępniane konwersje łańcuchów na liczby całkowite, które mogą mieć nawet bazę liczącą 10 cyfr. Konwersje te mogą również uwzględniać litery alfabetu.

Obiekty daty i czasu są powiązane zarówno z łańcuchami, jak i z obiektami numerycznymi. Możesz wykonywać w ich przypadku obliczenia, takie jak dodawanie *n* miesięcy do danej daty. Ponadto możliwe jest przetwarzanie ich jako łańcuchów z wykorzystaniem takich technik jak metoda `Time#strftime` używana w połączeniu ze specyfikatorami formatu danych wyjściowych.

Świat obiektów skalarnych w języku Ruby jest rozbudowany i dynamiczny. Większość działań podejmowanych za pomocą języka Ruby i środowiska Rails będzie bazować na zdobytej tutaj wiedzy na temat obiektów skalarnych. Mowa o takich działaniach, jak bezpośrednie modyfikowanie takich obiektów, przetwarzanie obiektów, które mają część wspólnych cech (na przykład parametry CGI ze stałymi w postaci łańcuchów), lub stosowanie kolekcji wielu obiektów skalarnych. Obiekty skalarne nie są jedynymi, ale stanowią fundament niemal całej reszty. Omówienie w rozdziale ważnych klas i metod skalarnych zapewni odpowiednie przygotowanie do prezentacji w następnym rozdziale kolekcji i kontenerów, czyli dwuwymiarowych (a czasami o większej liczbie wymiarów) „obywateli" świata obiektów języka Ruby.

# 9 Obiekty kolekcji i kontenerów

**W tym rozdziale:**

- Sekwencyjnie uporządkowane kolekcje z tablicami.
- Kolekcje z kluczem i tablicami asocjacyjnymi.
- Testy dołączania i członkostwa za pomocą zakresów.
- Unikatowe kolekcje nieuporządkowane ze zbiorami.
- Argumenty nazwane używające składni tablic asocjacyjnych.

Podczas tworzenia kodu zwykle nie przetwarza się pojedynczych obiektów, lecz ich *kolekcje*. Kolekcje są przeszukiwane w celu znalezienia obiektu spełniającego określone kryteria (np. obiekt czasopisma zawierającego konkretny artykuł). Kolekcje są sortowane pod kątem dalszego przetwarzania lub wizualnej prezentacji. Filtrowanie kolekcji służy do dołączania lub wykluczania określonych elementów itd. Wszystkie te operacje i podobne do nich zależą od obiektów dostępnych w kolekcjach.

W języku Ruby kolekcje obiektów są reprezentowane przez umieszczenie ich wewnątrz obiektów *kontenera*. W języku tym dwie klasy wbudowane zdominowały „krajobraz" obiektów kontenera: *tablice* i *tablice asocjacyjne*. Rozdział zaczniemy od omówienia klas `Array` i `Hash`: najpierw zostaną one ze sobą porównane w celu umożliwienia ich ogólnego zrozumienia, a następnie będą prezentowane osobno.

Przedstawimy dwie inne klasy: `Range` i `Set`. Zakresy są czymś w rodzaju hybrydy: po części działają one jak filtry boolowskie (w tym sensie, że mogą wykonać test prawdy/fałszu odnośnie do tego, czy dana wartość należy do określonego zakresu), ale też w przypadku niektórych kontekstów pełnią rolę kolekcji. Zbiory są w pełni kolekcjami. Jedynym powodem, dla którego klasa `Set` wymaga specjalnego wprowadzenia,

jest to, że nie jest to klasa podstawowa języka Ruby, lecz klasa biblioteki standardowej. Choć w książce nie będziemy obszernie zajmować się zbiorami, ładnie lokują się one tuż za tablicami i tablicami asocjacyjnymi, a ponadto zasługują na naszą uwagę.

W trakcie lektury rozdziału pamiętaj o tym, że stanowi on pierwszą próbę przedstawienia czegoś w rodzaju ogromnego zagadnienia, które zostanie również poruszone w następnym rozdziale. W języku Ruby kolekcje są implementowane głównie z wykorzystaniem techniki polegającej na definiowaniu klas, które dołączają moduł `Enumerable`. Moduł ten zapewnia pakiet metod, które umożliwiają sortowanie, selekcjonowanie, filtrowanie, liczenie i transformowanie kolekcji. W rozdziale przyjrzymy się przede wszystkim możliwościom oferowanym w przypadku podstawowych klas kolekcji, *innym* niż wynikające z faktu dołączania do nich modułu `Enumerable`. W rozdziale 10. zajmiemy się bezpośrednio tym modułem i sposobami jego wykorzystania. W wystarczającym stopniu zainicjujemy prezentowanie możliwości tego modułu, a następnie powrócimy do niego w następnym rozdziale.

W niniejszym rozdziale pojawi się też nawiązanie do rozdziału 2., w którym obszernie omówiono listy argumentów i parametrów metod, a także sposoby wiązania argumentów z parametrami. Po dokładniejszym przeanalizowaniu tablic asocjacyjnych uzupełnimy lukę z rozdziału 2., zajmując się *argumentami nazwanymi* języka Ruby, które korzystają ze składni tablic asocjacyjnych.

Pamiętaj też o tym, że kolekcje same są obiektami. Wysyłane są do nich komunikaty, przypisywane są one do zmiennych oraz wykonywane są inne działania w typowy dla obiektów sposób. Kolekcje mają po prostu dodatkowy wymiar, którego są pozbawione obiekty skalarne.

## 9.1. Porównanie tablic i tablic asocjacyjnych

*Tablica* to kolekcja obiektów *uporządkowana* w tym sensie, że obiekty możesz wybierać z kolekcji na podstawie spójnego indeksu kolejnych liczb. Jak zauważyłeś, tablice zostały już wykorzystane w niektórych przykładach zamieszczonych wcześniej w książce. Ciężko *nie* używać tablic w kodzie Ruby. Zadaniem tablicy jest przechowywanie innych obiektów. W tablicy może być składowany dowolny obiekt, w tym inne tablice, tablice asocjacyjne, uchwyty plików, klasy, a także obiekty `true` i `false`. Zawartość tablicy zawsze pozostaje tak samo uporządkowana, chyba że obiekty zostaną jawnie przemieszczone (albo dodane lub usunięte).

W nowszych wersjach języka Ruby *tablice asocjacyjne* również są uporządkowanymi kolekcjami. Jest to duża zmiana w porównaniu z wcześniejszymi wersjami, w których tablice asocjacyjne są nieuporządkowane (w tym sensie, że nie mają żadnych informacji o tym, jaki element jest pierwszym, ostatnim lub *n*-tym). Tablice asocjacyjne przechowują obiekty parami, a każda z nich składa się z *klucza* i *wartości*. Wartość jest pobierana za pomocą klucza. Tablice asocjacyjne zapamiętują kolejność, w jakiej ich klucze zostały wstawione. W tej kolejności tablica asocjacyjna automatycznie jest powtarzana, jeśli dla zawartych w niej par złożonych z klucza i wartości przeprowadzasz iterację lub na ekranie wyświetlasz reprezentację łańcuchową takiej tablicy.

Choć dowolny obiekt języka Ruby może pełnić rolę klucza i/lub wartości tablicy asocjacyjnej, klucze są unikatowe dla każdej takiej tablicy. Dla dowolnego danego klucza może istnieć tylko jedna para składająca się z klucza i wartości. Tablice asocjacyjne (lub podobnego typu magazyny danych) są czasami nazywane w innych językach *słownikami*. Tablice te oferują niesamowicie, a czasami zaskakująco duże możliwości przechowywania i pobierania danych.

Tablice i tablice asocjacyjne są blisko ze sobą powiązane. W pewnym sensie tablica jest tablicą asocjacyjną, w której klucze okazują się być kolejnymi liczbami całkowitymi. W pewnym sensie tablice asocjacyjne to tablice, w których indeksy mogą być czymkolwiek, a nie tylko liczbami całkowitymi. Jeśli jako kluczy tablicy asocjacyjnej używasz kolejnych liczb całkowitych, tablice i tablice asocjacyjne zaczynają zachowywać się podobnie podczas operacji wyszukiwania:

```
array = ["rubin", "diament", "szmaragd"]
hash = { 0 => "rubin", 1 => "diament", 2 => "szmaragd" }
puts array[0]    # rubin
puts hash[0]     # rubin
```

Jeśli nawet nie stosujesz liczb całkowitych, tablice asocjacyjne ujawniają swego rodzaju właściwość „metaindeksu". Wynika to z faktu, że zawierają one określoną liczbę par złożonych z klucza i wartości, które mogą być kolejno odliczane. Aby w praktyce sprawdzić tę właściwość, przetwórz krokowo tablicę asocjacyjną za pomocą metody `with_index`, która zwraca blokowi wartość licznika wraz z kluczem i wartością:

```
hash = { "czerwony" => "rubin", "biały" => "diament", "zielony" => "szmaragd" }
hash.each.with_index {|(key,value),i|
  puts "Para #{i} to: #{key}/#{value}"
}
```

Dane wyjściowe tego fragmentu kodu są następujące:

```
Para 0 to: czerwony/rubin
Para 1 to: biały/diament
Para 2 to: zielony/szmaragd
```

*Indeks* jest licznikiem w postaci liczby całkowitej zachowywanym na czas występowania par. Pary stanowią faktyczną zawartość tablicy asocjacyjnej.

> **WSKAZÓWKA** Nawiasy okrągłe używane w parametrach bloku `(key,value)` służą do podziału tablicy. Każda para klucz i wartość trafia do bloku jako tablica dwuelementowa. Jeśli parametry miałyby postać `key,value,i`, parametr `key` zostałby ostatecznie powiązany z całą tablicą `[key,value]`. Parametr `value` zostałby powiązany z indeksem, a parametr `i` byłby obiektem `nil`. Oczywiście nie czegoś takiego się oczekuje. Grupowanie w nawiasach okrągłych w postaci `(key,value)` jest sygnałem, że tablica element po elemencie ma być dystrybuowana między tymi dwoma parametrami.

Powszechne są konwersje różnego rodzaju między tablicami i tablicami asocjacyjnymi. Część z nich odbywa się automatycznie: jeśli wykonasz określone operacje polegające na wybieraniu lub wyodrębnianiu par z tablicy asocjacyjnej, w rezultacie uzyskasz

tablicę. Inne konwersje wymagają jawnych instrukcji, takich jak przekształcenie zwykłej tablicy (`[1,2,3,4]`) w tablicę asocjacyjną (`{1 => 2, 3 => 4}`). Zarówno w tym rozdziale, jak i w wielu kodach Ruby spotkasz się ze sporą liczbą konwersji odbywających się między tymi dwoma klasami kolekcji.

W następnych dwóch podrozdziałach przyjrzymy się dokładniej tablicom i tablicom asocjacyjnym. Zacznijmy od tablic.

## 9.2. Przetwarzanie kolekcji za pomocą tablic

Tablice zapewniają podstawowy sposób obsługi kolekcji obiektów. W niniejszym podrozdziale sprawdzimy tablice w praktyce: omówimy różne dostępne techniki tworzenia tablic, sposób wstawiania, pobierania i usuwania elementów tablicy, łączenie tablic ze sobą, przekształcanie (na przykład w celu „spłaszczenia" zagnieżdżonej tablicy do postaci tablicy jednowymiarowej) oraz odpytywanie ich odnośnie do ich właściwości i stanu.

### 9.2.1. Tworzenie nowej tablicy

Tablicę możesz utworzyć na jeden z czterech następujących sposobów:

- Za pomocą metody `Array.new`.
- Za pomocą konstruktora tablicy literału (nawiasy kwadratowe).
- Za pomocą metody najwyższego poziomu o nazwie `Array`.
- Za pomocą specjalnych notacji `%w{...}` i `%i{...}`.

Ponieważ wszystkie te techniki będą intensywnie wykorzystywane w kodzie Ruby, warte są poznania. Kolejno przyjrzymy się każdej z nich.

#### METODA ARRAY.NEW

Użycie metody `new` dla klasy tablicy odbywa się w następujący standardowy sposób:

```
a = Array.new
```

Możesz następnie dodać obiekty do tablicy za pomocą technik, które zostaną później przedstawione.

Metoda `Array.new` umożliwia określenie wielkości tablicy, a także, jeśli jest to wymagane, zainicjowanie jej zawartości. Oto zapis sesji narzędzia irb, który ilustruje obie możliwości:

```
>> Array.new(3)          ←❶
=> [nil, nil, nil]
>> Array.new(3,"abc")    ←❷
=> ["abc", "abc", "abc"]
```

Jeśli metodzie `Array.new` zapewnisz jeden argument ❶, uzyskasz tablicę o żądanej wielkości z wszystkimi elementami ustawionymi na wartość `nil`. W przypadku podania dwóch argumentów ❷ otrzymasz tablicę o żądanej wielkości, w której każdy zainicjowany element będzie zawierał drugi argument.

Możliwe jest nawet przekazanie metodzie `Array.new` bloku kodu. W tym przypadku elementy tablicy są inicjowane przez powtarzane wywołania bloku:

```
>> n = 0
=> 0
>> Array.new(3) { n += 1; n * 10 }  ←❶
=> [10, 20, 30]                     ←❷
```

W tym przykładzie nowa tablica ma wielkość 3. Dla każdego z trzech elementów ustawiana jest wartość zwracana z bloku kodu. Kod wewnątrz bloku ❶ wykonywany trzykrotnie generuje wartości `10`, `20` i `30`. Są to wartości początkowe w tablicy ❷.

> **OSTRZEŻENIE** Podczas inicjowania wielu elementów tablicy za pomocą drugiego argumentu metody `Array.new`, tak jak w przypadku metody `Array.new(3, "abc")`, wszystkie elementy są inicjowane jako ten sam obiekt. Jeśli użyjesz kodu `a = Array.new(3,"abc"); a[0] << "def"; puts a[1]`, okaże się, że drugi element tablicy to teraz `"abcdef"` nawet pomimo tego, że do pierwszego elementu dołączono `"def"`. Wynika to stąd, że na pierwszej i drugiej pozycji w tablicy znajduje się ten sam obiekt łańcucha, a nie dwa różne łańcuchy, które są złożone ze znaków `"abc"`. Aby utworzyć tablicę wstawiającą na każdej pozycji inny łańcuch `"abc"`, należy zastosować kod `Array.new(3) { "abc" }`. Blok kodu jest trzykrotnie wykonywany, generując za każdym razem nowy łańcuch (te same znaki, inny obiekt łańcucha).

Wstępne inicjowanie tablic nie zawsze jest konieczne, ponieważ tablice zwiększają się wraz z dodawaniem do nich elementów. Jeśli jednak taka funkcjonalność jest niezbędna, a ponadto spotkałeś się z jej użyciem i chcesz ją zrozumieć, jest ona dostępna.

### KONSTRUKTOR TABLICY LITERAŁU: []

Drugi sposób tworzenia tablicy polega na zastosowaniu *konstruktora tablicy literału* w postaci `[]` (nawiasy kwadratowe):

```
a = []
```

Podczas tworzenia tablicy za pomocą konstruktora literału możesz jednocześnie umieszczać w niej obiekty:

```
a = [1,2,"trzy",4, []]
```

Zauważ, że ostatni element w tej tablicy jest inną tablicą. Jest to w pełni dozwolone. Tablice mogą być zagnieżdżane przy użyciu dowolnej żądanej liczby poziomów.

W języku Ruby nawiasy kwadratowe mogą oznaczać wiele różnych rzeczy: konstrukcję tablicy, indeksowanie tablicy (a także indeksowanie łańcuchów i tablic asocjacyjnych), klasy znaków w wyrażeniach regularnych, ograniczniki w notacji łańcuchów opartej na formacie `%q[]`, a nawet wywołanie funkcji anonimowej. W celu dokonania początkowego podziału różnych zastosowań nawiasów kwadratowych możesz odróżnić przypadki, w których są one konstrukcją semantyczną, od przypadków, gdzie pełnią one rolę nazwy metody. Warto sprawdzić kilka przykładów podobnych do poniższego, aby zorientować się, jaką rolę nawiasy kwadratowe odgrywają w różnych kontekstach:

```
[1,2,3][1]  ←——— Indeks 1 tablicy [1,2,3]
```

Pora wrócić do tworzenia tablic.

### METODA ARRAY

Trzecim sposobem tworzenia tablicy jest wykorzystanie *metody* (nawet pomimo tego, że ma nazwę przypominającą nazwę klasy!) o nazwie `Array`. Jak już wiesz po zaprezentowaniu metod `Integer` i `Float`, dozwolone jest definiowanie metod, których nazwy rozpoczynają się od dużych liter. Takie nazwy wyglądają dokładnie jak nazwy stałych. W samym języku Ruby metody o nazwie zaczynającej się od dużej litery mają zwykle takie same nazwy jak klasy, z którymi są powiązane.

**Więcej metod wbudowanych o nazwach zaczynających się od dużych liter**

Oprócz metody `Array` i dwóch wcześniej przedstawionych metod konwersji z nazwą rozpoczynającą się od dużej litery (metody `Integer` i `Float`, czyli „wybredne" wersje metod `to_i` i `to_f`) język Ruby zapewnia kilka innych metod najwyższego poziomu, których nazwy przypominają nazwy klas: `Complex`, `Rational` i `String`. W każdym przypadku metoda zwraca obiekt klasy, której nazwa wygląda podobnie.

Metoda `String` opakowuje metodę `to_s`, co oznacza, że `String(obj)` jest równoważne `String(obj)`. Metody `Complex` i `Rational` odpowiadają metodom `to_c` i `to_r` dostępnym w przypadku liczb i łańcuchów, z tym wyjątkiem, że metody `Complex` i `Rational`, tak jak metody `Integer` i `Float`, są „wybredne": nie akceptują wprost łańcuchów nienumerycznych. Wywołanie `"abc".to_c` zapewni wynik `(0+0i)`, ale wywołanie `Complex("abc")` spowoduje zgłoszenie błędu `ArgumentError`. Metody `Rational` i `to_r` zachowują się podobnie.

Nie uwzględniamy tutaj liczb wymiernych i zespolonych, ale na wypadek, gdyby okazały się dla Ciebie interesujące, wiesz już, jak je wygenerować!

Metoda `Array` tworzy tablicę przy użyciu swojego pojedynczego argumentu. Jeśli obiekt argumentu ma zdefiniowaną metodę `to_ary`, metoda `Array` wywołuje ją dla obiektu w celu wygenerowania tablicy (pamiętaj o tym, że metoda `to_ary` to metoda konwersji tablic, która pozornie przeprowadza rzutowanie typów). W przypadku braku metody `to_ary` metoda `Array` próbuje wywołać metodę `to_a`. Jeśli ta metoda również nie jest zdefiniowana, metoda `Array` opakowuje obiekt w tablicy i zwraca następujący wynik:

```
>> string = "Łańcuch"
=> "Łańcuch"
>> string.respond_to?(:to_ary)
=> false
>> string.respond_to?(:to_a)
=> false
>> Array(string)    ←——— ❶
=> ["Łańcuch"]
>> def string.to_a  ←——— ❷
>>   split(//)
>> end
=> nil
>> Array(string)
=> ["Ł", "a", "ń", "c", "u", "c", "h"]
```

W tym przykładzie pierwsza próba uruchomienia metody `Array` dla łańcucha ❶ powoduje zwrócenie jednoelementowej tablicy, w której jedynym elementem jest łańcuch. Wynika to stąd, że łańcuchy nie zawierają ani metody `to_ary`, ani metody `to_a`. Jednak po zdefiniowaniu metody `to_a` dla łańcucha ❷ inny jest wynik wywołującej metody `Array`: tym razem uruchamia ona metodę `to_a` i używa jej jako swojej wartości zwracanej (metoda `to_a` dzieli łańcuch na osobne znaki).

Wśród różnych konstruktorów tablic literał `[]` jest najpowszechniejszy, a za nim plasują się kolejno metody `Array.new` i `Array`. Każdy z tych konstruktorów ma jednak swoje miejsce. Konstruktor literału jest najbardziej zwięzły. Gdy poznasz jego znaczenie, to po napotkaniu go od razu będziesz wiedzieć, że chodzi o „tablicę". Metoda `Array` jest ograniczona koniecznością udostępnienia metody `to_ary` lub `to_a`.

### KONSTRUKTORY TABLIC %W I %W

W celu ułatwienia tworzenia tablic łańcuchów w języku Ruby zapewniono specjalny operator `%w`. Należy on do tej samej rodziny operatorów co wcześniej omówione operatory `%q`. Operator `%w` automatycznie generuje tablicę łańcuchów przy użyciu łańcuchów rozdzielonych spacją umieszczanych wewnątrz tablicy. Aby dowiedzieć się, jak to przebiega w praktyce, w narzędziu irb użyj następującego wiersza i przyjrzyj się uzyskanemu wynikowi:

```
>> %w{ Dawid A. Nowak }
=> ["Dawid", "A.", "Nowak"]
```

Jeśli dowolny łańcuch na liście zawiera biały znak, musisz zmienić jego znaczenie za pomocą znaku `\`:

```
>> %w{ Dawid\ A.\ Nowak używa języka Ruby. }
=> ["Dawid A. Nowak", "używa", "języka", "Ruby."]
```

Łańcuchy na liście są analizowane jako łańcuchy ujęte w apostrofy. Jeśli jednak niezbędne jest umieszczenie łańcuchów w cudzysłowie, zamiast operatora `%w` możesz użyć operatora `%W`:

```
>> %W{ Dawid ma #{2014 - 1959} lat. }
=> ["Dawid", "ma", "55", "lat."]
```

### KONSTRUKTORY TABLIC %I I %I

Tak jak jest w przypadku tworzenia tablic za pomocą operatorów `%w` i `%W`, możliwe jest też tworzenie tablic symboli z wykorzystaniem operatorów `%i` i `%I`. Podobnie jak jest w przypadku operatorów `%w` i `%W`, rozróżnienie operatorów `%i` i `%I` dotyczy interpretowania łańcuchów ujętych w apostrofy lub cudzysłów:

```
>> %i{ a b c }
=> [:a, :b, :c]
>> d = "Dawid"
=> "Dawid"
>> %I{"#{d}"}
=> [:"\"Dawid\""]
```

Przejdźmy teraz do kwestii obsługi elementów tablicy.

**Rodzina metod try_convert**

Każda z kilku klas wbudowanych w języku Ruby zawiera metodę klasy o nazwie `try_convert`, która zawsze pobiera jeden argument. Metoda szuka metody konwersji w obiekcie argumentu. Jeśli metoda istnieje, zostanie wywołana. W przeciwnym razie metoda `try convert` zwraca obiekt `nil`. Jeżeli metoda konwersji zwraca obiekt klasy innej niż klasa docelowa dokonywanej konwersji, zgłaszany jest błąd krytyczny (`TypeError`).

Klasy implementujące metodę `try_convert` (oraz nazwy wymaganych metod konwersji) są następujące: `Array` (`to ary`), `Hash` (`to hash`), `IO` (`to io`), `Regexp` (`to regexp`) i `String` (`to_str`). Poniżej zamieszczono przykład obiektu korzystającego z metody `Array.try_convert` (inne metody `try convert` działają podobnie).

```
>> obj = Object.new
=> #<Object:0x000001028033a8>
>> Array.try convert(obj)
=> nil
>> def obj.to ary
>>   [1,2,3]
>> end
=> :to_ary
>> Array.try_convert(obj)
=> [1, 2, 3]
>> def obj.to_ary
>>   "To nie jest tablica!"
>> end
=> :to ary
>> Array.try convert(obj)
TypeError: can't convert Object to Array (Object#to_ary gives String
```

### *9.2.2. Wstawianie, pobieranie i usuwanie elementów tablicy*

Tablica to kolekcja uporządkowana numerycznie. Dowolny obiekt dodany do tablicy trafia na jej początek, na koniec lub gdzieś do środka. Najbardziej ogólna technika wstawiania do tablicy jednego lub większej liczby elementów polega na zastosowaniu metody ustawiającej `[]=` (nawiasy kwadratowe i znak równości). Choć metoda ta wygląda dziwnie w środku tego akapitu, dzięki swojemu odpowiednikowi w postaci „lukru” składniowego w praktyce działa bezproblemowo.

Aby użyć metody `[]=`, musisz wiedzieć, że każdy element w tablicy zajmuje miejsce o określonej liczbie. Pierwszy element znajduje się na pozycji *zero* (a nie *jeden*). Drugi element jest na pozycji jeden itd.

W celu wstawienia elementu za pomocą metody `[]=` (używając „lukru” składniowego, który umożliwia uniknięcie standardowego znaku kropki w wywołaniu metody) zastosuj następujące wiersze:

```
a = []
a[0] = "pierwszy"
```

Drugi wiersz stanowi „lukier” składniowy dla wywołania `a.[]=(0,"pierwszy")`. W tym przykładzie uzyskasz jednoelementową tablicę, której pierwszym i jedynym elementem jest łańcuch `"pierwszy"`.

Gdy w tablicy występują obiekty, możesz je pobrać za pomocą metody `[]`, która jest odpowiednikiem pobierającym metody ustawiającej `[]=`:

```
a = [1,2,3,4,5]
p a[2]
```

W tym przypadku drugi wiersz to „lukier” składniowy dla wywołania `a.[](2)`. Żądasz trzeciego elementu (zakładając, że indeksowanie rozpoczyna się od pozycji zerowej), którym jest liczba całkowita 3.

Powyższe metody pobierające i ustawiające możesz też zastosować dla więcej niż jednego elementu jednocześnie.

### USTAWIANIE LUB POBIERANIE JEDNOCZEŚNIE WIĘCEJ NIŻ JEDNEGO ELEMENTU TABLICY

Jeśli metodzie `Array#[]` lub `Array#[]=` (metoda pobierająca lub ustawiająca) przekazano drugi argument, jest on traktowany jako długość, czyli liczba elementów do ustawienia lub pobrania. W przypadku pobierania wyniki są zwracane wewnątrz nowej tablicy.

Oto przebieg sesji w narzędziu irb, ilustrujący operacje na wielu elementach wykonywane przez metody `[]` i `[]=`:

```
>> a = ["czerwony","pomarańczowy","żółty","purpurowy","szary","indygo","fioletowy"]
=> ["czerwony","pomarańczowy","żółty","purpurowy","szary","indygo","fioletowy"]
>> a[3,2]                                  ❶
=> ["purpurowy", "szary"]
>> a[3,2] = "zielony", "niebieski"         ❷   „Lukier” składniowy dla wywołania
=> ["zielony", "niebieski"]                    a.[]=(3,2,["zielony", "niebieski"]
>> a
=> ["czerwony","pomarańczowy","żółty","zielony","niebieski","indygo","fioletowy"]  ❸
```

Po zainicjowaniu tablicy `a` pobierane są dwa elementy ❶, począwszy od indeksu 3 (czwarty element) tablicy `a`. W tablicy zwracane są dwa elementy. Następnie dla czwartego i piątego elementu ustawiane są nowe wartości za pomocą notacji `[3,2]` ❷. Gdy później w narzędziu irb zażądamy na koniec wyświetlenia tablicy wartości, te są obecne w niej całej ❸.

Dla metody `[]` istnieje synonim `slice`. Podobnie jak metoda `[]` metoda `slice` pobiera dwa argumenty: indeks początkowy i opcjonalną długość. Ponadto metoda o nazwie `slice!` usuwa trwale z tablicy elementy zmodyfikowane przez metodę `slice`.

Inna technika wyodrębniania wielu elementów tablicy polega na użyciu metody `values_at`. Pobiera ona co najmniej jeden argument reprezentujący indeksy i zwraca tablicę składającą się z wartości przechowywanych w tych indeksach w tablicy odbiorcy:

```
array = ["osa", "pies", "ala", "las", "kot"]
articles = array.values_at(0,3)
p articles                  Dane wyjściowe: ["osa", "las"]
```

Operacje ustawiania i pobierania mogą być wykonywane na elementach znajdujących się w dowolnym miejscu tablicy. Jednak operacje dotyczące w szczególności początku i końca tablic występują najczęściej. W związku z tym, jak się przekonasz, istnieje kilka metod przeznaczonych specjalnie do dodawania lub usuwania elementów z początku lub końca tablicy.

### SPECJALNE METODY MODYFIKUJĄCE POCZĄTEK I KONIEC TABLIC

Aby dodać obiekt na początku tablicy, możesz użyć metody `unshift`. Po wykonaniu następującej operacji:

```
a = [1,2,3,4]
a.unshift(0)
```

tablica `a` przyjmie postać: `[0,1,2,3,4]`.

W celu dodania obiektu na końcu tablicy użyj metody `push`. Wykonanie następującego kodu:

```
a = [1,2,3,4]
a.push(5)
```

spowoduje uzyskanie tablicy z piątym elementem: `[1,2,3,4,5]`.

Możesz też zastosować metodę o nazwie << (dwa znaki mniejszości), która zastępuje obiekt na końcu tablicy. Podobnie jak wiele metod, których nazwy przypominają operatory, metoda << oferuje użycie „lukru" składniowego w postaci operatora wrostkowego. Następujący kod dodaje liczbę `5` jako piąty element tablicy `a`, tak jak metoda `push` w poprzednim przykładzie:

```
a = [1,2,3,4]
a << 5
```

Metody << i `push` różnią się tym, że druga z nich pobiera więcej niż jeden argument. Następujący kod:

```
a = [1,2,3,4,5]
a.push(6,7,8)
```

dodaje trzy elementy do tablicy `a`, co powoduje, że ma ona postać: `[1,2,3,4,5,6,7,8]`.

Odpowiednikami metod `unshift` i `push`, lecz z odwrotnym efektem, są metody `shift` i `pop`. Metoda `shift` usuwa jeden obiekt z początku tablicy (powodując „przesunięcie" pozostałych obiektów o jedną pozycję w lewo), a metoda `pop` usuwa obiekt z jej końca. Metody `shift` i `pop` zwracają usunięty przez siebie element tablicy, co prezentuje następujący przykład:

```
a = [1,2,3,4,5]
print "Oryginalna tablica: "
p a
popped = a.pop
print "Element usunięty przez metodę pop: "
puts popped
print "Nowy stan tablicy: "
p a
shifted = a.shift
print "Element usunięty przez metodę shift: "
puts shifted
print "Nowy stan tablicy: "
p a
```

Oto dane wyjściowe:

```
Oryginalna tablica: [1, 2, 3, 4, 5]
Element usunięty przez metodę pop: 5
```

```
Nowy stan tablicy: [1, 2, 3, 4]
Element usunięty przez metodę shift: 1
Nowy stan tablicy: [2, 3, 4]
```

Na podstawie kolejnych komentarzy w danych wyjściowych możesz stwierdzić, że wartością zwracaną metod `shift` i `pop` jest element usunięty z tablicy. Tablica jest trwale modyfikowane przez operacje realizowane przez te metody. Elementy są usuwane, a nie tylko przywoływane lub przechwytywane.

Metody `shift` i `pop` mogą usunąć jednocześnie więcej niż jeden element. Wystarczy podać argument w postaci liczby całkowitej, a zostanie usunięta liczba elementów równa tej wartości. Usunięte elementy będą zwracane jako tablica (nawet wtedy, gdy podano liczbę 1):

```
>> a = %w{ jeden dwa trzy cztery pięć }
=> ["jeden", "dwa", "trzy", "cztery", "pięć"]
>> a.pop(2)
=> ["cztery", "pięć"]
>> a
=> ["jeden", "dwa", "trzy"]
>> a.shift(2)
=> ["jeden", "dwa"]
>> a
=> ["trzy"]
```

Dalej przejdziemy od modyfikowania jednej tablicy do sposobów łączenia dwóch lub większej liczby tablic.

### 9.2.3. Łączenie tablic z innymi tablicami

Kilka metod umożliwia łączenie wielu tablic na różne sposoby. Jak się okazuje, coś takiego jest często wykorzystywane i przydatne, gdy zaczynasz modyfikować wiele danych na listach. Pamiętaj, że w każdym przypadku, nawet pomimo tego, że przetwarzasz dwie lub większą liczbę tablic, jedna tablica zawsze jest odbiorcą komunikatu. Inne tablice objęte operacją są argumentami metody.

Aby dodać zawartość jednej tablicy do innej tablicy, możesz użyć metody `concat`:

```
>> [1,2,3].concat([4,5,6])
=> [1, 2, 3, 4, 5, 6]
```

Zauważ, że metoda `concat` różni się w istotny sposób od metody `push`. Spróbuj w powyższym przykładzie metodę `concat` zastąpić metodą `push`, aby zobaczyć, co się stanie.

Metoda `concat` trwale zmienia zawartość swojego odbiorcy. W celu połączenia dwóch tablic w trzecią, nową tablicę możesz skorzystać z metody `+`:

```
>> a = [1,2,3]
=> [1, 2, 3]
>> b = a + [4,5,6]
=> [1, 2, 3, 4, 5, 6]
>> a
=> [1, 2, 3]  ←——❶
```

Odbiorca komunikatu `+`, którym w tym przypadku jest tablica `a`, pozostaje niezmieniony przez operację (co potwierdza zapis sesji narzędzia irb ❶).

`replace` (zastąp) to kolejna przydatna (a przynajmniej w odniesieniu do dość swobodnej interpretacji pojęcia „łączenia”) metoda łącząca tablice. Jak wskazuje nazwa metody, zastępuje ona zawartość tablicy zawartością innej tablicy:

```
>> a = [1,2,3]
=> [1, 2, 3]
>> a.replace([4,5,6])  ←❶
=> [4, 5, 6]
>> a
=> [4, 5, 6]
```

Oryginalna zawartość tablicy `a` została zastąpiona ❶ przez zawartość tablicy argumentu `[4,5,6]`. Pamiętaj, że operacja realizowana przez metodę `replace` różni się od operacji ponownego przypisania. Jeśli użyjesz następujących wierszy:

```
a = [1,2,3]
a = [4,5,6]
```

drugie przypisanie spowoduje odwołanie zmiennej `a` do całkowicie innego obiektu tablicy niż w przypadku przypisania w pierwszym wierszu. Nie jest to jednoznaczne z zastępowaniem elementów *tego samego* obiektu tablicy. Zaczyna to mieć znaczenie szczególnie w sytuacji, gdy istnieje inna zmienna odwołująca się do oryginalnej tablicy, tak jak w następującym kodzie:

```
>> a = [1,2,3]
=> [1, 2, 3]
>> b = a        ←❶
=> [1, 2, 3]
>> a.replace([4,5,6])
=> [4, 5, 6]
>> b            ←❷
=> [4, 5, 6]
>> a = [7,8,9]  ←❸
=> [7, 8, 9]
>> b
=> [4, 5, 6]    ←❹
```

Po zdefiniowaniu przypisania zmiennej `a` do zmiennej `b` ❶ zastępowanie zawartości zmiennej `a` oznacza, że zastąpiono zawartość zmiennej `b` ❷, ponieważ dwie zmienne odwołują się do tej samej tablicy. Gdy jednak zmienisz przypisanie do zmiennej `a` ❸, spowodujesz przerwanie powiązania między tą zmienną i tablicą. W efekcie zmienne `a` i `b` będą odwoływać się do różnych obiektów tablicy: zmienna `b` do tej samej, starej tablicy ❹, a zmienna `a` do nowej tablicy.

Oprócz łączenia wielu tablic możesz też dokonywać transformacji poszczególnych tablic do różnych postaci. W następnym punkcie przyjrzymy się związanym z tym technikom.

### *9.2.4. Transformacje tablic*

`flatten` to przydatna metoda transformacji tablic, która usuwa zagnieżdżenie tablic wewnętrznych. Możesz określić żądaną liczbę usuwanych poziomów zagnieżdżenia. W domyślnym wariancie metoda całkowicie usuwa zagnieżdżenie.

Oto tablica z potrójnym zagnieżdżeniem przetwarzana przez metodę `flatten` na różnych poziomach:

```
>> array = [1,2,[3,4,[5],[6,[7,8]]]]
=> [1, 2, [3, 4, [5], [6, [7, 8]]]]
>> array.flatten                  ← Całkowicie usuwa zagnieżdżenie
=> [1, 2, 3, 4, 5, 6, 7, 8]
>> array.flatten(1)               ← Usuwa zagnieżdżenie na jednym poziomie
=> [1, 2, 3, 4, [5], [6, [7, 8]]]
>> array.flatten(2)
=> [1, 2, 3, 4, 5, 6, [7, 8]]     ← Usuwa zagnieżdżenie na dwóch poziomach
```

Istnieje też metoda `flatten!`, która wprowadza trwałe zmiany w tablicy.

`reverse` (odwróć) to kolejna metoda transformacji tablic, która działa dokładnie tak, jak wskazuje na to jej nazwa:

```
>>[1,2,3,4].reverse
=>  [4, 3, 2, 1]
```

Metoda `Array#reverse` — podobnie jak jej odpowiednik łańcuchowy — dysponuje wersją metody z nazwą zawierającą wykrzyknik, która trwale odwraca wywoływaną tablicę.

Kolejną ważną metodą transformacji tablic jest metoda `join`. Jej wartością zwracaną nie jest tablica, lecz łańcuch złożony z reprezentacji łańcuchowej wszystkich połączonych ze sobą elementów tablicy:

```
>> ["abc", "def", 123].join
=> "abcdef123"
```

Metoda `join` pobiera opcjonalny argument. Jeśli zostanie podany, umieszczany jest między każdą parą elementów:

```
>> ["abc", "def", 123].join(", ")
=> "abc, def, 123"
```

Łączenie za pomocą przecinków (lub przecinka i spacji, jak w ostatnim przykładzie) jest dość częstą operacją.

W obszernym przykładzie stylu projektowania w języku Ruby występuje kolejny sposób łączenia tablicy, czyli metoda *. Choć jej działanie wygląda na mnożenie tablicy przez łańcuch, w rzeczywistości ma miejsce operacja łączenia:

```
>> a = %w{ jeden dwa trzy }
=> ["jeden", "dwa", "trzy"]
>> a * "-"
=> "jeden-dwa-trzy"
```

Możesz również dokonać transformacji tablicy za pomocą metody `uniq`. Oferuje ona nową tablicę złożoną z elementów oryginalnej tablicy z usuniętymi wszystkimi zduplikowanymi elementami:

```
>> [1,2,3,1,4,3,5,1].uniq
=> [1, 2, 3, 4, 5]
```

Status zduplikowania jest określany przez sprawdzanie par elementów za pomocą metody ==. Dowolne dwa elementy, dla których test przy użyciu metody == zwraca

wartość `true`, są uważane za duplikujące się wzajemnie. Metoda `uniq` dysponuje też wersją o nazwie zawierającej znak `!`, która trwale usuwa duplikaty z oryginalnej tablicy.

Czasami istnieje tablica zawierająca co najmniej jedno wystąpienie wartości `nil`, które ma zostać usunięte. Na przykład może zostać zdefiniowana tablica kodów pocztowych dla wszystkich członków organizacji. Być może jednak niektórzy członkowie nie mają kodu pocztowego. Aby utworzyć histogram dla kodów pocztowych, należałoby najpierw usunąć wystąpienia wartości `nil`.

W tym celu możesz zastosować metodę `compact`. Zwraca ona nową tablicę identyczną jak oryginalna tablica, z tą różnicą, że zostały usunięte wszystkie wystąpienia wartości `nil`.

```
>> zip_codes = ["06511", "08902", "08902", nil, "10027",
"08902", nil, "06511"]
=> ["06511", "08902", "08902", nil, "10027", "08902", nil, "06511"]
>> zip_codes.compact
=> ["06511", "08902", "08902", "10027", "08902", "06511"]
```

I tym razem dostępna jest wersja metody z wykrzyknikiem w nazwie (`compact!`).

Oprócz tego, że możesz zmieniać tablice na różne sposoby, możesz też je odpytywać przy użyciu różnych kryteriów.

### 9.2.5. Odpytywanie tablic

Kilka metod umożliwia pobieranie z tablicy informacji o niej. W tabeli 9.1 podsumowano część tych metod. Inne metody odpytujące wynikają z włączenia modułu `Enumerable` klasy `Array`. Oznacza to, że zostaną one zaprezentowane w następnym rozdziale.

**Tabela 9.1.** Podsumowanie typowych metod odpytujących tablice

| Nazwa metody/ przykładowe wywołanie | Znaczenie |
|---|---|
| `a.size` (synonim: `length`) | Liczba elementów w tablicy. |
| `a.empty?` | Wartość prawda, gdy tablica `a` jest pusta, natomiast wartość fałsz, jeśli tablica zawiera jakiekolwiek elementy. |
| `a.include?(item)` | Wartość prawda, gdy tablica zawiera elementy, natomiast wartość fałsz w przeciwnym razie. |
| `a.count(item)` | Liczba wystąpień elementu `item` w tablicy. |
| `a.first(n=1)` | *n* pierwszych elementów tablicy. |
| `a.last(n=1)` | *n* ostatnich elementów tablicy. |
| `a.sample(n=1)` | *n* losowych elementów tablicy. |

Jeśli w przypadku metod `first`, `last` i `sample` nie przekażesz argumentu, jako wynik uzyskasz po prostu jeden element. Jeżeli przekażesz argument *n*, zostanie zwrócona tablica *n* elementów (nawet jeśli *n* to 1).

Następne zagadnienie to tablice asocjacyjne. Dotychczas pojawiały się one w różnych miejscach książki. Pora przyjrzeć się im szczegółowo.

## 9.3. Tablice asocjacyjne

Tablica asocjacyjna — podobnie jak zwykła — stanowi kolekcję obiektów. Tablica asocjacyjna składa się z par klucz i wartość, w których klucz i wartość mogą być każdym obiektem języka Ruby. Tablice asocjacyjne umożliwiają wykonywanie operacji wyszukiwania z wykorzystaniem kluczy. Oprócz prostego pobierania wartości na podstawie klucza możesz też wykonywać bardziej złożone operacje wybierania i filtrowania.

Typowym zastosowaniem tablicy asocjacyjnej jest przechowywanie całych łańcuchów wraz z ich skrótami. Poniżej zaprezentowano tablicę asocjacyjną zawierającą wybrane nazwy województw i ich dwuliterowe skróty, a także kod, który sprawdza te dane. Operator => łączy klucz po lewej stronie z odpowiadającą mu wartością po prawej stronie:

```
state_hash = { "Lubelskie"          => "LU",
               "Małopolskie"        => "MA",
               "Wielkopolskie"      => "WI",
               "Zachodniopomorskie" => "ZA" }
print "Podaj nazwę województwa: "
state = gets.chomp
abbr = state_hash[state]
puts "Skrót to: #{abbr}."
```

Po uruchomieniu tego fragmentu kodu (przy założeniu, że wprowadzono jedno z województw zdefiniowanych w tablicy asocjacyjnej) zostanie wyświetlony skrót.

Tablice asocjacyjne zapamiętują kolejność wstawiania ich kluczy. Kolejność ta nie zawsze jest szczególnie istotna. Jedną z zalet tablicy asocjacyjnej jest to, że zapewnia możliwość szybkiego wyszukiwania w czasie krótszym niż w przypadku wyszukiwania liniowego. W wielu sytuacjach elementy nie są dodawane do tablic asocjacyjnych w żadnej konkretnej kolejności. Uporządkowanie, jeśli w ogóle, pojawia się później, gdy na przykład chcesz przekształcić utworzoną z upływem czasu tablicę asocjacyjną imion i dat urodzin w tablicę posortowaną chronologicznie lub alfabetycznie. Niemniej jednak niezależnie od tego, czy będzie to przydatne, czy nie, tablice asocjacyjne zapamiętują kolejność wstawiania swoich kluczy, a ponadto monitorują ją podczas wykonywania dla kluczy operacji iteracji lub sprawdzania.

Tablice asocjacyjne — tak jak i zwykłe — mogą być tworzone na kilka różnych sposobów.

### 9.3.1. Tworzenie nowej tablicy asocjacyjnej

Istnieją następujące cztery sposoby tworzenia tablicy asocjacyjnej:

- Za pomocą konstruktora literału (nawiasy klamrowe).
- Za pomocą metody `Hash.new`.
- Za pomocą metody `Hash.[]` (metoda klasy `Hash` w postaci nawiasów kwadratowych).
- Za pomocą metody najwyższego poziomu o nazwie `Hash`.

Wymienione techniki tworzenia tablicy asocjacyjnej podano w stopniu, w jakim jest to możliwe, w kolejności od najpowszechniejszych do używanych najrzadziej. Inaczej mówiąc, omówienie zaczniemy od najczęściej używanej techniki i będziemy je kontynuować dla kolejnych sposobów.

### TWORZENIE TABLICY ASOCJACYJNEJ LITERAŁU

Gdy w nawiasach klamrowych wprowadzasz tablicę asocjacyjną literału, klucze od wartości oddzielasz za pomocą operatora => (chyba że używana jest specjalna składnia `{ key: value }` dla kluczy symboli). Po każdej kompletnej parze klucz i wartość umieszczasz przecinek (z wyjątkiem ostatniej pary, dla której jest to opcjonalne).

Konstruktor tablicy asocjacyjnej literału jest wygodnym rozwiązaniem, gdy istnieją mające trafić do takiej tablicy wartości, które się nie zmienią. Możesz jednokrotnie wpisać te wartości do pliku programu i odwoływać się do nich z poziomu programu. Dobrym przykładem są skróty województw.

Konstruktora tablicy asocjacyjnej literału możesz użyć do utworzenia pustej tablicy asocjacyjnej:

```
h = {}
```

Przypuszczalnie w pewnym momencie postanowisz dodać elementy do pustej tablicy asocjacyjnej. Technikom umożliwiającym to poświęcono następny punkt 9.3.2.

Drugi sposób tworzenia tablicy asocjacyjnej bazuje na tradycyjnym konstruktorze `new`.

### KONSTRUKTOR HASH.NEW

Konstruktor `Hash.new` tworzy pustą tablicę asocjacyjną. Jeśli jednak przekażesz mu argument, będzie on traktowany jak wartość domyślna nieistniejących kluczy tablicy asocjacyjnej. Do wartości domyślnych oraz niektórych dodatków konstruktora `Hash.new` powrócimy po omówieniu operacji wstawiania i pobierania kluczy/wartości.

### METODA KLASY HASH.[ ]

Trzeci sposób tworzenia tablicy asocjacyjnej uwzględnia inną metodę klasy `Hash`, czyli metodę `[]` (nawiasy kwadratowe). Pobiera ona listę elementów rozdzielonych przecinkami, przy założeniu, że istnieje parzysta liczba argumentów. Metoda traktuje elementy jako występujące na przemian klucze i wartości, których używa do tworzenia tablicy asocjacyjnej. Dzięki „lukrowi" składniowemu języka Ruby argumenty możesz umieścić w metodzie `[]` bezpośrednio wewnątrz nawiasów, z których możesz zrezygnować w przypadku użycia znaku kropki wywołania metody:

```
>> Hash["Wrocław", "WR", "Kraków", "KR" ]
=> {"Wrocław"=>"WR", "Kraków"=>"KR"}
```

Jeśli podasz nieparzystą liczbę argumentów, zostanie zgłoszony błąd krytyczny, ponieważ taka liczba nie może być odwzorowywana na serię par klucz i wartość. Możesz jednak przekazać tablicę tablic, w której każda podtablica składa się z dwóch elementów. Metoda `Hash.[ ]` użyje tablic wewnętrznych jako par klucz i wartość:

```
>> Hash[ [[1,2], [3,4], [5,6]] ]
=> {1=>2, 3=>4, 5=>6}
```

Istnieje też możliwość przekazania czegokolwiek, co zawiera metodę o nazwie `to_hash`. Nowa tablica asocjacyjna będzie wynikiem wywołania tej metody.

Inna technika tworzenia tablicy asocjacyjnej dotyczy metody najwyższego poziomu `Hash`.

#### METODA HASH

Metoda `Hash` wyróżnia się trochę specyficznym działaniem. Jeśli zostanie wywołana z pustą tablicą (`[]`) lub wartością `nil`, zwróci pustą tablicę asocjacyjną. W przeciwnym razie metoda wywołuje metodę `to_hash` dla swojego pojedynczego argumentu. Jeżeli argument nie zawiera metody `to_hash`, zgłaszany jest błąd krytyczny (`TypeError`).

Zaprezentowano już kilka sposobów tworzenia tablic asocjacyjnych. Pamiętaj, że wymieniono je w przybliżeniu w kolejności od najbardziej do najmniej popularnych. W reszcie prezentowanych technik znajdziesz znacznie więcej konstruktorów tablicy asocjacyjnej literału i wywołań metody `Hash.new`. Niemniej jednak warto wiedzieć, co jest dostępne, a także jak działają różne techniki.

Przejdźmy teraz do kwestii związanej z modyfikowaniem zawartości tablicy asocjacyjnej. W ramach omówienia podążymy taką samą „ścieżką" jak w przypadku tablic, zajmując się operacjami wstawiania i pobierania, łączeniem tablic asocjacyjnych z innymi tego typy tablicami, transformacjami tablic asocjacyjnych i ich odpytywaniem. Niezależnie od tego przyjrzymy się również ustawianiu wartości domyślnych dla nieistniejących kluczy tablicy asocjacyjnej.

### 9.3.2. Wstawianie, pobieranie i usuwanie par tablic asocjacyjnych

Jak się okaże, tablice asocjacyjne mają wiele wspólnego z tablicami, gdy pod uwagę weźmie się operacje pobierania i wstawiania, choć występuje kilka ważnych różnic i technik specyficznych dla tablic i tablic asocjacyjnych.

#### DODAWANIE DO TABLICY ASOCJACYJNEJ PARY KLUCZ I WARTOŚĆ

Aby do tablicy asocjacyjnej dodać parę klucz i wartość, należy użyć w zasadzie tej samej techniki co w przypadku dodawania elementu do tablicy. Mowa o metodzie `[]=` oraz „lukrze" składniowym.

W celu dodania województwa do tablicy asocjacyjnej `state_hash` zastosuj następujący wiersz:

```
state_hash["Mazowieckie"] = "MZ"
```

Jest to wersja następującego wiersza, dla którego użyto „lukru" składniowego:

```
state_hash.[]=("Mazowieckie", "MZ")
```

Możliwe jest także zastosowanie dla tej operacji równoznacznej metody `store`, która pobiera dwa argumenty (klucz i wartość):

```
state_hash.store("Mazowieckie", "MZ")
```

Dodając element do tablicy asocjacyjnej, pamiętaj o ważnej zasadzie, która głosi, że klucze są unikatowe. Z danym kluczem może istnieć tylko jedna pozycja. Wartości

tablicy asocjacyjnej nie muszą być unikatowe. Ta sama wartość może zostać przypisana do co najmniej dwóch kluczy. Nie mogą jednak istnieć zduplikowane klucze.

Jeśli para klucz i wartość zostanie dodana do tablicy asocjacyjnej, która zawiera już pozycję dla dodawanego klucza, stara pozycja zostanie nadpisana. Oto przykład:

```
h = Hash.new
h["a"] = 1
h["a"] = 2
puts h["a"]  ←——— Dane wyjściowe: 2
```

W tym kodzie dwie wartości są przypisywane kluczowi `"a"` tablicy asocjacyjnej `h`. Drugie przypisanie eliminuje pierwsze, co potwierdza wartość `2` wyświetlana przez instrukcję `puts`.

Jeśli dokonasz ponownego przypisania do danego klucza tablicy asocjacyjnej, zachowa on swoją pozycję w kolejności wstawiania elementów tej tablicy. Modyfikacja wartości tworzącej parę z kluczem nie jest traktowana jak nowa operacja wstawiania do tablicy asocjacyjnej.

### POBIERANIE WARTOŚCI Z TABLICY ASOCJACYJNEJ

W przypadku pobierania wartości z tablicy asocjacyjnej „siłą napędową" jest metoda `[]`. Aby na przykład pobrać wartość `"WR"` z tablicy asocjacyjnej `state_hash` i przypisać ją do zmiennej, użyj następującego wiersza kodu:

```
conn_abbrev = state_hash["Wrocław"]
```

Użycie klucza tablicy asocjacyjnej bardzo przypomina indeksowanie tablicy, z tą różnicą, że indeks (klucz) może być czymkolwiek, natomiast w tablicy jest to zawsze liczba całkowita.

Tablice asocjacyjne oferują też metodę `fetch`, która zapewnia alternatywny sposób pobierania wartości według klucza:

```
conn_abbrev = state_hash.fetch("Wrocław")
```

Metoda `fetch` różni się od metody `[]` sposobem działania w przypadku zażądania od niej wyszukania nieistniejącego klucza: metoda `fetch` zgłasza wyjątek, natomiast metoda `[]` zwraca wartość `nil` lub podaną wartość domyślną (więcej na ten temat w następnym punkcie). Jeśli tablicy asocjacyjnej przekażesz drugi argument, zostanie on zwrócony zamiast wyjątku zgłaszanego, gdy klucz nie zostanie znaleziony. Na przykład dla następującego kodu:

```
state_hash.fetch("Podhale", "Nieznane województwo")
```

zostanie określony łańcuch `"Nieznane województwo"`.

Możliwe jest też w ramach jednej operacji pobranie wartości dla wielu kluczy za pomocą metody `values_at`:

```
two_states = state_hash.values_at("Małopolskie","Wielkopolskie")
```

Kod ten zwraca tablicę w postaci `["MA","WI"]` i przypisuje ją do zmiennej `two_states`.

Gdy już wiesz, jakie są mechanizmy wstawiania informacji do tablicy asocjacyjnej i pobierania ich z niej, zajmijmy się kwestią zapewniania wartości domyślnej (lub domyślnego bloku kodu) podczas tworzenia tablicy asocjacyjnej.

### 9.3.3. Określanie domyślnych wartości i zachowania tablic asocjacyjnych

Domyślnie, gdy zażądasz od tablicy asocjacyjnej wartości odpowiadającej nieistniejącemu kluczowi, uzyskasz wartość `nil`:

```
>> h = Hash.new
=> {}
>> h["brak takiego klucza!"]
=> nil
```

Przekazując argument metodzie `Hash.new`, możesz jednak określić inną wartość domyślną:

```
>> h = Hash.new(0)
=> {}
>> h["brak takiego klucza!"]
=> 0
```

W tym przypadku uzyskiwana jest wartość domyślna `0` tablicy asocjacyjnej, gdy zostanie użyty nieistniejący klucz (możesz również ustawić wartość domyślną dla już istniejącej tablicy asocjacyjnej za pomocą metody `default`).

Ważne jest, aby zapamiętać, że cokolwiek zostanie ustawione jako wartość domyślna, zostanie przekazane po określeniu *nieistniejącego* klucza. Klucz ten zachowa taki status do momentu przypisania mu wartości. Inaczej mówiąc, użycie kodu `h["ple"]` nie oznacza, że tablica asocjacyjna zawiera klucz `"ple"`. Aby ten klucz znalazł się w takiej tablicy, musisz go w niej umieścić. Możesz potwierdzić to, że tablica asocjacyjna `h` nie ma żadnych kluczy, sprawdzając ją po wykonaniu w ostatnim przykładzie operacji wyszukiwania nieistniejącego klucza:

```
>> h
=> {}
```

Aby odwołania do nieistniejących kluczy spowodowały utworzenie ich, metodzie `Hash.new` możesz przekazać blok kodu. Będzie on wykonywany każdorazowo w momencie odwołania do nieistniejącego klucza. Do bloku zostaną dostarczone dwa obiekty: tablica asocjacyjna i (nieistniejący) klucz.

Technika ta stanowi dobry początek w przypadku automatycznego ustawiania kluczy po użyciu ich po raz pierwszy. Nie jest to najbardziej elegancka lub usprawniona technika w języku Ruby, ale działa. Najpierw tworzysz blok kodu, który pobiera tablicę asocjacyjną i klucz, a następnie wykonujesz operację ustawiania.

Aby na przykład każdy nieistniejący klucz został dodany do tablicy asocjacyjnej z wartością `0`, utwórz następującą tablicę:

```
h = Hash.new {|hash,key| hash[key] = 0 }
```

Gdy od tablicy asocjacyjnej `h` zażądasz pobrania wartości dla nieistniejącego w niej klucza, wykonywany jest blok z argumentem `hash` ustawionym na samą tablicę oraz

argumentem `key` ustawionym na nieistniejący klucz. Dzięki kodowi w bloku klucz jest dodawany do tablicy asocjacyjnej z wartością `0`.

Dla tego przypisania nowej tablicy asocjacyjnej do `h` możesz aktywować następujący blok:

```
>> h["nowy klucz!"] ←——— ❶
=> 0
>> h                ←——— ❷
=> {"nowy klucz!"=>0}
```

Gdy spróbujesz wyszukać klucz `"nowy klucz!"` ❶, nie będzie go w tej tablicy, jednak dzięki blokowi klucz ten zostanie dodany z wartością `0`. Przy następnym zażądaniu od narzędzia irb wyświetlenia całej tablicy asocjacyjnej ❷ okaże się, że zawiera ona automatycznie dodaną parę klucz i wartość.

Technika ta ma wiele zastosowań. Umożliwia przyjmowanie, co znajduje się w tablicy asocjacyjnej, nawet jeśli początkowo jest ona pusta. Technika prezentuje też inny aspekt rozbudowanego „repertuaru" dynamicznych technik programistycznych języka Ruby oraz elastyczności tablic asocjacyjnych.

Zajmiemy się teraz sposobami łączenia ze sobą tablic asocjacyjnych, tak jak to miało miejsce w przypadku łańcuchów i tablic.

### 9.3.4. Łączenie tablic asocjacyjnych z innymi tablicami asocjacyjnymi

Proces łączenia dwóch tablic asocjacyjnych w jedną przebiega w dwóch wariantach: destrukcyjnym, w przypadku którego do pierwszej tablicy asocjacyjnej są bezpośrednio dodawane pary klucz i wartość z drugiej tablicy asocjacyjnej, oraz niedestrukcyjnym — polegającym na tym, że tworzona jest nowa, trzecia tablica asocjacyjna łącząca elementy dwóch oryginalnych tablic.

Operacja destrukcyjna jest wykonywana za pomocą metody `update`. Pozycje pierwszej tablicy asocjacyjnej są trwale nadpisywane, jeśli w drugiej tablicy asocjacyjnej znajduje się odpowiadający im klucz:

```
h1 = {"Nowak" => "Jan",
      "Nowacka" => "Joanna" }
h2 = {"Nowak" => "Jerzy" }
h1.update(h2)
puts h1["Nowak"]  ←——— Dane wyjściowe: Jerzy
```

W tym przykładzie wartość pozycji `"Nowak"` tablicy asocjacyjnej `h1` została zmieniona (zaktualizowana) na wartość, jaką pozycja ma w tablicy asocjacyjnej `h2`. Żądasz odświeżenia tablicy asocjacyjnej w celu uwzględnienia zawartości drugiej tablicy asocjacyjnej. Jest to destrukcyjna wersja łączenia tablic asocjacyjnych.

Aby przeprowadzić niedestrukcyjne łączenie dwóch tablic asocjacyjnych, użyj metody `merge`, która zapewnia trzecią tablicę asocjacyjną, pozostawiając niezmienione oryginalne tablice:

```
h1 = {"Nowak" => "Jan",
      "Nowacka" => "Joanna" }
h2 = {"Nowak" => "Jerzy" }
```

```
h3 = h1.merge(h2)
p h1["Nowak"]  ←——— Dane wyjściowe: Jan
```

W tym przypadku para "Nowak"/"Jan" tablicy asocjacyjnej h1 nie jest nadpisywana przez parę "Nowak"/"Jan" tablicy asocjacyjnej h2. Zamiast tego tworzona jest nowa tablica asocjacyjna z parami pochodzącymi z obu oryginalnych tablic. Sprawdzana uzyskana tablica asocjacyjna będzie mieć następującą postać:

```
{"Nowak"=>"Jerzy", "Nowacka"=>"Joanna"}
```

Zauważ, że od tablicy asocjacyjnej h3 wymagane jest podjęcie decyzji: którą z pozycji Nowak powinna zawierać? Odpowiedź jest taka, że gdy dwie scalane tablice asocjacyjne mają wspólny klucz, pierwszeństwo ma druga tablica asocjacyjna (h2 w tym przykładzie). Wartością klucza "Nowak" tablicy asocjacyjnej h3 będzie "Jerzy".

Nawiasem mówiąc, metoda merge! (wersja metody merge z wykrzyknikiem w nazwie) jest synonimem metody update. W celu wykonania tej operacji możesz użyć metody merge! lub update.

Oprócz łączenia z innymi tablicami asocjacyjnymi tego rodzaju tablice mogą też być poddawane transformacji na kilka sposobów, o czym dowiesz się w następnym punkcie.

### 9.3.5. Transformacje tablic asocjacyjnych

W przypadku tablic asocjacyjnych możliwe jest przeprowadzenie kilku transformacji. W tym kontekście *transformacja* oznacza, że dla tablicy asocjacyjnej wywoływana jest metoda, a wynikiem operacji (wartość zwracana metody) jest tablica asocjacyjna. W rozdziale 10. poznasz inne metody wybierania i filtrowania w przypadku tablic asocjacyjnych, które swoje zestawy wynikowe zwracają w tablicach. Przyjrzymy się teraz operacjom między tablicami asocjacyjnymi.

#### WYBIERANIE ELEMENTÓW Z TABLICY ASOCJACYJNEJ I ODRZUCANIE ICH

Aby z istniejącej tablicy asocjacyjnej uzyskać podtablicę asocjacyjną, użyj metody select. Pary klucz i wartość będą przekazywane kolejno do podanego bloku kodu. W wynikowej tablicy asocjacyjnej zostanie uwzględniona każda para, dla której blok zwróci wartość prawda:

```
>> h = Hash[1,2,3,4,5,6]
=> {1=>2, 3=>4, 5=>6}
>> h.select {|k,v| k > 1 }
=> {3=>4, 5=>6}
```

Odrzucanie elementów z tablicy asocjacyjnej przebiega w odwrotny sposób. Oznacza to, że w wynikowej tablicy asocjacyjnej nie znajdą się te pary klucz i wartość, dla których blok zwróci wartość prawda:

```
>> h.reject {|k,v| k > 1 }
=> {1=>2}
```

Metody select i reject mają swoje odpowiedniki w postaci metod z wykrzyknikiem w nazwie (wersje trwale zmieniające oryginalną tablicę asocjacyjną, ale niezwracające nowej tego rodzaju tablicy): select! i reject!. Te dwie metody zwracają wartość nil,

jeśli tablica asocjacyjna nie zostanie zmieniona. W celu wykonania operacji wewnętrznej zwracającej oryginalną tablicę asocjacyjną (nawet jeśli jest niezmieniona) możesz użyć metod `keep_if` i `delete_if`.

#### ODWRACANIE TABLICY ASOCJACYJNEJ

Metoda `Hash#invert` odwraca klucze i wartości. Wartości stają się kluczami, a klucze wartościami:

```
>> h = { 1 => "jeden", 2 => "dwa" }
=> {1=>"jeden", 2=>"dwa"}
>> h.invert
=> {"dwa"=>2, "jeden"=>1}
```

Należy zachować ostrożność przy odwracaniu tablic asocjacyjnych. Wynika to stąd, że klucze tych tablic są unikatowe, ale wartości już nie. Gdy przekształcisz w klucze zduplikowane wartości, jedna z par zostanie odrzucona:

```
>> h = { 1 => "jeden", 2 => "większa niż 1", 3 => "większa niż 1" }
=> {1=>"jeden", 2=>"większa niż 1", 3=>"większa niż 1"}
>> h.invert
=> {"jeden"=>1, "większa niż 1"=>3}
```

W przypadku wykonania operacji odwrócenia tylko jedna z wartości `"większa niż 1"` może zostać zachowana jako klucz. Druga wartość zostanie odrzucona. Tablica asocjacyjna powinna być odwracana tylko wtedy, gdy masz pewność, że unikatowe są zarówno wartości, jak i klucze.

#### CZYSZCZENIE TABLICY ASOCJACYJNEJ

Metoda `Hash#clear` czyści tablicę asocjacyjną:

```
>> {1 => "jeden", 2 => "dwa" }.clear
=> {}
```

Jest to operacja wewnętrzna: pusta tablica asocjacyjna jest tą samą tego typu tablicą (obiektem) co tablica asocjacyjna, do której wysyłany jest komunikat `clear`.

#### ZASTĘPOWANIE ZAWARTOŚCI TABLICY ASOCJACYJNEJ

Podobnie jak łańcuchy i tablice, tak i tablice asocjacyjne zawierają metodę `replace`:

```
>> { 1 => "jeden", 2 => "dwa" }.replace({ 10 => "dziesięć", 20 => "dwadzieścia"})
=> {10 => "dziesięć", 20 => "dwadzieścia"}
```

To również jest operacja wewnętrzna, co sugeruje nazwa metody `replace`.

W dalszej kolejności zajmiemy się metodami odpytującymi tablice asocjacyjne.

### *9.3.6. Odpytywanie tablic asocjacyjnych*

Podobnie jak tablice (i wiele innych obiektów języka Ruby) tablice asocjacyjne zapewniają kilka metod, za pomocą których możesz odpytywać obiekt o jego stan. W tabeli 9.2 zebrano niektóre typowe metody odpytujące tablice asocjacyjne.

Żadna z metod w tablicy 9.2 nie powinna w tym miejscu sprawiać jakichkolwiek niespodzianek. Metody te mają podobną ideę działania, a w niektórych sytuacjach są

**Tabela 9.2.** Typowe metody odpytujące tablice asocjacyjne oraz ich znaczenie

| Nazwa metody/ przykładowe wywołanie | Znaczenie |
|---|---|
| `h.has_key?(1)` | Wartość prawda, jeśli tablica asocjacyjna `h` zawiera klucz `1`. |
| `h.include?(1)` | Synonim metody `has_key?`. |
| `h.key?(1)` | Synonim metody `has_key?`. |
| `h.member?(1)` | Synonim metody `has_key?`. |
| `h.has_value?("trzy")` | Wartość prawda, jeśli dowolna wartość w tablicy `h` to `"trzy"`. |
| `h.value?("trzy")` | Synonim metody `has_value?`. |
| `h.empty?` | Wartość prawda, jeśli tablica asocjacyjna `h` nie zawiera żadnych par klucz i wartość. |
| `h.size` | Liczba par klucz i wartość w tablicy asocjacyjnej `h`. |

dokładnie takie same jak w przypadku tablic. Z wyjątkiem metody `size` wszystkie metody zwracają wartość prawda lub fałsz. Jedynym zaskoczeniem może być to, ile spośród tych metod jest synonimami. Następujące cztery metody sprawdzają, czy istnieje konkretny klucz: `has_key?`, `include?`, `key?` i `member?`. Możliwe jest stwierdzenie, że są to dwa, a nawet trzy synonimy za dużo. Metoda `has_key?` wydaje się być najpopularniejszą z tych czterech metod, a ponadto najbardziej właściwą w odniesieniu do tego, co metoda testuje.

Metoda `has_value?` ma jeden synonim: `value?`. Podobnie jak jej odpowiednik, czyli metoda `has_key?`, metoda `has_value?` wydaje się być bardziej popularna.

Inne metody, takie jak `empty?` i `size`, pozwalają odpowiednio stwierdzić, czy tablica asocjacyjna jest pusta oraz jaka jest jej wielkość (metoda `size` może też zostać wywołana jako metoda `length`). Wielkość tablicy asocjacyjnej jest liczbą znajdujących się w niej par klucz i wartość.

Jak się okaże w następnym punkcie, tablice asocjacyjne uzyskują specjalne miejsce na listach argumentów metod.

### 9.3.7. Tablice asocjacyjne jako ostatnie argumenty metody

Jeśli wywołujesz metodę w taki sposób, że *ostatni* argument na liście argumentów jest tablicą asocjacyjną, w języku Ruby możliwe jest utworzenie tablicy asocjacyjnej bez nawiasów klamrowych. Ta być może trywialnie brzmiąca reguła specjalna w praktyce sprawia, że listy argumentów prezentują się znacznie lepiej niż w sytuacji, gdyby reguły tej nie zastosowano.

Poniżej zamieszczono przykład. Pierwszy argument metody `add_to_city_database` to nazwa miasta. Drugim argumentem jest tablica asocjacyjna danych o mieście zdefiniowana bez nawiasów klamrowych (oraz z użyciem specjalnej notacji symboli *`klucz: wartość`*):

```
add_to_city_database("Nowy Jork",
  state: "Nowy Jork",
  population: 7000000,
  nickname: "Wielkie Jabłko")
```

Metoda `add_to_city_database` musi wykonać więcej dodatkowych działań, aby uzyskać dostęp do przekazywanych jej danych, niż w sytuacji, gdyby wiązała parametry z argumentami w kolejności od lewej do prawej strony listy:

```
def add_to_city_database(name, info)
  c = City.new
  c.name = name
  c.state = info[:state]
  c.population = info[:population]
  # itp.
```

**Tablice asocjacyjne jako pierwsze argumenty**

Oprócz specjalnej składni dostępnej w przypadku stosowania tablic asocjacyjnych w roli ostatnich argumentów metody bez nawiasów klamrowych warto też poznać pułapkę związaną z użyciem tablicy asocjacyjnej jako pierwszego argumentu metody. W tym przypadku reguła jest taka, że musisz nie tylko ująć tablicę asocjacyjną w nawiasy klamrowe, ale też umieścić całą listę argumentów w nawiasach okrągłych. W przeciwnym razie interpreter języka Ruby uzna, że tablica asocjacyjna jest blokiem kodu. Inaczej mówiąc, gdy użyjesz następującego kodu:

```
my_method { "NJ" => "Nowy Jork" }, 100, "inny argument"
```

wyrażenie w nawiasach zostanie w języku Ruby zinterpretowane jako blok. Aby w takiej sytuacji wysłać tablicę asocjacyjną jako argument, musisz zastosować nawiasy okrągłe dla całej listy argumentów w celu wyraźnego wskazania, że nawiasy klamrowe są powiązane z tablicą asocjacyjną, a nie z blokiem.

Oczywiście dokładny proces związany z usuwaniem opakowania tablicy asocjacyjnej będzie się zmieniać dla poszczególnych przypadków (być może obiekty `City` przechowują swoje informacje jako tablicę asocjacyjną, co ułatwiłoby trochę zadanie metodzie). Czy jednak w ten lub inny sposób metoda musi obsługiwać tablicę asocjacyjną?

Miej świadomość tego, że choć musisz pominąć nawiasy klamrowe, gdy literał tablicy asocjacyjnej jest ostatnią pozycją na liście argumentów, w każdym przypadku możesz użyć dowolnej żądanej liczby tablic asocjacyjnych jako argumentów metody. Pamiętaj jedynie, że pominięcie nawiasów jest możliwe tylko wtedy, gdy tablica asocjacyjna jest ostatnim argumentem.

Do czasu pojawienia się języka Ruby 2 tego rodzaju argumenty w postaci tablicy asocjacyjnej miały najwięcej wspólnego z argumentami nazwanymi lub argumentami słów kluczowych. Uległo to jednak całkowitej zmianie. Obecnie w języku Ruby występują prawdziwe argumenty nazwane. Ich składnia w dużym stopniu przypomina tablice asocjacyjne. Z tego właśnie powodu zajmujemy się nimi tutaj, a nie w rozdziale 2.

### *9.3.8. Powrót do składni argumentów: argumenty nazwane (słów kluczowych)*

Użycie argumentów nazwanych pozwala uniknąć kłopotu związanego z „usuwaniem opakowania" tablic asocjacyjnych w stosowanych metodach. Oto podstawowe przykłady prezentujące najprostszą wersję argumentów nazwanych:

```
>> def m(a:, b:)
>>   p a,b
>> end
=> :m
>> m(a: 1, b: 2)
1
2
=> [1, 2]
```

Po stronie metody występują dwa parametry zakończone dwukropkami. Po stronie wywołania znajduje się coś, co bardzo przypomina tablicę asocjacyjną. W języku Ruby wszystko jest tak dopasowywane, że wartości argumentów `a` i `b` są wiązane zgodnie z oczekiwaniami. Nie ma potrzeby wnikania w tablicę asocjacyjną.

W poprzednim przykładzie `a` i `b` reprezentują wymagane argumenty słów kluczowych. Nie możesz wywołać metody bez nich:

```
>> m
ArgumentError: missing keywords: a, b
>> m(a: 1)
ArgumentError: missing keyword: b
```

Podając wartości domyślne dla parametrów nazwanych, możesz zapewnić opcjonalność argumentów słów kluczowych. Dzięki temu lista parametrów w jeszcze większym stopniu przypomina tablicę asocjacyjną:

```
>> def m(a: 1, b: 2)
>>   p a,b
>> end
=> :m
>> m          ←❶
1
2
=> [1, 2]
>> m(a:10)    ←❷
10
2
=> [10, 2]
```

W przypadku wywołania metody bez żadnych argumentów ❶ stosowane są wartości domyślne dla `a` i `b`. Jeśli podasz argument `a`, lecz nie `b` ❷, pierwszy z nich będzie mieć określoną wartości, a drugi wartość domyślną.

Co będzie, gdy obierzesz inny kierunek, wywołując metodę przy użyciu argumentów słów kluczowych, które nie są deklarowane przez metodę? Jeśli lista parametrów metody zawiera nazwę z dwoma gwiazdkami, zmienna o takiej nazwie pobierze do tablicy asocjacyjnej wszystkie nieznane argumenty słów kluczowych w następujący sposób:

```
>> def m(a: 1, b: 2, **c)
>>   p a,b,c
>> end
=> :m
>> m(x: 1, y: 2)
1
```

```
2
{:x=>1, :y=>2}
=> [1, 2, {:x=>1, :y=>2}]
```

W przypadku braku parametru pobierającego słowa kluczowe wywołanie metody, takie jak `m(x:1, y:2)`, po prostu przekazuje tablicę asocjacyjną, co może zakończyć się powodzeniem lub nie, zależnie od tego, jakie są oczekiwane argumenty.

Oczywiście możliwe jest połączenie argumentów słów kluczowych i argumentów innego typu:

```
>> def m(x, y, *z, a: 1, b:, **c, &block)
>>   p x,y,z,a,b,c
>> end
=> :m
>> m(1,2,3,4,5,b:10,p:20,q:30)
1
2
[3, 4, 5]
1
10
{:p=>20, :q=>30}
=> [1, 2, [3, 4, 5], 1, 10, {:p=>20, :q=>30}]
```

W tym przypadku metoda `m`:

- pobiera dwa wymagane argumenty pozycyjne (`x` i `y` powiązane z wartościami `1` i `2`);
- zawiera parametr „pobierania" (`z`), który zajmuje się dodatkowymi argumentami znajdującymi się za argumentami pozycyjnymi (`3`, `4`, `5`);
- zawiera po jednym opcjonalnym i wymaganym argumencie słów kluczowych (`a` i `b` powiązane odpowiednio z wartościami `1` i `10`);
- zawiera słowo kluczowe „pobierania" (`c`) w celu pobierania nieznanych argumentów nazwanych (tablica asocjacyjna argumentów `p` i `h`);
- zawiera zmienną do powiązania z blokiem kodu, jeśli istnieje (`block`).

Rzadko spotkasz się z sygnaturami metod o takiej złożoności, jeśli więc jesteś w stanie poradzić sobie z elementami powyższej metody, raczej poradzisz sobie z innymi przypadkami.

W dalszej kolejności przyjrzymy się zakresom, które zapewne nie są w pełni obiektami kolekcji, ale jak się okazuje, mają wiele z nimi wspólnego.

## 9.4. Zakresy

*Zakres* jest obiektem z punktem początkowym i końcowym. Semantyka operacji na zakresach obejmuje następujące dwa podstawowe pojęcia:

- *Włączenie*. Czy dana wartość znajduje się wewnątrz zakresu?
- *Wyliczenie*. Zakres jest traktowany jako umożliwiająca przechodzenie kolekcja pojedynczych elementów.

Logika włączenia dotyczy wszystkich zakresów. Zawsze możesz przeprowadzić test włączenia. Logika wyliczenia obowiązuje tylko określone zakresy, a mianowicie te,

które obejmują skończoną liczbę dyskretnych wartości możliwych do zidentyfikowania. Nie możesz dokonywać iteracji zakresu wyznaczonego przez dwie liczby zmiennoprzecinkowe, ponieważ taki zakres składa się z nieskończonej liczby wartości. Możliwa jest jednak iteracja zakresu między dwoma liczbami całkowitymi.

Dalszą analizę iteracji zakresu i logiki wyliczenia odłożymy do następnego rozdziału, w którym dokładnie przyjrzymy się wyliczeniu i modułowi `Enumerable`. W tym podrozdziale zajmiemy się przede wszystkim innym pojęciem semantycznym, czyli logiką włączenia. Zaczniemy od wybranych technik tworzenia zakresu.

### 9.4.1. Tworzenie zakresu

Obiekty zakresu możesz utworzyć za pomocą metody `Range.new`. Jeśli operację tę wykonasz w narzędziu irb, zostanie zaprezentowana składnia konstrukcji zakresu literału:

```
>> r = Range.new(1,100)
=> 1..100
```

Oczywiście do utworzenia zakresu może też być bezpośrednio użyta składnia literału:

```
>> r = 1..100
=> 1..100
```

Gdy widzisz zakres z dwoma kropkami między wartością początkową i końcową, tak jak w poprzednim przykładzie, masz do czynienia z zakresem *włącznym*. Zakres z trzema kropkami w środku to zakres *wykluczający*:

```
>> r = 1...100
=> 1...100
```

Różnica między tymi zakresami tkwi w tym, czy punkt końcowy jest uważany za należący do zakresu, czy nie. Powracając do punktu wyjścia, możesz też określić wariant włączny lub wykluczający podczas tworzenia zakresu za pomocą metody `Range.new`: domyślnie jest używany zakres włączny, ale możesz wymusić zakres wykluczający przez przekazanie konstruktorowi trzeciego argumentu `true`:

```
>> Range.new(1,100)
=> 1..100
>> Range.new(1,100,true)
=> 1...100
```

Niestety, trzeba samemu zapamiętać, jaki wariant zakresu ma być domyślny, a ponadto jaki jest wyzwalany przez argument `true`.

Na ogół trudno też zapamiętać, jaka liczba kropek powiązana jest z jakim typem zakresu.

#### ZAPAMIĘTYWANIE SPOSOBU STOSOWANIA DWÓCH I TRZECH ZNAKÓW KROPKI

Jeśli śledzisz fora dyskusyjne poświęcone językowi Ruby, to co jakiś czas napotkasz wiadomości i wpisy osób, które mają trudność z zapamiętaniem, co jest czym: mowa o rozróżnianiu dwóch i trzech kropek lub zakresu włącznego i wykluczającego.

Jednym ze sposobów na zapamiętanie tego jest pomyślenie o zakresie jako o zawsze osiągającym punkt reprezentowany przez cokolwiek, co następuje po drugiej kropce. W zakresie włącznym punkt po drugiej kropce jest wartością końcową zakresu. W następującym przykładzie wartość `100` wchodzi w skład zakresu:

```
1..100
```

Jednak w przypadku następującego zakresu wykluczającego wartość `100` znajduje się poza faktycznym końcem zakresu:

```
1...100
```

Inaczej mówiąc, wartość `100` możesz potraktować jak „przeciągniętą" w prawą stronę w taki sposób, że jest ona poza zakresem.

Przejdziemy teraz do logiki włączenia do zakresów, czyli do punktu ściśle odpowiadającego punktom dotyczącym odpytywania, które wchodzą w skład omówienia łańcuchów, tablic i tablic asocjacyjnych. Wynika to stąd, że większość działań związanych z zakresami obejmuje odpytywanie ich z wykorzystaniem kryteriów włączenia.

### 9.4.2. Logika włączenia do zakresów

Zakresy oferują metody `begin` i `end`, które informują o punktach początkowych i końcowych zakresów:

```
>> r = 1..10
=> 1..10
>> r.begin
=> 1
>> r.end
=> 10
```

Zakres dysponuje też informacją o tym, czy jest zakresem wykluczającym (z trzema kropkami):

```
>> r.exclude_end?
=> false
```

Po określeniu punktów granicznych zakresu możesz rozpocząć test dotyczący włączenia.

Do testowania włączenia wartości do zakresu udostępniono dwie metody: `cover?` i `include?` (istnieje też jej alias `member?`).

#### TESTOWANIE WŁĄCZENIA DO ZAKRESU ZA POMOCĄ METODY COVER?

Metoda `cover?` przeprowadza prosty test: jeśli argument metody jest większy niż punkt początkowy zakresu, a mniejszy niż jego punkt końcowy (lub jemu równy w przypadku zakresu włącznego), uznaje się, że zakres *obejmuje* obiekt. Testy są wykonywane z wykorzystaniem boolowskich testów porównawczych. Wynik fałszywy występuje w sytuacjach, w których porównanie nie ma sensu.

Wszystkie następujące porównania mają sens (jedno z nich kończy się niepowodzeniem, ponieważ element nie należy do zakresu):

```
>> r = "a".."z"
=> "a".."z"
>> r.cover?("a")    ←——— true: "a" >= "a" i "a" <= "z"
=> true
>> r.cover?("abc") ←——— true: "abc" >= "a" i "abc" <= "z"
=> true
>> r.cover?("A")    ←——— false: "A" < "a"
=> false
```

Następujący test kończy się jednak niepowodzeniem, ponieważ element sprawdzany pod kątem włączenia nie może zostać porównany z punktem początkowym i końcowym zakresu:

```
>> r.cover?([])
=> false
```

Bezsensowne jest pytanie o to, czy tablica jest większa niż łańcuch `"a"`. Jeśli samemu spróbujesz dokonać takiego porównania, uzyskasz błąd krytyczny. Na szczęście zakresy bazują na bardziej konserwatywnym sposobie działania, dlatego przekażą informację o tym, że nie obejmują elementu.

Metoda `cover?` przeprowadza porównania z użyciem punktu początkowego i końcowego zakresu, natomiast inna metoda testująca włączenie, o nazwie `include?` (lub `member?`), korzysta z rozwiązania opartego w większym stopniu na kolekcjach.

### TESTOWANIE WŁĄCZENIA DO ZAKRESU ZA POMOCĄ METODY INCLUDE?

Test wykonywany przez metodę `include?` traktuje zakres jak coś w rodzaju kryptotablicy, czyli kolekcji wartości. Na przykład zakres `"a".."z"` jest uważany za uwzględniający (zgodnie z ustaleniami metody `include?`) tylko 26 wartości, które zawierają się między łańcuchami `"a"` i `"z"`.

A zatem metoda `include?` generuje wyniki różniące się od zwracanych przez metodę `cover?`:

```
>> r.include?("a")
=> true
>> r.include?("abc")
=> false
```

W sytuacjach, gdy zakres nie może być interpretowany jako ograniczona kolekcja, tak jak zakres liczb zmiennoprzecinkowych, metoda `include?` ucieka się do uporządkowania i porównywania liczbowego:

```
>> r = 1.0..2.0
=> 1.0..2.0
>> r.include?(1.5)
=> true
```

Jak obiecano, w następnym rozdziale dowiesz się więcej na temat quasi-kolekcji. W tym rozdziale pozostała do omówienia jeszcze jedna podstawowa klasa kolekcji, czyli klasa `Set`.

**Czy istnieją zakresy wsteczne?**

Nieprowadząca do rozstrzygnięcia odpowiedź na pytanie dotyczące zakresów wstecznych brzmi: tak i nie. Choć możesz utworzyć zakres wsteczny, nie zapewni tego, czego prawdopodobnie oczekujesz:

```
>> r = 100...1
=> 100...1
>> r.include?(50)
=> false
```

Zakres nie stwarza problemów z przeprowadzeniem zwykłego testu włączenia. W ramach testu określane jest, czy kandydat do włączenia jest większy niż punkt początkowy zakresu i mniejszy od jego punktu końcowego. Ponieważ 50 nie jest ani większe niż 100, ani mniejsze od 1, test kończy się niepowodzeniem w trybie cichym. Jest to błąd logiczny, a nie błąd krytyczny lub błąd czasu wykonywania.

Zakresy wsteczne pojawiają się w jednym konkretnym zbiorze przypadków użycia: jako argumenty indeksu łańcuchów i tablic. Zwykle zakresy te przyjmują postać z dodatnim punktem początkowym i ujemnym punktem końcowym, który znajduje się po prawej stronie zakresu:

```
>> "To jest przykladowy lancuch"[10..-5]
=> "zykładowy łań"
>> ['a','b','c','d'][0..-2]
=> ["a", "b", "c"]
```

Możliwe jest nawet użycie wykluczającego zakresu wstecznego:

```
>> ['a','b','c','d'][0...-2]
=> ["a", "b"]
```

W tych przypadkach to, co nie działa (a przynajmniej w sposób, jakiego mogłeś oczekiwać) w samym zakresie, sprawdza się po zastosowaniu dla łańcucha lub tablicy.

## 9.5. Zbiory

`Set` to jedyna z klas omawianych w tym rozdziale, która — mówiąc wprost — nie jest podstawową klasą języka Ruby. Jest to klasa biblioteki standardowej. Oznacza to, że w celu jej użycia konieczne jest zastosowanie następującego wiersza kodu:

```
require 'set'
```

Zgodnie z ogólną regułą obowiązującą w książce zajmujemy się podstawowymi elementami języka, a nie biblioteką standardową. Klasa `Set` zasługuje jednak na potraktowanie jej wyjątkowo, ponieważ bardzo dobrze wpasowuje się w inne analizowane klasy kontenerów i kolekcji.

*Zbiór* to unikatowa kolekcja obiektów. Obiekty mogą być dowolne — łańcuchy, liczby całkowite, tablice, inne zbiory. Jednak żaden obiekt nie może wystąpić w zbiorze więcej niż raz. Unikalność jest też wymuszana zdroworozsądkowo na poziomie zawartości: jeśli zbiór zawiera łańcuch `"Nowy Jork"`, do zbioru nie możesz dodać takiego samego łańcucha nawet pomimo tego, że technicznie dwa łańcuchy mogą być różnymi obiektami. To samo dotyczy tablic z jednakową zawartością.

**UWAGA** Wewnętrznie zbiory korzystają z tablicy asocjacyjnej w celu wymuszenia unikalności swojej zawartości. Gdy element jest dodawany do zbioru,

wewnętrzna tablica asocjacyjna tego zbioru uzyskuje nowy klucz. A zatem dowolne dwa obiekty, które zostałyby uznane za duplikaty w przypadku użycia w roli kluczy tablicy asocjacyjnej, nie mogą występować razem w zbiorze.

Przyjrzyjmy się teraz sposobowi tworzenia zbiorów.

### 9.5.1. Tworzenie zbiorów

Aby utworzyć zbiór, użyj konstruktora `Set.new`. Możesz utworzyć pusty zbiór lub przekazać obiekt kolekcji (zdefiniowany jako obiekt odpowiadający metodzie `each` lub `each_entry`). W drugim przypadku wszystkie elementy kolekcji są umieszczane pojedynczo w zbiorze:

```
>> new_england = ["Connecticut", "Maine", "Massachusetts",
                  "New Hampshire", "Rhode Island", "Vermont"]
=> ["Connecticut", "Maine", "Massachusetts",
    "New Hampshire","Rhode Island", "Vermont"]
>> state_set = Set.new(new_england)
=> #<Set: {"Connecticut", "Maine", "Massachusetts",
           "New Hampshire", "Rhode Island", "Vermont"}>
```

W tym przypadku utworzono tablicę `new_england` i użyto jej jako argumentu konstruktora w celu utworzenia zbioru `state_set`. Zauważ, że nie występuje konstruktor zbioru literału (brak odpowiednika `[]` lub `{}` odpowiednio dla tablic lub tablic asocjacyjnych). Wynika to z tego, że zbiory są częścią biblioteki standardowej, a nie rdzenia języka Ruby, a ponadto jego składnia jest już dostępna przed załadowaniem biblioteki zbioru.

Możliwe jest również zapewnienie bloku kodu konstruktorowi. W takim przypadku każdy podany element w obiekcie kolekcji jest przekazywany za pośrednictwem bloku (dostarczany do niego) z wartością wynikową wstawianą do zbioru. Oto na przykład kod ilustrujący sposób inicjowania zbioru w celu uzyskania listy łańcuchów złożonych z dużych liter:

```
>> names = ["Dawid", "Karol", "Cezary", "Anna"]
=> ["Dawid", "Karol", "Cezary", "Anna"]
>> name_set = Set.new(names) {|name| name.upcase }
=> #<Set: {"ANNA", "KAROL", "CEZARY", "DAWID"}>
```

Zamiast używać tablicy nazw jako swoich wartości początkowych, konstruktor zbioru dostarcza każdą nazwę do bloku i wstawia do zbioru to, co uzyskuje w odpowiedzi (czyli wersję łańcucha złożoną z dużych liter).

Gdy już dysponujemy zbiorem, możemy go zmodyfikować.

### 9.5.2. Modyfikowanie elementów zbioru

Podobnie jak tablice zbiory mają dwa tryby dodawania elementów: wstawianie nowego elementu do zbioru lub wykorzystanie innego obiektu kolekcji jako źródła dla wielu nowych elementów. W świecie tablic jest to różnica między metodami `<<` i `concat`. W przypadku zbiorów rozróżnienie to jest odzwierciedlane w różnych metodach, którym tutaj się przyjrzymy.

### DODAWANIE OBIEKTU DO ZBIORU I USUWANIE GO ZE ZBIORU

Aby do zbioru dodać pojedynczy obiekt, możesz użyć operatora/metody <<:

```
>> tri_state = Set.new(["New Jersey", "Nowy Jork"])
=> #<Set: {"New Jersey", "Nowy Jork"}>          ←— Oj, tylko dwa!
>> tri_state << "Connecticut"                   ←— Dodaje trzeci
=> #<Set: {"New Jersey", "Nowy Jork", "Connecticut"}>
```

Jak w przypadku tablic, łańcuchów i innych obiektów w tym kodzie metoda << kojarzy się z dołączaniem do kolekcji lub obiektu zmiennego. Jeśli spróbujesz dodać obiekt już znajdujący się w zbiorze (lub obiekt o zawartości odpowiadającej zawartości obiektu w zbiorze), nic się nie wydarzy:

```
>> tri_state << "Connecticut"                   ←— Drugi raz
=> #<Set: {"New Jersey", "Nowy Jork", "Connecticut"}>
```

W celu usunięcia obiektu użyj metody `delete`:

```
>> tri_state << "Pennsylwania"
=> #<Set: {"New Jersey", "Nowy Jork", "Connecticut", "Pennsylwania"}>
>> tri_state.delete("Connecticut")
=> #<Set: {"New Jersey", "Nowy Jork", "Pennsylwania"}>
```

Usuwanie obiektu, którego nie ma w zbiorze, nie powoduje zgłoszenia błędu. Tak jak w przypadku zduplikowanego obiektu nic się nie wydarzy.

Metoda << jest też dostępna jako metoda `add`. Istnieje również metoda o nazwie `add?`, która różni się od metody `add` tym, że zwraca wartość `nil` (zamiast samego zbioru), jeśli po wykonaniu operacji zbiór pozostaje niezmieniony:

```
>> tri_state.add?("Pennsylwania")
=> nil
```

Możliwe jest sprawdzenie wartości zwracanej metody `add?` w celu określenia, czy ma zostać użyta inna gałąź warunkowa, jeśli okaże się, że element, który próbowano dodać, już istniał.

### ILOCZYN, SUMA I RÓŻNICA ZBIORÓW

W przypadku zbiorów występuje pojęcie iloczynu, sumy i różnicy określanych z użyciem innych zbiorów, a także, jak się okazuje, innych obiektów wyliczanych. Klasa `Set` oferuje metody niezbędne do wykonania tych operacji.

Metody mają anglojęzyczne nazwy i aliasy symboliczne. Są one następujące:

- `intersection` z aliasem `&`,
- `union` z aliasami `+` i `|`,
- `difference` z aliasem `-`.

Każda z tych metod zwraca nowy zbiór złożony z oryginalnego zbioru poszerzonego lub pomniejszonego o odpowiednie elementy z obiektu zapewnionego jako argument metody. Oryginalny zbiór pozostaje bez zmian.

Przyjrzyjmy się kilku operacjom na zbiorach na przykładzie trzech zgrupowanych wschodnich stanów Stanów Zjednoczonych:

```
>> tri_state = Set.new(["Connecticut", "New Jersey", "Nowy Jork"])
=> #<Set: {"Connecticut", "New Jersey", "Nowy Jork"}>
# Odejmowanie (difference/-)
>> state_set - tri_state
=> #<Set: {"Maine", "Massachusetts", "New Hampshire", "Rhode Island",
"Vermont"}>
# Dodawanie (union/+/|)
>> state_set + tri_state
=> #<Set: {"Connecticut", "Maine", "Massachusetts", "New Hampshire",
"Rhode Island", "Vermont", "New Jersey", "Nowy Jork"}>
# Iloczyn (&)
>> state_set & tri_state
=> #<Set: {"Connecticut"}>
>> state_set | tri_state
=> #<Set: {"Connecticut", "Maine", "Massachusetts", "New Hampshire",
"Rhode Island", "Vermont", "New Jersey", "Nowy Jork"}>
```

Dostępny jest też operator alternatywy wyłączającej ^ umożliwiający uzyskanie sumy wykluczającej dla zbioru i obiektu wyliczeniowego, czyli zbioru złożonego z wszystkich elementów, które występują w zbiorze lub w obiekcie wyliczeniowym, lecz nie w obu:

```
>> state_set ^ tri_state
=> #<Set: {"New Jersey", "Nowy Jork", "Maine", "Massachusetts",
   "New Hampshire", "Rhode Island", "Vermont"}>
```

Istniejący zbiór możesz rozszerzyć za pomocą techniki, która pod względem efektu bardzo przypomina technikę opartą na metodzie `Set.new`. Mowa o metodzie `merge`, która jako swój argument może pobrać dowolny obiekt odpowiadający metodzie `each` lub `each_entry`. Obejmuje to tablice, tablice asocjacyjne i zakresy, a także oczywiście inne zbiory.

### SCALANIE KOLEKCJI Z KOLEJNYM ZBIOREM

To, co ma miejsce podczas scalania ze zbiorem kolejnego obiektu, zależy od tego, jaki jest przebieg iteracji samego obiektu. Oto przykład z tablicą obejmujący sprawdzenie obiektu `object_id` w celu potwierdzenia, że oryginalny zbiór został wewnętrznie zmodyfikowany:

```
>> tri_state = Set.new(["Connecticut", "New Jersey"])
=> #<Set: {"Connecticut", "New Jersey"}>
>> tri_state.object_id
=> 2703420
>> tri_state.merge(["Nowy Jork"])
=> #<Set: {"Connecticut", "New Jersey", "Nowy Jork"}>
>> tri_state.object_id
=> 2703420
```

Scalanie tablicy asocjacyjnej ze zbiorem powoduje dodanie do niego dwuelementowych tablic złożonych z klucza i wartości. Wynika to z tego, jak tablice asocjacyjne dzielą się podczas wykonywania dla nich iteracji. Oto trochę nierzeczywisty przykład, który demonstruje to rozwiązanie:

```
>> s = Set.new([1,2,3])
=> #<Set: {1, 2, 3}>
>> s.merge({ "New Jersey" => "NJ", "Maine" => "ME" })
=> #<Set: {1, 2, 3, ["New Jersey", "NJ"], ["Maine", "ME"]}>
```

Jeśli w metodzie `Set.new` podasz argument w postaci tablicy asocjacyjnej, zachowanie nie zmieni się: uzyskasz nowy zbiór z dwuelementowymi tablicami, które bazują na tablicy asocjacyjnej.

Możesz zdecydować się na scalenie ze zbiorem tylko kluczy tablicy asocjacyjnej, a nie jej całej. Jak się okazuje, wewnętrznie członkostwo w zbiorze oparte jest na unikalności kluczy tablicy asocjacyjnej. W tym celu możesz użyć metody `keys`:

```
>> state_set = Set.new(["Nowy Jork", "New Jersey"])
=> #<Set: {"Nowy Jork", "New Jersey"}>
>> state_hash = { "Maine" => "ME", "Vermont" => "VT" }
=> {"Maine"=>"ME", "Vermont"=>"VT"}
>> state_set.merge(state_hash.keys)
=> #<Set: {"Nowy Jork", "New Jersey", "Maine", "Vermont"}>
```

Gdy wypróbujesz kilka kombinacji operacji scalania ze zbiorem, zauważysz, że cechuje się ona sporą elastycznością (podobnie jak operacja tworzenia zbiorów), pod warunkiem że argument jest obiektem z metodą `each` lub `each_entry`.

Zbiory nie byłyby zbiorami bez podzbiorów i nadzbiorów. Obiekty zbiorów w języku Ruby obsługują podzbiory i nadzbiory.

### 9.5.3. Podzbiory i nadzbiory

Za pomocą metod mających zgodne z oczekiwaniami nazwy `subset` i `superset` możesz sprawdzać relacje między zbiorami (argumenty muszą być zbiorami, a nie tablicami, tablicami asocjacyjnymi albo dowolnego innego rodzaju obiektami wyliczeniowymi lub kolekcją) dotyczące podzbiorów/nadzbiorów:

```
>> small_states = Set.new(["Connecticut", "Delaware", "Rhode Island"])
=> #<Set: {"Connecticut", "Delaware", "Rhode Island"}>
>> tiny_states = Set.new(["Delaware", "Rhode Island"])
=> #<Set: {"Delaware", "Rhode Island"}>
>> tiny_states.subset?(small_states)
=> true
>> small_states.superset?(tiny_states)
=> true
```

Dostępne są także metody `proper_subset` i `proper_superset`. *Podzbiorem właściwym* jest podzbiór mniejszy niż zbiór nadrzędny przynajmniej o jeden element. Jeśli dwa zbiory są równe, są własnymi podzbiorami, lecz nie podzbiorami właściwymi. *Nadzbiór właściwy* zbioru jest drugim zbiorem, który zawiera wszystkie elementy pierwszego zbioru oraz przynajmniej jeden element nieobecny w pierwszym zbiorze. Pojęcie „właściwy” jest sposobem na wyizolowanie przypadku, w którym zbiór jest nadzbiorem lub podzbiorem samego siebie, ponieważ wszystkie zbiory należą do obu tych kategorii.

W następnym rozdziale będziemy kontynuować zagadnienie zbiorów. Kolejny raz zajmiemy się obiektami kolekcji, aby obszerniej zaprezentować moduł `Enumerable` oraz zapewniane przez niego usługi oparte na kolekcjach.

## 9.6. Podsumowanie

W rozdziale zaprezentowano następujące zagadnienia:

- Sposób tworzenia, modyfikowania i transformacji obiektów kolekcji, w tym:
  - tablic,
  - tablic asocjacyjnych,
  - zakresów,
  - zbiorów.
- Argumenty nazwane.

Przyjrzeliśmy się bliżej podstawowym klasom kontenerów `Array` i `Hash` języka Ruby. Zajęliśmy się też zakresami, które choć głównie pełnią rolę kryteriów testu włączenia, potrafią zachowywać się jak kolekcje, gdy zezwala im na to ich natura (kwestia ta nabierze większego sensu po dokładniejszym zaprezentowaniu modułu `Enumerable`). W dalszej części rozdziału omówiono zbiory, które zdefiniowano w bibliotece standardowej, a ponadto zapewniają kolejne ważne narzędzie w zestawie narzędziowym kolekcji języka Ruby. Kod źródłowy klasy `Set` tworzony jest w języku Ruby. Zapewnia to możliwość przyjrzenia się prawdziwemu kodowi produkcyjnemu Ruby.

Omówiliśmy również argumenty nazwane, które eliminują konieczność używania kluczy tablicy asocjacyjnej jako pseudosłów kluczowych i „rozpakowywania" tablic asocjacyjnych argumentów w metodach.

Pojęcie *kolekcji* w języku Ruby jest blisko powiązane z modułem `Enumerable` oraz jego zasadą zależności od metody `each`. W następnym rozdziale bardziej obszernie zaprezentujemy moduł `Enumerable`. Oznacza to omówienie wielu dostępnych operacji wyszukiwania, filtrowania, sortowania i transformacji wykonywanych na obiektach, których klasy dołączają ten moduł.

# 10 Kolekcje: moduł Enumerable i klasa Enumerator

**W tym rozdziale:**

- Dołączanie modułu `Enumerable` do własnych klas.
- Użycie metod modułu `Enumerable` w obiektach kolekcji.
- Łańcuchy jako obiekty quasi-wyliczeniowe.
- Sortowanie obiektów wyliczeniowych za pomocą modułu `Comparable`.
- Enumeratory.

Obiekty kolekcji nie są tworzone jednakowo, ale naprawdę mnóstwo z nich ma wiele wspólnych cech. W języku Ruby wspólne cechy wielu obiektów zlokalizowane są zwykle w modułach. Kolekcje nie są wyjątkiem: obiekty kolekcji w języku Ruby zazwyczaj uwzględniają moduł `Enumerable`.

Klasy używające tego modułu „zawierają" swego rodzaju umowę: klasa musi zdefiniować metodę instancji o nazwie `each`, a w zamian moduł `Enumerable` zapewnia obiektom klasy wszelkiego rodzaju zachowania powiązane z kolekcjami. Metody stojące za tymi zachowaniami są definiowane na warunkach metody `each`. Pod pewnymi względami możesz stwierdzić, że całe pojęcie „kolekcji" w języku Ruby tkwi w module `Enumerable`, a definiowane przez niego metody bazują na metodzie `each`.

Wcześniej zaprezentowano już w pewnym stopniu sposób działania metody `each`. W tym rozdziale dowiesz się o niej znacznie więcej. Pamiętaj jednak o tym, że choć każda podstawowa klasa kolekcji obejmuje jakąś część modułu `Enumerable`, każda z nich ma też własne metody. Metody tablicy nie są takie same jak metody zbioru. Z kolei

metody zakresu są inne niż metody tablicy asocjacyjnej. Czasami klasy kolekcji mają metody o wspólnych nazwach, ale metody te nie zapewniają takiego samego działania. *Nie mogą* one zawsze realizować tego samego. Rzecz w tym, aby dysponować wieloma klasami kolekcji, ale mieć możliwość wyodrębnienia do jednego modułu jak najwięcej wspólnego zachowania.

Moduł `Enumerable` możesz dołączyć do własnych klas:

```
class C
  include Enumerable
end
```

Samo w sobie niewiele to powoduje. Aby skorzystać z zalet modułu `Enumerable`, we własnej klasie musisz zdefiniować metodę instancji `each`:

```
class C
  include Enumerable
  def each
    # w tym miejscu znajduje się odpowiedni kod
  end
end
```

Na tym etapie obiekty klasy `C` będą mieć możliwość wywołania dowolnej metody instancji zdefiniowanej w module `Enumerable`.

W rozdziale przyjrzymy się nie tylko modułowi `Enumerable`, ale też blisko powiązanej z nim klasie o nazwie `Enumerator`. *Enumeratory* są obiektami zawierającymi informacje o sposobie dokonywania iteracji dla określonej kolekcji. Zapewniając obiektowi niezależnemu od samej kolekcji „inteligencję" związaną z iteracją, enumeratory dodają kolejny, bogaty w możliwości „wymiar" do już znaczących narzędzi języka Ruby służących do modyfikowania kolekcji.

Zacznijmy od dokładniejszego omówienia metody `each` oraz jej roli jako mechanizmu zachowania obiektów wyliczeniowych.

## 10.1. Zapewnianie możliwości wyliczania za pośrednictwem metody each

Dowolna klasa aspirująca do bycia wyliczeniową musi zawierać metodę `each`, której zadaniem jest przekazywanie udostępnionemu blokowi kodu elementów po jednym na raz.

To, co dokładnie metoda `each` realizuje, będzie się zmieniać dla poszczególnych klas. W przypadku tablicy metoda ta przekazuje pierwszy element, a następnie drugi itd. Dla tablicy asocjacyjnej metoda przekazuje pary klucz-wartość w postaci dwuelementowych tablic. W przypadku uchwytu pliku metoda `each` dostarcza po jednym wierszu z pliku. Iteracja zakresów polega na stwierdzeniu najpierw, czy operacja ta jest możliwa (jest tak na przykład, gdy punkt początkowy reprezentuje liczba zmiennoprzecinkowa), po czym pretendują one do bycia tablicą. Jeżeli metoda `each` zostanie zdefiniowana we własnej klasie, jej znaczenie może być dowolne, pod warunkiem że będzie coś przez nią przekazywane. A zatem metoda `each` oferuje odmienną semantykę w przypadku różnych klas. Niezależnie jednak od sposobu implementacji metody `each` metody w module `Enumerable` są zależne od możliwości jej wywołania.

Aby dobrze zorientować się w sposobie działania modułu `Enumerable`, możesz utworzyć niewielką, demonstracyjną klasę, która z niego korzysta. Poniższy listing 10.1 prezentuje taką klasę o nazwie `Rainbow`. Klasa zawiera metodę `each`, która przekazuje po jednym kolorze jednocześnie. Ponieważ klasa dołącza moduł `Enumerable`, jej instancje są automatycznie wyposażane w metody instancji zdefiniowane w tym module.

**Listing 10.1. Klasa Rainbow i implementacja jej metody each**

```
class Rainbow
  include Enumerable
  def each
    yield "czerwony"
    yield "pomarańczowy"
    yield "żółty"
    yield "zielony"
    yield "niebieski"
    yield "indygowy"
    yield "fioletowy"
  end
end
```

Każda instancja klasy `Rainbow` będzie dysponować informacją o sposobie iteracji kolorów. W najprostszym wariancie możemy użyć metody `each`:

```
r = Rainbow.new
r.each do |color|
  puts "Następny kolor: #{color}"
end
```

Oto dane wyjściowe tej prostej iteracji:

```
Następny kolor: czerwony
Następny kolor: pomarańczowy
Następny kolor: żółty
Następny kolor: zielony
Następny kolor: niebieski
Następny kolor: indygowy
Następny kolor: fioletowy
```

To jednak dopiero początek. Ze względu na to, że klasa `Rainbow` dołączyła moduł `Enumerable`, instancje tej klasy są automatycznie wyposażane w mnóstwo metod utworzonych na bazie metody `each`.

Oto przykład: wartość `true` została przekazana przez metodę `find` zwracającą pierwszy element w obiekcie wyliczeniowym, dla którego udostępniono blok kodu. Załóżmy, że ma zostać znaleziony pierwszy kolor o nazwie zaczynającej się od litery `c`. W tym celu w następujący sposób zostanie użyta metoda `find`:

```
r = Rainbow.new
c_color = r.find {|color| color.start_with?('c') }
puts "Pierwszy kolor o nazwie zaczynającej się od litery c to #{c_color}."
```

**Dane wyjściowe: Pierwszy kolor o nazwie zaczynającej się od litery „c" to czerwony**

Działanie metody `find` polega na wywołaniu metody `each`, która przekazuje elementy. Z kolei metoda `find` używa zapewnionego bloku kodu do testowania tych elementów po jednym na raz pod kątem dopasowania. Gdy metoda `each` zajmie się przekazaniem

elementu `żółty`, metoda `find` przetworzy go przy użyciu bloku, pomyślnie wykonując test. A zatem zmienna `y_color` odbiera wartość `żółty`. Zauważ, że nie ma potrzeby definiowania metody `find`. Jest ona częścią modułu `Enumerable`, który został dołączony. Moduł dysponuje informacjami o tym, jak ma zadziałać, a także w jaki sposób w tym celu wykorzystać metodę `each`.

Definiowanie metody `each` wraz z dołączaniem modułu `Enumerable` zapewnia obiektom mnóstwo funkcjonalności. Spora część funkcji związanych z wyszukiwaniem i odpytywaniem obecnych w tablicach, tablicach asocjacyjnych oraz innych obiektach kolekcji języka Ruby pochodzi bezpośrednio z modułu `Enumerable`. Aby dowiedzieć się, jakie metody zapewnia ten moduł, użyj następującego kodu:

```
>> Enumerable.instance_methods(false).sort
=> [:all?, :any?, :chunk, :collect, :collect_concat, :count, :cycle, :detect,
:drop, :drop_while, :each_cons, :each_entry, :each_slice, :each_with_index,
:each_with_object, :entries, :find, :find_all, :find_index, :first,
:flat_map, :grep, :group_by, :include?, :inject, :lazy, :map, :max, :max_by,
:member?, :min, :min_by, :minmax, :minmax_by, :none?, :one?, :partition,
:reduce, :reject, :reverse_each, :select, :slice_before, :sort, :sort_by,
:take, :take_while, :to_a, :to_h, :zip]
```

Dzięki argumentowi `false` lista zawiera tylko metody zdefiniowane w samym module `Enumerable`. Każda z tych metod bazuje na metodzie `each`.

W dalszych punktach zostaną zaprezentowane przykłady wielu spośród tych metod. Część pozostałych metod będzie się pojawiać w kolejnych rozdziałach. Przykłady znajdujące się w reszcie rozdziału będą korzystać w mniej lub bardziej swobodny sposób z wszystkich czterech podstawowych klas kolekcji: `Array`, `Hash`, `Range` i `Set`. W rozdziale 9. dokonano osobnego wprowadzenia w te klasy. Jeśli orientujesz się w tym, co powoduje działanie każdej z nich, jesteś odpowiednio przygotowany do poznania ich wspólnych elementów.

Niektóre metody klas wyliczeniowych języka Ruby są w rzeczywistości przesłaniane w nich samych. Na przykład w pliku kodu źródłowego *array.c* napotkasz implementacje metod `map`, `select`, `sort` oraz innych metod instancji modułu `Enumerable`. Klasa `Array` nie zapewnia po prostu metody `each` ani nie dołącza modułu `Enumerable` (pomimo tego zyskuje zachowania za jego pośrednictwem). Takie przesłonięcia są stosowane, ponieważ dana klasa wymaga specjalnego zachowania w przypadku danej metody modułu `Enumerable`, albo ze względu na efektywność. Nie będziemy analizować wszystkich przesłonięć. W tym miejscu głównym celem jest poznanie sposobów, za pomocą których klasy kolekcji współużytkują zachowania i interfejs.

W dalszej kolejności przyjrzymy się kilku kategoriom metod modułu `Enumerable`. Zaczniemy od wybranych metod boolowskich.

## 10.2. Zapytania boolowskie dotyczące modułu Enumerable

Kilka metod modułu `Enumerable` zwraca wartość prawda lub fałsz, zależnie od tego, czy jeden lub więcej elementów spełnia określone kryteria. Dla danej tablicy `array`, która zawiera nazwy wszystkich stanów w Stanach Zjednoczonych, wybrane zapytania boolowskie mogą zostać wykonane w następujący sposób:

```
# Czy tablica zawiera stan Luizjana?
>> states.include?("Luizjana")
=> true
# Czy wszystkie stany zawierają w nazwie spację?
>> states.all? {|state| state =~ / / }
=> false
# Czy jakikolwiek stan zawiera w nazwie spację?
>> states.any? {|state| state =~ / / }
=> true
# Czy istnieje wyłącznie jeden stan zawierający w nazwie słowo Zachodnia?
>> states.one? {|state| state =~ /Zachodnia/ }
=> true
# Czy nie istnieją stany zawierające w nazwie słowo Wschodnia?
>> states.none? {|state| state =~ /Wschodnia/ }
=> true
```

Gdyby obiekt `states` nie był tablicą, lecz tablicą asocjacyjną z nazwami stanów i ich skrótami będącymi odpowiednio kluczami i wartościami, możliwe byłoby wykonanie podobnych testów, choć konieczne byłoby uwzględnienie tego, że metoda `Hash#each` każdorazowo przekazuje zarówno klucz, jak i wartość. Metoda `Hash#include?` sprawdza, czy dołączono klucz, o czym była mowa w rozdziale 9., ale inne metody z poprzedniego przykładu obsługują pary klucz-wartość:

```
# Czy tablica asocjacyjna zawiera stan Luizjana?
>> states.include?("Luizjana")
=> true
# Czy wszystkie stany zawierają w nazwie spację?
>> states.all? {|state, abbr| state =~ / / }
=> false
# Czy istnieje wyłącznie jeden stan zawierający w nazwie słowo Zachodnia?
>> states.one? {|state, abbr| state =~ /Zachodnia/ }
=> true
```

**Metoda `include?` sprawdza klucze tablicy asocjacyjnej**

**Tablica asocjacyjna przekazuje blokowi pary klucz-wartość**

We wszystkich tych przypadkach możesz uzyskać tablicę za pośrednictwem wywołania `states.keys` i wykonać testy bezpośrednio dla tej tablicy:

```
# Czy wszystkie stany zawierają w nazwie spację?
>> states.keys.all? {|state, abbr| state =~ / / }
=> false
```

Uprzednie generowanie całej tablicy `keys` zamiast przetwarzania już istniejącej tablicy asocjacyjnej to w pewnym stopniu marnotrawstwo pamięci. Nowa tablica zawiera jednak już istniejące obiekty kluczy, dlatego „marnotrawi" tylko pamięć przeznaczoną na opakowywanie kluczy w tablicy. Wielkość pamięci wykorzystanej przez same klucze nie zwiększa się.

A co ze zbiorami i zakresami? Iteracja zbioru działa bardzo podobnie do iteracji tablicy przeprowadzanej na potrzeby zapytania boolowskiego (oraz większości innych): gdyby tablica `states` była zbiorem, możliwe byłoby zastosowanie dokładnie tych samych zapytań co w przykładzie z identycznymi wynikami. W przypadku zakresów możliwość wyliczania staje się trochę bardziej utrudniona.

Postrzeganie niektórych zakresów jako wyliczeniowych (w postaci kolekcji elementów, które mogą być kolejno przetwarzane) ma większy sens niż innych. Metoda

**Tablice asocjacyjne iterowane za pomocą tablic dwuelementowych**

W przypadku iteracji tablicy asocjacyjnej przy użyciu metody `each` lub dowolnego innego wbudowanego iteratora tablica ta jest przekazywana blokowi kodu po jednej parze klucz-wartość na raz. Pary są tablicami dwuelementowymi. W razie potrzeby możesz udostępnić tylko jeden parametr bloku i przechwycić całą niewielką tablicę:

```
hash.each {|pair| ... }
```

W takim przypadku klucz znajdziesz na pozycji `pair[0]`, a wartość na pozycji `pair[1]`. Zwykle większy sens ma pobranie klucza i wartości w osobnych parametrach bloku. To, co jednak ma tutaj miejsce, polega na opakowywaniu klucza i wartości w tablicy dwuelementowej, która jest przekazywana. Jeśli chcesz przetwarzać dane w takiej postaci, jest to możliwe.

`include?` może być użyta dla dowolnego zakresu. Jednak inne metody boolowskie modułu `Enumerable` powodują przeforsowanie kwestii możliwości wyliczania: jeśli zakres może być wyrażony jako lista osobnych elementów, takie metody zadziałają. W przeciwnym jednak razie, jak w przypadku zakresu liczb zmiennoprzecinkowych, wywołanie dowolnej z metod spowoduje wyzwolenie błędu krytycznego:

```
>> r = Range.new(1, 10)
=> 1..10
>> r.one? {|n| n == 5 }
=> true
>> r.none? {|n| n % 2 == 0 }
=> false
>> r = Range.new(1.0, 10.0)
=> 1.0..10.0
>> r.one? {|n| n == 5 }
TypeError: can't iterate from Float
>> r = Range.new(1, 10.3)
=> 1..10.3
>> r.any? {|n| n > 5 }
=> true
```

Dla danego zakresu obejmującego dwie liczby całkowite możesz wykonać testy, takie jak `one?` i `none?` ❶, ponieważ zakres z łatwością może zacząć zachowywać się jak kolekcja: w efekcie zakres `1..10` adaptuje interfejs API odpowiedniej tablicy `[1,2,3,4,5,6,7,8,9,10]`.

Zakres między dwoma liczbami zmiennoprzecinkowymi ❷ nie może się zachowywać jak skończona kolekcja dyskretnych wartości. Bezsensowne jest generowanie liczby zmiennoprzecinkowej metody `each` wewnątrz zakresu. Zakres zawiera tę metodę, ale jest ona zdefiniowana w taki sposób, aby odmawiać iteracji liczb zmiennoprzecinkowych ❸. To, że błędem jest błąd `TypeError`, a nie błąd `NoMethodError`, wskazuje, że metoda `each` istnieje, ale nie może realizować działań dla danego zakresu.

Liczby zmiennoprzecinkowej możesz użyć jako punktu końcowego zakresu i w dalszym ciągu mieć możliwość wyliczania, pod warunkiem że punktem początkowym jest liczba całkowita ❹. W przypadku wywołania metody `each` (lub jednej z metod opartych na tej metodzie) zakres zachowuje się jak kolekcja liczb całkowitych rozpoczynających się od punktu początkowego, a kończących na punkcie końcowym zaokrąglonym w dół do najbliższej liczby całkowitej. Liczba ta jest traktowana jako należąca

do zakresu, niezależnie od tego, czy zakres jest włączny, czy rozłączny (wynika to stąd, że w końcu oficjalny punkt końcowy jest liczbą zmiennoprzecinkową znajdującą się wyżej od liczby całkowitej).

Poza tym, że zapewniają odpowiedzi na różne pytania typu prawda/fałsz, które dotyczą ich zawartości, obiekty wyliczeniowe wyróżniają się pod względem realizowania operacji wyszukiwania i wybierania. Zajmiemy się nimi teraz.

## 10.3. Wyszukiwanie i wybieranie obiektów wyliczeniowych

Często wymagane jest filtrowanie kolekcji obiektów na podstawie jednego lub większej liczby kryteriów wyboru. Jeśli na przykład istnieje baza danych osób rejestrujących się na konferencję, a ponadto chcesz wysłać przypomnienia o płatności osobom, które jej nie dokonały, kompletną listę możesz filtrować na podstawie statusu płatności. Możesz też wymagać zawężenia listy numerów tylko do parzystych itd. Przypadki użycia dotyczące wybierania elementów z obiektów wyliczeniowych są nieograniczone.

Moduł `Enumerable` zapewnia kilka udogodnień na potrzeby filtrowania kolekcji i przeszukiwania ich w celu znalezienia jednego lub większej liczby elementów zgodnych z co najmniej jednym kryterium. Przyjrzymy się tutaj kilku metodom filtrowania i wyszukiwania. Wszystkie z nich są iteratorami: oczekują na udostępnienie bloku kodu. Blok kodu jest filtrem wyboru. Wewnątrz bloku definiujesz kryteria wyboru (testy dotyczące włączania i wykluczania). Wartość zwracana całej metody może, w zależności od tego, jaka używana jest metoda, i co ona znajduje, być jednym obiektem, tablicą (być może pustą) obiektów spełniających kryteria lub obiektem `nil` wskazującym, że kryteria nie zostały spełnione.

Zaczniemy od wyszukiwania jednego obiektu za pomocą metody `find`, a następnie zajmiemy się kolejno kilkoma technikami umożliwiającymi uzyskanie wieloobiektowego zestawu wynikowego z zapytania wyliczeniowego.

### 10.3.1. Uzyskiwanie pierwszego dopasowania za pomocą metody find

Metoda `find` (dostępna też jako synonim w postaci metody `detect`) znajduje w tablicy pierwszy element, dla którego blok kodu (wywołany z tym elementem jako argumentem) zwraca wartość prawda. Aby na przykład w tablicy liczb całkowitych znaleźć pierwszą liczbę większą niż `5`, możesz użyć metody `find` w następujący sposób:

```
>> [1,2,3,4,5,6,7,8,9,10].find {|n| n > 5 }
=> 6
```

Metoda `find` dokonuje iteracji tablicy, przekazując kolejno każdy jej element blokowi. Jeśli blok nie zwraca niczego z wartością boolowską prawda, przekazany element „wygrywa", a metoda `find` kończy iterację. Jeżeli metodzie nie uda się znaleźć elementu, dla którego pomyślnie kończy się test bloku kodu, zwraca ona obiekt `nil` (aby się o tym przekonać, spróbuj w przykładzie zmienić `n > 5` na `n > 100`). Interesujące będzie rozważenie przypadku, w którym tablica zawiera obiekt `nil` jako jeden ze swoich elementów, a używany blok kodu szuka elementu równego temu obiektowi:

```
[1,2,3,nil,4,5,6].find {|n| n.nil? }
```

W takiej sytuacji metoda `find` zawsze zwraca obiekt `nil`, niezależnie od tego, czy wyszukiwanie kończy się pomyślnie, czy nie. Oznacza to, że test jest nieprzydatny. Nie możesz stwierdzić, czy się powiódł. Aby poradzić sobie z taką sytuacją, możesz skorzystać z innych technik, takich jak metoda `include?`; dzięki nim możesz określić, czy tablica zawiera element w postaci obiektu `nil`. Jako argumentu metody `find` możesz też użyć funkcji „nic nie znaleziono", czyli obiektu `Proc`. W takim przypadku funkcja ta będzie wywoływana, jeśli operacja metody `find` zakończy się niepowodzeniem. Nie omawiano jeszcze obszernie obiektów `Proc`, choć zaprezentowano kilka ich przykładów w nawiązaniu do obsługi bloków kodu. Oto przykład umożliwiający późniejsze odwoływanie się, który prezentuje sposób zapewniania metodzie `find` funkcji obsługującej niepowodzenie:

```
>> failure = lambda { 11 }          ←❶
=> #<Proc:0x434810@(irb):6 (lambda)>
>> over_ten = [1,2,3,4,5,6].find(failure) {|n| n > 10 }
=> 11
```

W tym przykładzie funkcja anonimowa (obiekt `Proc`) zwraca wartość `11` ❶, dlatego jeśli nawet w tablicy nie ma liczby większej niż `10`, i tak uzyskasz taką większą wartość (w rozdziale 14. dokładniej omówiono lambdy i obiekty `Proc`).

Choć metoda `find` zawsze zwraca jeden obiekt, metoda `find_all`, znana również pod nazwą `select`, zawsze zwraca tablicę tak samo jak jej przeciwstawny odpowiednik, czyli metoda `reject`.

**Dominacja tablicy**

Tablice pełnią ogólnie rolę kontenerów w przypadku większości wyników uzyskiwanych z operacji wybierania i filtrowania zapewniających możliwość wyliczania, niezależnie od tego, czy obiekt, dla którego wykonywane są te operacje, to tablica, czy nie. Występują wyjątki od tej quasi-reguły, ale generalnie ma ona zastosowanie.

Najprostszym sposobem przekonania się o tym jest utworzenie własnej klasy wyliczeniowej i obserwowanie tego, co jest uzyskiwane w odpowiedzi z zapytań operacji wybierania. Przyjrzyj się ponownie klasie `Rainbow` z listingu 10.1. Spójrz teraz na wynik uzyskany po wykonaniu kilku zapytań:

```
>> r = Rainbow.new
=> #<Rainbow:0x45b708>
>> r.select {|color| color.size == 8 }
=> ["czerwony", "indygowy"]
>> r.map {|color| color[0,3] }
=> ["cze", "pom", "żół", "zie", "nie", "ind", "fio"]
>> r.drop_while {|color| color.size < 5 }
=> ["czerwony", "pomarańczowy", "żółty", "zielony", "niebieski", "indygowy", "fioletowy"]
```

W każdym przypadku zestaw wynikowy jest zwracany w tablicy.

Tablica to najbardziej ogólny kontener, a zatem logiczny kandydat do roli uniwersalnego formatu wyniku. Pojawia się kilka wyjątków. W przypadku operacji metody `select` lub `reject` tablica asocjacyjna zwraca tablicę asocjacyjną. Zbiory zwracają tablice dla operacji metody `map`, ale możesz wywołać metodę `map!` dla zbioru, aby wewnętrznie zmienić jego elementy. Przeważnie jednak wyniki operacji wybierania i filtrowania z możliwością wyliczenia są zwracane wewnątrz tablic.

### 10.3.2. Uzyskiwanie wszystkich dopasowań za pomocą metod find_all (inaczej select) i reject

Metoda `find_all` (inaczej metoda `select`) zwraca nową kolekcję zawierającą wszystkie elementy oryginalnej kolekcji, które spełniają kryteria w bloku kodu, a nie tylko pierwszy taki element (jak w przypadku metody `find`). Jeśli nie znaleziono żadnych pasujących elementów, metoda `find_all` zwraca pusty obiekt kolekcji.

W ogólnym przypadku, na przykład gdy we własnych klasach używasz modułu `Enumerable`, „kolekcja" zwracana przez metodę `select` będzie tablicą. W języku Ruby są jednak realizowane specjalne działania wobec tablic asocjacyjnych i zbiorów: jeśli dla tablicy asocjacyjnej lub zbioru użyjesz metody `select`, w odpowiedzi uzyskasz tablicę asocjacyjną lub zbiór. Jest to rozszerzone zachowanie, które nie musi być częścią modułu `Enumerable`.

Pozostaniemy tutaj przy przykładach z tablicami:

```
>> a = [1,2,3,4,5,6,7,8,9,10]
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> a.find_all {|item| item > 5 }
=> [6, 7, 8, 9, 10]          ←❶
>> a.select {|item| item > 100 }
=> []                        ←❷
```

Pierwsza operacja metody `find_all` zwraca tablicę wszystkich elementów, które przechodzą pomyślnie test w bloku, czyli są większe od wartości `5` ❶. Druga operacja także zwraca tablicę, która tym razem składa się z wszystkich elementów oryginalnej tablicy większych od wartości `10`. Ponieważ żaden element nie spełnia tego warunku, zwracana jest pusta tablica ❷.

W przypadku tablic, tablic asocjacyjnych i zbiorów dostępna jest wersja metody `select` z wykrzyknikiem w nazwie (`select!`) trwale zmniejszająca kolekcję tylko do tych elementów, które pomyślnie przeszły test operacji wybierania. Nie istnieje synonim w postaci metody `find_all!`. Musisz użyć metody `select!`.

Tak jak możesz wybierać elementy, tak też możesz je odrzucać. Oznacza to określenie, jakie elementy tablicy nie zwracają wartości prawda po przekazaniu do bloku. Używając tablicy `a` z poprzedniego przykładu, możesz zastosować operację odrzucania w celu uzyskania tablicy pozbawionej wszystkich elementów większych od wartości `5`:

```
>> a.reject {|item| item > 5 }
=> [1, 2, 3, 4, 5]
```

I tym razem występuje wersja metody `reject!` specjalnie przeznaczonej dla tablic, tablic asocjacyjnych i zbiorów.

Jeśli kiedykolwiek używałeś narzędzia wiersza poleceń `grep`, następna metoda okaże się znajoma. W przeciwnym razie i tak szybko ją opanujesz.

### 10.3.3. Wybieranie dopasowań operatora równości === za pomocą metody grep

Metoda `Enumerable#grep` umożliwia wykonanie operacji wybierania w odniesieniu do obiektu wyliczeniowego na bazie operatora równości `===`. Najpowszechniejsze zastosowanie tej metody odpowiada w największym stopniu typowej operacji realizowanej

przez narzędzie wiersza poleceń o takiej samej nazwie, czyli dopasowywaniu wzorców dla łańcuchów:

```
>> colors = %w{ czerwony pomarańczowy żółty zielony niebieski indygowy fioletowy }
=> ["czerwony", "pomarańczowy", "żółty", "zielony", "niebieski", "indygowy", "fioletowy"]
>> colors.grep(/o/)
=> ["czerwony", "pomarańczowy", "zielony", "indygowy", "fioletowy"]
```

Ogólność operatora === umożliwia jednak zrealizowanie za pomocą metody `grep` kilku wyszukanych działań:

```
>> miscellany = [75, "witaj", 10...20, "żegnaj"]
=> [75, "witaj", 10...20, "żegnaj"]
>> miscellany.grep(String)
=> ["witaj", "żegnaj"]                    ❶
>> miscellany.grep(50..100)
=> [75]                                   ❷
```

Wyrażenie `String === object` jest prawdziwe dla dwóch łańcuchów w tablicy, dlatego tablica złożona z tych dwóch łańcuchów jest wynikiem zwracanym po użyciu metody `grep` dla argumentu `String` ❶. Zakresy implementują operator === jako test włączania. Zakres `50..100` obejmuje wartość `75`, stąd też jest to wynik użycia metody `grep` dla obiektu `miscellany` z tym zakresem jako argumentem ❷.

Generalnie pod względem funkcjonalnym instrukcja `enumerable.grep(`*`wyrażenie`*`)` jest równoważna następującemu wierszowi:

```
enumerable.select {|element| wyrażenie === element }
```

Inaczej mówiąc, instrukcja wybiera wartość prawda w oparciu o wywołanie ===. Ponadto metoda `grep` może pobrać blok. W tym przypadku przekazuje ona blokowi każdy element swojego zestawu wynikowego przed zwróceniem wyników:

```
>> colors = %w{ czerwony pomarańczowy żółty zielony niebieski indygowy fioletowy }
=> ["czerwony", "pomarańczowy", "żółty", "zielony", "niebieski", "indygowy", "fioletowy"]
>> colors.grep(/o/) {|color| color.capitalize }
=> ["Pomarańczowy", "Żółty", "Indygowy", "Fioletowy"]
```

Oto pełna składnia metody `grep`:

```
enumerable.grep(wyrażenie) {|element| ... }
```

Oznacza to, że w efekcie metoda działa w następujący sposób:

```
enumerable.select {|element| wyrażenie === element}.map {|element| ... }
```

Z metodą `grep` będziesz się najczęściej spotykać (i prawdopodobnie najczęściej będziesz jej w ten sposób używać) jako z selektorem łańcuchów opartym na wzorcu. Pamiętaj jednak o tym, że operacja realizowana przez tę metodę jest powiązana z operatorem równości przypadków ===, dlatego może być stosowana odpowiednio w różnych sytuacjach.

Niezależnie od tego, czy wybieranie jest realizowane z wykorzystaniem metody `select` lub `grep`, czy jakiejś innej operacji, często wymagają one grupowania wyników w obrębie klastrów lub kategorii. Metody `Enumerable#group_by` i `#partition` ułatwiają przeprowadzenie dokładnie tego rodzaju grupowania.

### 10.3.4. Organizowanie wyników wybierania za pomocą metod group_by i #partition

W ramach operacji metody group_by dotyczącej obiektu wyliczeniowego pobierany jest blok i zwracana tablica asocjacyjna. Blok wykonywany jest dla każdego obiektu. Dla każdej unikatowej wartości zwracanej bloku tablica asocjacyjna wyniku uzyskuje klucz. Jego wartością jest tablica wszystkich elementów wyliczeniowych, dla których blok zwrócił wartość.

Następujący przykład powinien sprawić, że działanie metody group_by będzie przejrzyste:

```
>> colors = %w{ czerwony pomarańczowy żółty zielony niebieski indygowy fioletowy }
=> ["czerwony", "pomarańczowy", "żółty", "zielony", "niebieski", "indygowy", "fioletowy"]
>> colors.group_by {|color| color.size }
=> {8=>["czerwony", "indygowy"], 12=>["pomarańczowy"], 5=>["żółty"],
 7=>["zielony"], 9=>["niebieski", "fioletowy"]}
```

Blok {|color| color.size } zwraca liczbę całkowitą dla każdego koloru. Tablica asocjacyjna zwracana przez operację metody group_by jest rozdzielana do kluczy różnych wielkości (5, 7, 8, 9, 12). Wartości są tablicami zawierającymi wszystkie łańcuchy z oryginalnej tablicy, które mają wielkość reprezentowaną przez odpowiednie klucze.

Metoda partition przypomina metodę group_by, ale dzieli elementy wyliczeniowe na dwie tablice na podstawie tego, czy blok kodu zwraca dla elementu wartość prawda.

Nie występuje tablica asocjacyjna, a jedynie tablica złożona z dwóch tablic. Dwie tablice są zawsze zwracane w kolejności wartości prawda/fałsz.

Przyjrzyj się klasie Person, w której każda osoba ma określony wiek. Klasa definiuje również metodę instancji teenager?, która zwraca wartość prawda, gdy wiek osoby zawiera się w przedziale od 13 do 19 włącznie:

```
class Person
  attr_accessor :age
  def initialize(options)
    self.age = options[:age]
  end
  def teenager?
    (13..19) === age
  end
end
```

Wygenerujmy teraz tablicę osób:

```
people = 10.step(25,3).map {|i| Person.new(:age => i) }
```

Kod ten dokonuje iteracji wartości od 10 do 25 w odstępach o wartości 3 (10, 13, 16, 19, 22, 25), przekazując kolejno do bloku każdą z wartości. Każdorazowo tworzony jest nowy obiekt Person z wiekiem odpowiadającym inkrementacji. Dzięki metodzie map obiekty Person są akumulowane w tablicy przypisanej zmiennej people (tworzenie łańcucha między iteratorem map i iteratorem step jest możliwe dzięki temu, że metoda step zwraca enumerator; wkrótce dowiesz się więcej o enumeratorach).

Utworzono sześć obiektów osób Person. Przeprowadźmy teraz partycjonowanie ich na nastolatków i pozostałe osoby:

```
teens =  people.partition {|person| person.teenager? }
```

Tablica `teens` ma następującą zawartość:

```
[[#<Person:0x000001019d1a50 @age=13>, #<Person:0x000001019d19d8 @age=16>,
#<Person:0x000001019d1988 @age=19>], [#<Person:0x000001019d1ac8 @age=10>,
#<Person:0x000001019d1910 @age=22>, #<Person:0x000001019d1898 @age=25>]]
```

Zauważ, że jest to tablica zawierająca dwie tablice podrzędne. W pierwszej znajdują się osoby, dla których metoda `person.teenager?` zwróciła wartość prawda. Druga tablica reprezentuje osoby, które nie są nastolatkami.

Możemy teraz użyć informacji, aby na przykład dowiedzieć się, ilu jest nastolatków, a ile pozostałych osób:

```
puts "nastolatki: #{teens[0].size}; inne osoby: #{teens[1].size}"
```

Dane wyjściowe tej instrukcji odzwierciedlają fakt, że połowa osób to nastolatkowie, a połowa to inne osoby:

```
nastolatkowie: 3; inne osoby: 3
```

Przyjrzyjmy się teraz kilku operacjom dotyczących elementów, które uwzględniają stosunkowo szczegółowe modyfikowanie konkretnych elementów kolekcji.

## 10.4. Operacje wyliczeniowe dotyczące elementów

Z założenia kolekcje mają być poddawane operacji przechodzenia, ale zawierają też pojedyncze obiekty o specjalnym statusie: jest to pierwszy lub ostatni obiekt w kolekcji oraz największy lub najmniejszy obiekt. Obiekty `Enumerable` są wyposażone w kilka narzędzi służących do obsługi elementów.

### 10.4.1. Metoda first

Jak wskazuje nazwa, metoda `Enumerable#first` zwraca pierwszy element napotkany podczas iteracji obiektów wyliczeniowych:

```
>> [1,2,3,4].first
=> 1
>> (1..10).first
=> 1
>> {1 => 2, "jeden" => "dwa"}.first
=> [1, 2]
```

Obiekt zwracany przez metodę `first` jest taki sam jak pierwszy obiekt uzyskiwany podczas iteracji obiektu nadrzędnego. Inaczej mówiąc, jest to pierwszy element przekazywany przez metodę `each`. Biorąc pod uwagę to, że tablice asocjacyjne przekazują pary klucz-wartość w tablicach dwuelementowych, pobranie pierwszego elementu tablicy asocjacyjnej zapewnia tablicę dwuelementową, która zawiera pierwszą parę wstawioną do tablicy asocjacyjnej (lub pierwszy klucz wstawiony wraz z jego nową wartością, jeśli została ona zmieniona w dowolnym miejscu):

```
>> hash = { 3 => "trzy", 1 => "jeden", 2 => "dwa" }
=> {3=>"trzy", 1=>"jeden", 2=>"dwa"}
>> hash.first                    ← Metoda first oznacza wstawienie pierwszego elementu
=> [3, "trzy"]
>> hash[3] = "trois"
```

```
=> "trois"
>> hash.first  ←——— Nowa wartość nie zmienia kolejności wstawiania
=> [3, "trois"]
```

Być może najbardziej interesującą kwestią związaną z metodą `Enumerable#first` jest to, że nie występuje metoda `Enumerable#last`. Wynika to stąd, że znajdowanie końca iteracji nie jest tak proste jak znajdowanie początku. Rozważ przypadek, w którym iteracja trwa bez końca. Poniżej zaprezentowano niewielką klasę `Die` (z obiektami kostki do gry). Iteracja tej klasy odpowiada ciągłemu rzucaniu kostką do gry i każdorazowemu przekazywaniu wyniku:

```
class Die
  include Enumerable
  def each
    loop do
      yield rand(6) + 1
    end
  end
end
```

Pętla korzysta z metody `Kernel#rand`. Gdy zostanie wywołana bez argumentu, metoda ta generuje losową liczbę zmiennoprzecinkową $n$ taką, że $0 <= n < 1$. Dla argumentu $i$ metoda zwraca losową liczbę całkowitą $n$ taką, że $0 <= n < i$. A zatem metoda `rand(6)` generuje liczbę całkowitą w zakresie `(0..5)`. Dodanie jedynki do tej liczby da liczbę z przedziału od `1` do `6`, która odpowiada temu, co zostanie uzyskane po rzuceniu kostką.

Główną kwestią jednak jest to, że metoda `Die#each` nieustannie działa. Jeśli używasz klasy `Die`, musisz zapewnić przerwanie pętli. Oto niewielka gra, w której wygrana jest osiągana od razu po wyrzuceniu na kostce sześciu oczek:

```
puts "Witamy w grze Wygrywasz po wyrzuceniu sześciu oczek!"
d = Die.new
d.each do |roll|
  puts "Liczba wyrzuconych oczek: #{roll}."
  if roll == 6
    puts "Wygrałeś!"
    break
  end
end
```

Oto typowe dane wyjściowe po uruchomieniu tego kodu:

```
Witamy w grze Wygrywasz po wyrzuceniu sześciu oczek!
Liczba wyrzuconych oczek: 3.
Liczba wyrzuconych oczek: 2.
Liczba wyrzuconych oczek: 2.
Liczba wyrzuconych oczek: 1.
Liczba wyrzuconych oczek: 6.
Wygrałeś!
```

Pomijając trywialność gry, rzecz w tym, że bezsensowne byłoby wywoływanie metody `last` dla obiektu kostki, ponieważ nie występuje ostatni rzut kostką. W przeciwieństwie do pobierania pierwszego elementu pobieranie ostatniego elementu obiektu wyliczeniowego nie ma żadnego ogólnego znaczenia.

Z tego samego powodu, czyli nieosiągalności końca wyliczania, klasa wyliczeniowa z przekazującą bez końca wynik metodą `each` nie może zbyt dużo osiągnąć w przypadku metod takich jak `select` i `map`, które nie zwracają swoich wyników do momentu zakończenia bazowej iteracji. W dowolnej sytuacji niewiele jest okazji do wystąpienia nieskończonej iteracji, ale obserwacja zachowania i wpływu nieskończonej metody `each` może być pouczająca pod względem tego, co ujawnia ona w nawiązaniu do powszechniejszej iteracji skończonej.

Pamiętaj jednak o tym, że niektóre klasy wyliczeniowe, a w szczególności klasy `Array` i `Range`, dysponują metodą `last`. Co więcej, wszystkie obiekty wyliczeniowe zawierają metodę `take` (coś w rodzaju uogólnienia metody `first`) oraz towarzyszącą jej metodę `drop`.

### 10.4.2. Metody take i drop

Obiekty wyliczeniowe potrafią „pobrać" określoną liczbę elementów z własnego początku, a w odwrotnym wariancie mają możliwość „usunięcia" konkretnej liczby elementów. Zasadniczo operacje metod `take` i `drop` realizują to samo, czyli dokonują podziału kolekcji w określonym miejscu. Różnią się jednak tym, co zwracają:

```
>> states = %w{ NJ NY CT MA VT FL }
=> ["NJ", "NY", "CT", "MA", "VT", "FL"]
>> states.take(2)                    ←—— Pobiera pierwsze dwa elementy
=> ["NJ", "NY"]
>> states.drop(2)        ←—— Pobiera kolekcję, z wyjątkiem pierwszych dwóch elementów
=> ["CT", "MA", "VT", "FL"]
```

Pobranie elementów umożliwia ich uzyskanie. Usunięcie elementów powoduje uzyskanie oryginalnej kolekcji pomniejszonej o usunięte elementy. Możesz ograniczyć operacje realizowane przez metody `take` i `drop`, udostępniając blok i używając wariantów tych metod w postaci metod `take_while` i `drop_while`, które określają wielkość „pobrania" nie za pomocą argumentu jako liczby całkowitej, lecz przy użyciu wartości prawda bloku:

```
>> states.take_while {|s| /N/.match(s) }
=> ["NJ", "NY"]
>> states.drop_while {|s| /N/.match(s) }
=> ["CT", "MA", "VT", "FL"]
```

Operacje metod `take` i `drop` są czymś w rodzaju hybrydy metod `first` i `select`. Operacje te są powiązane z początkiem iteracji. Kończą się po spełnieniu wymagania dotyczącego ilości lub wystąpieniu niepowodzenia bloku.

Możliwe jest też określenie wartości minimalnej i maksymalnej w kolekcji obiektów wyliczeniowych.

### 10.4.3. Metody min i max

Metody `min` i `max` realizują to, co sugerują ich nazwy:

```
>> [1,3,5,4,2].max
=> 5
>> %w{ Ruby C APL Perl Smalltalk }.min
=> "APL"
```

Minimum i maksimum są ustalane zgodnie z logiką operatora porównania <=>, który w przypadku tablicy łańcuchów wstawia łańcuch "APL" jako pierwszy w kolejności rosnącej. Aby przeprowadzić test minimum lub maksimum na podstawie kryterium innego niż domyślne, możesz zapewnić następujący blok kodu:

```
>> %w{ Ruby C APL Perl Smalltalk }.min {|a,b| a.size <=> b.size }
=> "C"
```

Efektywniejszym rozwiązaniem opartym na bloku będzie jednak użycie metody min_by lub max_by, która niejawnie wykonuje operację porównania:

```
>> %w{ Ruby C APL Perl Smalltalk }.min_by {|lang| lang.size }
=> "C"
```

**Nie jest konieczne jawne porównywanie dwóch parametrów w bloku kodu**

Istnieje też metoda minmax (oraz odpowiadająca jej metoda minmax_by), która zapewnia parę wartości, po jednej dla minimum i maksimum:

```
>> %w{ Ruby C APL Perl Smalltalk }.minmax
=> ["APL", "Smalltalk"]
>> %w{ Ruby C APL Perl Smalltalk }.minmax_by {|lang| lang.size }
=> ["C", "Smalltalk"]
```

Pamiętaj o tym, że rodzina metod wyliczeniowych min/max jest dostępna zawsze, nawet wtedy, gdy korzystanie z niej nie jest dobrym pomysłem. Nie byłoby na przykład wskazane zastosowanie następujących wierszy:

```
die = Die.new
puts die.max
```

Pętla nieskończona, w której zaimplementowano metodę Die#each, w ogóle nie umożliwi ustalenia wartości maksymalnej. W efekcie program się zawiesi.

W przypadku tablic asocjacyjnych do określenia uporządkowania metody min i max stosują klucze. Jeśli chcesz użyć wartości, pomocni mogą się okazać członkowie rodziny metod min/max o nazwie reprezentowanej przez wzorzec *_by:

```
>> state_hash = {"Nowy Jork" => "NY", "Maine" => "ME",
"Alaska" => "AK", "Alabama" => "AL" }
=> {"New York"=>"NY", "Maine"=>"ME", "Alaska"=>"AK", "Alabama"=>"AL"}
>> state_hash.min
=> ["Alabama", "AL"]
>> state_hash.min_by {|name, abbr| name }
=> ["Alabama", "AL"]
>> state_hash.min_by {|name, abbr| abbr }
=> ["Alaska", "AK"]
```

**Para minimum według klucza**

**To samo co w przypadku metody min**

**Para minimum według wartości**

Oczywiście w razie potrzeby możesz wykonywać obliczenia wewnątrz bloku, które uwzględniają zarówno klucz, jak i wartość.

Zaprezentowaliśmy już przykłady metody each, a także wyjaśniliśmy, jak jest ona powiązana z kilkoma metodami, które na niej bazują. Pora przyjrzeć się niektórym metodom podobnym do metody each, lecz trochę bardziej specjalizowanym. Wśród tych metod najważniejsza jest metoda map. Okazuje się, że jest ona na tyle istotna, że poświęcimy jej osobny podrozdział. Omówmy najpierw kilka innych metod powiązanych z metodą each.

## 10.5. Metody powiązane z metodą each

Moduł `Enumerable` udostępnia kilka metod przypominających metodę `each` w tym, że przetwarzają całą kolekcję i przekazują z niej elementy, kontynuując działanie do momentu napotkania ostatniego elementu (a w jednym przypadku nawet to nie spowoduje zakończenia ich pracy!). Każdy członek tej rodziny metod ma swoją własną, specyficzną semantyką i przeznaczenie. Do metod tych należą: `reverse_each`, `each_with_index`, `each_slice`, `each_cons`, `cycle` i `inject`. Przyjrzymy się im kolejno.

### 10.5.1. Metoda reverse_each

Metoda `reverse_each` działa zgodnie z tym, na co może wskazywać, czyli dokonuje iteracji obiektu wyliczeniowego w kierunku wstecznym. Oto przykładowy kod:

```
[1,2,3].reverse_each {|e| puts e * 10 }
```

Generuje on następujące dane wyjściowe:

```
30
20
10
```

Przy stosowaniu tej metody musisz zachować ostrożność: nie używaj jej dla iteratora nieskończonego, ponieważ pojęcie przetwarzania wstecznego zależy od znajomości ostatniego elementu, co w przypadku iteratora nieskończonego nie ma sensu. Spróbuj wywołać metodę `reverse_each` dla instancji zaprezentowanej wcześniej klasy `Die`, ale bądź gotów na użycie kombinacji klawiszy *Ctrl+C* w celu wyjścia z pętli nieskończonej.

### 10.5.2. Metoda each_with_index (oraz metoda each.with_index)

Metoda `Enumerable#each_with_index` różni się od metody `each` tym, że przekazuje dodatkowy element każdorazowo przy przetwarzaniu kolekcji. Elementem tym jest liczba całkowita reprezentująca pozycję porządkową elementu. Taki indeks może okazać się przydatny, między innymi podczas etykietowania obiektów:

```
>> names = ["George Washington", "John Adams", "Thomas Jefferson",
"James Madison"]
=> ["George Washington", "John Adams", "Thomas Jefferson",
"James Madison"]
>> names.each_with_index do |pres, i|
?>   puts "#{i+1}. #{pres}"        ← Dodaje wartość 1 w celu uniknięcia zerowej pozycji listy
>> end
1. George Washington
2. John Adams
3. Thomas Jefferson
4. James Madison
```

Z metodą `each_with_index` związana jest anomalia: choć każdy obiekt wyliczeniowy zawiera tę metodę, nie każdy dysponuje informacją o tym, czym jest indeks. Aby się o tym przekonać, zażądaj od takich obiektów wykonania operacji metody `each_index` (zamiast metody `each_with_index`). Wyniki będą różne dla poszczególnych obiektów wyliczeniowych:

```
>> %w{a b c }.each_index {|i| puts i }
0
1
2
=> ["a", "b", "c"]
```

Oznacza to, że tablice mają fundamentalne „poczucie” istnienia indeksu. Tablice asocjacyjne już nie, choć rozpoznają indeks metody `with`:

```
>> letters = {"a" => "ay", "b" => "bee", "c" => "see" }
=> {"a"=>"ay", "b"=>"bee", "c"=>"see"}
>> letters.each_with_index {|(key,value),i| puts i }
0
1
2
=> {"a"=>"ay", "b"=>"bee", "c"=>"see"}
>> letters.each_index {|(key,value),i| puts i }
NoMethodError: undefined method `each_index' for {"a"=>"ay",
"b"=>"bee", "c"=>"see"}:Hash
```

Możemy przyjąć, że klucze tablicy asocjacyjnej są jej indeksami, a liczby porządkowe generowane przez iterację metody `each_with_index` to dodatkowe indeksy lub meta-indeksy. Jest to interesująca kwestia teoretyczna. W praktyce jednak nie ma już tak dużego znaczenia, ponieważ wyjątkowo rzadko pojawia się konieczność wykonywania operacji metody `each_with_index` dla tablicy asocjacyjnej.

Metoda `Enumerable#each_with_index` działa, ale jakoś w ograniczonym zakresie. Zamiast niej rozważmy użycie metody `#with_index` enumeratora uzyskiwanego w wyniku wywołania metody `each`. Technika ta została już zaprezentowana w rozdziale 9.:

```
>> array = %w{ czerwony żółty niebieski }
=> ["czerwony", "żółty", "niebieski"]
>> array.each.with_index do |color, i|
?>   puts "Numer koloru #{i} to #{color}."
>> end
```

Działanie tego kodu jest tak proste jak zastąpienie znaku podkreślenia kropką, choć wewnętrznie realizowanych jest trochę więcej operacji, co się okaże, gdy trochę później dowiesz się więcej o enumeratorach (w punkcie 10.11.2 podano więcej na temat metody `each_index`). Użycie metody `each_index` zapewnia też pewną funkcjonalność: możesz podać argument, który będzie stosowany jako pierwsza wartość indeksu, eliminując dzięki temu konieczność dodawania jedynki do indeksu w sytuacji takiej jak wcześniej zamieszczona lista prezydentów Stanów Zjednoczonych:

```
>> names.each.with_index(1) do |pres, i|
?>   puts "#{i} #{pres}"
>> end
```

Kolejną podrodziną powiązaną z metodą `each` jest para metod `each_slice` i `each_cons`.

### 10.5.3. Metody each_slice i each_cons

Metody `each_slice` i `each_cons` są specjalizowanymi wersjami metody `each`, które jednocześnie przetwarzają określoną liczbę elementów kolekcji, przekazując blokowi w każdej iteracji tablicę zawierającą taką liczbę elementów. Różnica między nimi

polega na tym, że metoda `each_slice` obsługuje każdy element tylko raz, natomiast metoda `each_cons` pobiera dla każdego elementu nową grupę, tym samym generując nakładające się tablice, które są przez metodę przekazywane.

Oto ilustracja tej różnicy:

```
>> array = [1,2,3,4,5,6,7,8,9,10]
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>> array.each_slice(3) {|slice| p slice }
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
[10]
=> nil
>> array.each_cons(3) {|cons| p cons }
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
[4, 5, 6]
[5, 6, 7]
[6, 7, 8]
[7, 8, 9]
[8, 9, 10]
=> nil
```

Operacja metody `each_slice` przekazuje kolekcję stopniowo w porcjach o wielkości *n* (lub mniejszych niż *n*, jeśli pozostało mniej niż *n* elementów). Dla porównania metoda `each_cons` przetwarza po jednym elemencie kolekcji, a ponadto dla każdego z nich przekazuje tablicę *n* elementów, przerywając po jednokrotnym przekazaniu ostatniego elementu kolekcji.

Istnieje jeszcze jeden ogólny sposób iteracji obiektu wyliczeniowego za pomocą metody `cycle`.

### *10.5.4. Metoda cycle*

Metoda `Enumerable#cycle` przekazuje nieustannie wszystkie elementy w obiekcie w ramach pętli. Jeśli podasz argument w postaci liczby całkowitej, pętla będzie uruchamiana liczbę razy określoną przez ten argument. W przeciwnym razie pętla będzie uruchamiana w nieskończoność.

Metoda `cycle` umożliwia określenie w sposób dynamiczny żądanej liczby iteracji w obrębie kolekcji, czyli zasadniczo określenie, ile razy pod rząd mają zostać wykonane operacje uruchomienia podobne jak w przypadku metody `each`. Oto przykład dotyczący talii kart do gry:

```
class PlayingCard
  SUITS = %w{ trefle kara kiery piki }        ❶
  RANKS = %w{ 2 3 4 5 6 7 8 9 10 J Q K A }
  class Deck
    attr_reader :cards                        ❷
    def initialize(n=1)                       ❸
      @cards = []
      SUITS.cycle(n) do |s|                   ❹
        RANKS.cycle(1) do |r|                 ❺
          @cards << "#{r} of #{s}"            ❻
```

```
        end
      end
    end
  end
end
```

Klasa `PlayingCard` definiuje stałe reprezentujące kolory i rangi ❶, natomiast klasa `PlayingCard::Deck` modeluje talię. Karty są przechowywane w tablicy w zmiennej instancji `@cards` talii. Zmienna jest też dostępna jako atrybut odczytu ❷. Dzięki metodzie `cycle` ułatwione jest zapewnienie możliwości połączenia dwóch lub większej liczby talii. Metoda `Deck.new` pobiera argument o wartości domyślnej `1` ❸. W przypadku przesłonięcia tej wartości rozszerzany jest proces, w ramach którego zapełniana jest tablica `@cards`.

Na przykład następujące polecenie zapewnia podwójną talię kart zawierającą po dwie z każdej kart, co w sumie daje 104 karty:

```
deck = PlayingCard::Deck.new(2)
```

Wynika to stąd, że metoda dwukrotnie wykonuje cykl dla kolorów, wykonując cykl dla rang jednokrotnie dla każdej iteracji dotyczącej kolorów ❹. Cykl dla rang zawsze jest wykonywany tylko raz ❺. Oznacza to, że wywołanie `cycle(1)` stanowi inny sposób zdefiniowania metody `each`. Dla każdej permutacji nowa karta reprezentowana przez łańcuch opisowy jest umieszczana w talii ❻.

Ostatnią metodą w omówieniu metod z rodziny `each` jest metoda `inject`, która jest też znana pod nazwą `reduce`.

### 10.5.5. Zmniejszanie obiektu wyliczeniowego za pomocą metody inject

Działanie metody `inject` (znana również pod nazwą `reduce` i przypominająca metody „zwijające" w językach funkcjonalnych) polega na inicjowaniu obiektu akumulatora, a następnie iteracji kolekcji (obiekt wyliczeniowy), wykonywaniu obliczenia dla każdej iteracji i resetowaniu akumulatora na potrzeby następnej iteracji do postaci wyniku tego obliczenia.

Klasycznym przykładem użycia metody `inject` jest sumowanie liczb w tablicy. W tym celu należy użyć następującego wiersza kodu:

```
>> [1,2,3,4].inject(0) {|acc,n| acc + n }
=> 10
```

Proces realizowany przez ten kod jest złożony z następujących kroków:

1. Akumulator jest inicjowany z wartością `0`, dzięki temu, że metodzie `inject` przekazano argument `0`.
2. Podczas pierwszej iteracji (w obrębie bloku kodu) akumulator `acc` ma wartość `0`, a dla pierwszego elementu `n` w tablicy ustawiana jest wartość `1`. Wynikiem obliczenia wewnątrz bloku jest `0 + 1` lub `1`.
3. W przypadku drugiej iteracji dla akumulatora `acc` ustawiana jest wartość `1` (wynik zwrócony z bloku w ramach poprzedniej iteracji), a dla elementu `n` wartość `2` (drugi element tablicy). A zatem blok zwraca wartość `3`.

4. Podczas trzeciej iteracji dla `acc` i `n` ustawiane są wartości `3` (poprzedni wynik bloku) i `3` (następna wartość w tablicy). Blok zwraca wartość `6`.
5. W przypadku czwartej iteracji `acc` i `n` mają wartości `6` i `4`. W bloku określana jest wartość `10`. Ponieważ jest to ostatnia iteracja, wartość uzyskana z bloku pełni rolę wartości zwracanej całego wywołania metody `inject`. Oznacza to, że dla całego wywołania określana jest wartość `10`, co potwierdza wiersz w sesji narzędzia irb.

Jeśli dla metody `inject` nie podasz argumentu, używa ona pierwszego elementu w obiekcie wyliczeniowym jako wartości początkowej akumulatora `acc`. W tym przykładzie spowodowałoby to zwrócenie takiego samego wyniku, ponieważ w pierwszej iteracji i tak do wartości `1` dodano `0` oraz dla akumulatora `acc` ustawiono wartość `1`.

Oto rozbudowany przykład, w którym dla każdej iteracji wyświetlany jest komentarz umożliwiający stwierdzenie, co ma miejsce:

```
>> [1,2,3,4].inject do |acc,n|
     puts "Sumowanie #{acc} i #{n}...#{acc+n}"
     acc + n
   end
Sumowanie 1 i 2...3
Sumowanie 3 i 3...6
Sumowanie 6 i 4...10
=> 10
```

Instrukcja `puts` to czysty efekt uboczny (dla niej samej jest określany obiekt `nil`), dlatego nadal konieczne jest zakończenie bloku kodem `acc + n`, aby zapewnić, że blok określi poprawną wartość.

Prawdopodobnie na koniec pozostała nam do omówienia najważniejsza metoda powiązana z metodą `each`, czyli metoda `Enumerable#map`.

## 10.6. Metoda map

Metoda `map` (znana również pod nazwą `collect`) zapewnia jedną z najważniejszych i bogatych w możliwości operacji na kolekcjach lub obiektach wyliczeniowych w języku Ruby. Choć wspomniano o niej wcześniej (w rozdziale 6.), zapewnia ona więcej możliwości, zwłaszcza w kontekście ogólnego zagadnienia związanego z możliwościami wyliczania.

Niezależnie od tego, co znajduje się na początku obiektu wyliczeniowego, metoda `map` zawsze zwraca tablicę. Zwrócona tablica każdorazowo ma taką samą wielkość jak oryginalny obiekt wyliczeniowy. Elementy tablicy mają postać zakumulowanego wyniku wywołania bloku kodu kolejno dla każdego elementu w oryginalnym obiekcie.

Oto przykład odwzorowania tablicy imion na ich odpowiedniki złożone z dużych liter:

```
>> names = %w{ Dawid Janusz Cezary Anna }
=> ["Dawid", "Janusz", "Cezary", "Anna"]
>> names.map {|name| name.upcase }
=> ["DAWID", "JANUSZ", "CEZARY", "ANNA"]
```

Nowa tablica ma taką samą wielkość jak oryginalna tablica, a każdy z jej elementów odpowiada elementowi oryginalnej tablicy znajdującemu się na tej samej pozycji. Każdy element został jednak przetworzony przez blok.

**Użycie argumentu w postaci symbolu jako bloku**

Symbolu, takiego jak `:upcase`, możesz użyć z umieszczonym przed nim operatorem `&` na pozycji argumentu metody. W tym przypadku wynik będzie taki sam jak w sytuacji, gdy użyto bloku kodu, który wywołał dla każdego elementu metodę o takiej samej nazwie jak nazwa symbolu. Oznacza to, że w następujący sposób możesz zmodyfikować blok w poprzednim przykładzie, który dla każdego elementu wywołuje metodę `upcase`:

```
names.map(&:upcase)
```

Dokładne objaśnienie tego idiomu poznasz w trakcie lektury rozdziału 14. poświęconego obiektom możliwym do wywołania.

Może to być oczywiste, ale warto zauważyć, że w przypadku metody `map` istotna jest jej wartość zwracana.

### 10.6.1. Wartość zwracana metody map

Wartość zwracana metody `map` i przydatność tej wartości są tym, co odróżnia metodę `map` od metody `each`. Wartość zwracana metody `each` nie jest istotna. Prawie nigdy nie spotkasz się z następującym wierszem kodu:

```
result = array.each {|x| # w tym miejscu znajduje się kod... }
```

Dlaczego tak jest? Wynika to stąd, że metoda `each` zwraca swojego odbiorcę. Równie dobrze możesz użyć następującego kodu:

```
result = array
array.each {|x| ... }
```

Z kolei metoda `map` zwraca nowy obiekt: odwzorowanie oryginalnego obiektu na nowy obiekt. A zatem często będziesz mieć do czynienia z następującym wierszem kodu:

```
result = array.map {|x| # w tym miejscu znajduje się kod... }
```

Różnica między metodami `map` i `each` stanowi dobre przypomnienie tego, że metoda `each` istnieje wyłączenie na potrzeby efektów ubocznych wykonania bloku. Wartość zwracana każdorazowo przez blok podczas przetwarzania jest odrzucana. Z tego właśnie powodu metoda `each` zwraca swojego odbiorcę. Metoda nie ma nic innego do zwrócenia, ponieważ niczego nie zapisała. Z kolei metoda `map` zachowuje tablicę akumulatora z wynikami z bloku.

Nie oznacza to, że metoda `map` jest lepsza lub bardziej przydatna niż metoda `each`. Oznacza to, że różnią się one na kilka istotnych sposobów. Semantyka metody `map` powoduje jednak, że musisz zachować ostrożność wobec efektów ubocznych, które sprawiają, że metoda `each` jest przydatna.

**ZACHOWAJ OSTROŻNOŚĆ W PRZYPADKU OKREŚLANIA WARTOŚCI W BLOKU**

Przyjrzyj się następującemu kodowi i zastanów się, czy możesz przewidzieć, co będzie zawierać tablica `result` po wykonaniu kodu:

```
array = [1,2,3,4,5]
result = array.map {|n| puts n * 100 }
```

W odpowiedzi uzyskujemy następującą tablicę `result`:

```
[nil, nil, nil, nil, nil]
```

Dlaczego? Wynika to stąd, że wartością zwracaną metody `puts` zawsze jest obiekt `nil`. Wszystkim tym zajmuje się metoda `map`. Zgadza się, pięć wartości reprezentowanych przez działanie `n * 100` zostanie wyświetlonych na ekranie, ale jest to spowodowane wykonywaniem kodu w bloku. Wynikiem operacji, samego odwzorowania, są same obiekty `nil`, ponieważ każde wywołanie tego konkretnego bloku spowoduje zwrócenie obiektu `nil`.

W przypadku tablic i zbiorów dostępna jest wersja metody map z wykrzyknikiem w nazwie, czyli `map!` (znana też pod nazwą `collect!`).

### *10.6.2. Odwzorowywanie wewnętrzne za pomocą metody map!*

Ponownie przyjrzyj się tablicy `names`:

```
names = %w{ Dawid Janusz Cezary Anna }
```

Aby zmodyfikować wewnętrznie tablicę `names`, użyj dla niej metody `map!`, czyli „destrukcyjnej" wersji metody `map`:

```
names.map!(&:upcase)
```

← **Sprawdź zawartość wcześniej zamieszczonej ramki!**

Metoda `map!` obiektu `Array` jest zdefiniowana w klasie `Array`, a nie w module `Enumerable`. Ponieważ operacje odwzorowania przeważnie zwracają tablice, niezależnie od tego, jaka może być klasa ich odbiorcy, przeprowadzanie wewnętrznego odwzorowania nie ma sensu, chyba że obiekt jest już tablicą. Trudne byłoby na przykład wyobrażenie sobie, jaką miałoby postać wewnętrzne odwzorowanie zakresu. Metoda `Set#map!` wykonuje jednak wewnętrzne odwzorowanie zbioru z powrotem do postaci niego samego, co ma sens, biorąc pod uwagę to, że pod wieloma względami zbiór przypomina tablicę.

W dalszej kolejności przyjrzymy się klasie, która nie jest wyliczeniowa, czyli klasie `String`. Łańcuchy przypominają trochę zakresy w tym, że zarówno zachowują się jak kolekcje, jak i nie zachowują się jak one. W przypadku zakresów ich właściwości — podobne jak w przypadku kolekcji — są wystarczające do tego, aby klasa zagwarantowała dołączenie modułu `Enumerable`. W przypadku łańcuchów moduł ten nie jest stosowany. Jednak semantyka łańcuchów, gdy są traktowane jako iterowane sekwencje znaków lub bajtów, jest wystarczająco podobna do semantyki obiektów wyliczeniowych, którymi będziemy się tutaj zajmować.

## 10.7. Łańcuchy jako quazi-obiekty wyliczeniowe

Istnieje możliwość iteracji nieprzetworzonych bajtów lub znaków łańcucha za pomocą wygodnych w użyciu metod iteratora, które traktują łańcuch jak kolekcję bajtów, znaków, punktów kodowych lub wierszy. Z każdym z tych czterech sposobów iteracji łańcucha powiązana jest metoda działająca w stylu metody `each`. Aby dokonać iteracji bajtów, użyj metody `each_byte`:

```
str = "abcde"
str.each_byte {|b| p b }
```

Oto dane wyjściowe tego kodu:

```
97
98
99
100
101
```

Jeśli żądany jest każdy znak, a nie jego kod bajtowy, skorzystaj z metody `each_char`:

```
str = "abcde"
str.each_char {|c| p c }
```

Tym razem dane wyjściowe są następujące:

```
"a"
"b"
"c"
"d"
"e"
```

Iteracja przy użyciu punktu kodowego zapewnia kody znaków (liczby całkowite), dokładnie po jednym dla każdego znaku:

```
>> str = "100\u20ac"
=> "100€"
>> str.each_codepoint {|cp| p cp }
49
48
48
8364
```

Porównaj ten przykład z tym, co się stanie w przypadku iteracji tego samego łańcucha bajt po bajcie:

```
>> str.each_byte {|b| p b }
49
48
48
226
130
172
```

Z powodu kodowania liczba bajtów jest większa niż liczba punktów kodowych (lub liczba znaków, która jest równa liczbie punktów kodowych).

I na koniec, aby przetworzyć łańcuch wiersz po wierszu, użyj metody `each_line`:

```
str = "Ten łańcuch\nma trzy\nwiersze"
str.each_line {|l| puts "Następny wiersz: #{l}" }
```

Oto dane wyjściowe dla tego przykładu:

```
Następny wiersz: Ten łańcuch
Następny wiersz: ma trzy
Następny wiersz: wiersze
```

Łańcuch jest dzielony na końcu każdego wiersza, lub dokładniej to ujmując, w każdym wystąpieniu bieżącej wartości zmiennej globalnej `$/`. Jeśli modyfikujesz tę zmienną, zmieniany jest separator, który interpreter języka Ruby traktuje jako identyfikator początku następnego wiersza w łańcuchu:

```
str = "Dawid!Adam!Nowak"
$/ = "!"
str.each_line {|l| puts "Następny wiersz: #{l}" }
```

Pojęcie „wiersza" będzie teraz dla interpretera języka Ruby oparte na znaku `!`:

```
Następny wiersz: Dawid!
Następny wiersz: Adam!
Następny wiersz: Nowak
```

Nawet pomimo tego, że z technicznego punktu widzenia łańcuchy w języku Ruby nie są obiektami wyliczeniowymi (klasa `String` nie dołącza modułu `Enumerable`), w razie potrzeby zapewnia on narzędzia niezbędne do przechodzenia łańcuchów traktowanych jako kolekcje znaków, bajtów, punktów kodowych i/lub wierszy.

Cztery opisane tutaj metody działające w stylu metody `each` tworzą enumerator. Więcej o enumeratorach dowiesz się w podrozdziale 10.9. W tym momencie najważniejsze jest to, że dysponujesz kolejnym zestawem opcji w sytuacji, gdy po prostu wymagasz tablicy złożonej wyłącznie z bajtów, znaków, punktów kodowych lub wierszy. Aby skorzystać z tych opcji, pomiń przedrostek `each_` i do reszty nazwy metody dodaj literę `s`. Oto na przykład sposób umożliwiający uzyskanie tablicy samych bajtów łańcucha:

```
string = "Witaj"
p string.bytes
```

Oto dane wyjściowe:

```
[72, 101, 108, 108, 111]
```

Podobnie możesz postąpić w przypadku metod `chars`, `codepoints` i `lines`.

Używając jeszcze bardziej różnorodnych metod, dla obiektów kolekcji wykonywaliśmy operacje wyszukiwania, transformacji, filtrowania i odpytywania. Nie było jeszcze mowy o jednej operacji, czyli *sortowaniu* kolekcji. Zajmiemy się tym poniżej.

## 10.8. Sortowanie obiektów wyliczeniowych

Jeśli dla używanej klasy chcesz mieć możliwość uporządkowania wielu jej instancji, musisz wykonać następujące działania:

1. Zdefiniuj metodę porównującą dla klasy (<=>).
2. Umieść wiele instancji w kontenerze (prawdopodobnie w tablicy).
3. Przeprowadź sortowanie kontenera.

Kluczową kwestią jest to, że choć możliwość sortowania zapewnia moduł `Enumerable`, klasa nie musi go dołączać. Używane obiekty są umieszczane w obiekcie kontenera, który dołącza moduł `Enumerable`. Jako obiekt wyliczeniowy ten obiekt kontenera dysponuje dwoma metodami sortującymi `sort` i `sort_by`, których możesz użyć do sortowania kolekcji.

W zdecydowanej większości przypadków kontener, w którym umieścisz obiekty do posortowania, będzie tablicą. Czasami będzie to tablica asocjacyjna. W tym przypadku wynikiem będzie tablica (złożona z dwuelementowych tablic zawierających pary klucz-wartość, które są sortowane według klucza lub jakiegoś innego kryterium).

Zwykle nie musisz jawnie tworzyć tablicy elementów przed ich posortowaniem. Częściej sortowana jest kolekcja, która została już wygenerowana automatycznie przez program. Możesz na przykład wykonać operację metody `select` dla kolekcji obiektów i posortować wybrane obiekty. A zatem ręczne umieszczanie list obiektów w nawiasach kwadratowych w celu zaprezentowania przykładów tablic w tym podrozdziale jest trochę nienaturalne. Naszym celem jest jednak skoncentrowanie się bezpośrednio na technikach sortowania i tym się zajmiemy.

Oto prosty przykład sortowania obejmujący tablicę liczb całkowitych:

```
>> [3,2,5,4,1].sort
=> [1, 2, 3, 4, 5]
```

Wykonanie tej operacji jest proste w przypadku liczb, a nawet łańcuchów (sortowanie zapewnia uporządkowanie ich w kolejności alfabetycznej). Tablica, w której umieszczasz takie obiekty, dysponuje mechanizmem sortowania, a liczby całkowite lub łańcuchy potrafią w pewnym stopniu rozpoznać uporządkowanie.

A co będzie, gdy na przykład postanowisz posortować tablicę obiektów `Painting`?

```
>> [pa1, pa2, pa3, pa4, pa5].sort
```

Aby obiekty `Painting` miały informacje wystarczające do uczestniczenia w operacji sortowania, musisz zdefiniować operator „statku kosmicznego" (więcej informacji o nim zamieszczono w punkcie 7.6.2): `Painting#<=>`. Dzięki temu dla każdego takiego obiektu będzie wiadome, co oznacza bycie większym lub mniejszym od innego obiektu `Painting`. Ponadto umożliwi to tablicy sortowanie swojej zawartości. Pamiętaj, że sortowana jest tablica, a nie każdy obiekt `Painting`. Aby jednak można było posortować tablicę, jej elementy muszą mieć możliwość wzajemnego porównywania (nie musisz dołączać modułu `Comparable`, do którego wkrótce powrócimy; wystarczy jedynie metoda <=>).

Załóżmy, że obiekty `Painting` mają zostać posortowane w kolejności rosnących cen i że obiekty mają atrybut `price`. Gdzieś w klasie `Painting` należy umieścić następujący kod:

```
def <=>(other_painting)
    self.price <=> other_painting.price
end
```

Dzięki temu dowolna sortowana tablica obiektów `Painting` zostanie uporządkowana według ceny:

```
price_sorted = [pa1, pa2, pa3, pa4, pa5].sort
```

W języku Ruby dla tych elementów (dla dwóch jednocześnie) stosowany jest test metody <=>, który zapewnia informacje wystarczające do przeprowadzenia kompletnego sortowania.

Bardziej szczegółowe zestawienie kroków operacji sortowania może mieć następującą postać:

1. Przekazywanie obiektom informacji o sposobie wzajemnego porównywania się za pomocą metody <=>.
2. Umieszczenie tych obiektów wewnątrz obiektu wyliczeniowego (prawdopodobnie w tablicy).
3. Zażądanie od tego obiektu posortowania samego siebie. Sortowanie polega na zażądaniu przez obiekt od obiektów wzajemnego porównania się przy użyciu metody <=>.

Jeśli będziesz pamiętać o takim podziale zadań, zrozumiesz, jak przebiega sortowanie, a także jak jest powiązane z modułem `Enumerable`. Co jednak z modułem `Comparable`?

### 10.8.1. W jaki sposób moduł Comparable przydaje się (albo nie) podczas sortowania obiektów wyliczeniowych?

Po raz pierwszy wspomniano o operatorze <=> w kontekście dołączania modułu `Comparable` i umożliwiania mu tworzenia swoich różnych metod (>, < itd.) na bazie tego operatora. Przygotowując jednak obiekty do sortowania ich wewnątrz kontenerów obiektów wyliczeniowych, zdefiniowano jedynie operator <=>. Moduł `Comparable` nie był dołączany.

Całość będzie do siebie pasować, jeśli pomyślisz o tym jak o kilku osobnych technikach warstwowych:

- Jeśli zdefiniujesz metodę <=> dla klasy, jej instancje mogą być umieszczane wewnątrz tablicy lub innego obiektu wyliczeniowego na potrzeby sortowania.
- Jeśli nie zdefiniujesz metody <=>, w dalszym ciągu możesz sortować obiekty po umieszczeniu ich wewnątrz tablicy i udostępnieniu bloku kodu informującego tablicę, jak powinna szeregować dowolne dwa obiekty (jest to omawiane w dalszej kolejności w punkcie 10.8.2).
- Jeśli zdefiniujesz metodę <=>, a ponadto do klasy dołączysz moduł `Comparable`, zapewnisz możliwość sortowania w obrębie tablicy *oraz* wykonywania wszystkich operacji porównywania między dowolnymi dwoma obiektami (>, < itd.). Wspomniano o tym przy omawianiu modułu `Comparable` w rozdziale 9.

Inaczej mówiąc, metoda <=> przydaje się zarówno w przypadku klas, których instancje mają być sortowane, jak i klas z instancjami, które mają być ze sobą porównywane w bardziej szczegółowy sposób z wykorzystaniem pełnego zestawu operatorów porównywania.

Wracamy do sortowania obiektów wyliczeniowych, a w szczególności do jego wariantu, w przypadku którego zamiast metody <=> udostępniany jest blok kodu w celu określenia, jak obiekty powinny być porównywane i szeregowane.

### 10.8.2. Definiowanie logiki kolejności sortowania przy użyciu bloku

W sytuacjach, w których dla obiektów nie zdefiniowano metody <=>, możesz dynamicznie podać blok określający sposób, w jaki obiekty mają zostać posortowane. Jeśli istnieje ta metoda, udostępniając kod, możesz ją przesłonić na potrzeby bieżącej operacji sortowania.

Przyjmijmy na przykład, że zdefiniowano metodę `Painting#<=>` w taki sposób, że tak jak wcześniej dokonuje sortowania według ceny. Chcesz teraz jednak wykonać sortowanie przy użyciu roku. Tego rodzaju sortowanie możesz wymusić za pomocą bloku:

```
year_sort = [pa1, pa2, pa3, pa4, pa5].sort do |a,b|
  a.year <=> b.year
end
```

Blok pobiera dwa argumenty: `a` i `b`. Umożliwia to użycie w kodzie Ruby bloku wymaganą liczbę razy do porównania dwóch obiektów `Painting`. Kod wewnątrz bloku wykonuje za pomocą metody <=> porównanie odpowiednich lat dla dwóch obiektów `Painting`. W przypadku takiego wywołania metody `sort` kod w bloku jest używany zamiast kodu w metodzie <=> klasy `Painting`.

Takiej postaci bloku kodu metody `sort` możesz użyć do obsługi przypadków, w których obiekty są pozbawione metody <=>, a tym samym nie potrafią wzajemnie się porównywać. Taka postać może być też przydatna, gdy sortowane obiekty są różnych klas, a ponadto domyślnie nie mają możliwości wzajemnego porównywania się. Na przykład liczby całkowite i łańcuchy nie mogą być bezpośrednio porównywane: wyrażenie takie jak `"2" <=> 4` powoduje błąd krytyczny. Jeśli jednak najpierw zostanie przeprowadzona konwersja, porównanie będzie możliwe:

```
>> ["2",1,5,"3",4,"6"].sort {|a,b| a.to_i <=> b.to_i }
=> [1, "2", "3", 4, 5, "6"]
```

Elementy posortowanej tablicy wyjściowej są takie same jak elementy tablicy wejściowej. Jest to kombinacja łańcuchów i liczb całkowitych. Są one jednak uporządkowane tak, jakby wszystkie były liczbami całkowitymi. Wewnątrz bloku kodu zarówno łańcuchy, jak i liczby całkowite są normalizowane do postaci liczb całkowitych za pomocą metody `to_i`. Jeśli mowa o mechanizmie sortowania — przeprowadza on sortowanie na bazie serii porównań liczb całkowitych. Mechanizm stosuje następnie kolejność sortowania obecną w oryginalnej tablicy.

A zatem metoda `sort` z blokiem może być pomocna tam, gdzie istniejące metody porównujące nie są w stanie zrealizować działania. Istnieje jeszcze bardziej zwięzły sposób sortowania kolekcji z blokiem kodu. Mowa o metodzie `sort_by`.

### 10.8.3. Zwięzłe sortowanie za pomocą metody sort_by

Podobnie jak metoda `sort` metoda `sort_by` jest metodą instancji modułu `Enumerable`. Główną różnicą jest to, że metoda `sort_by` zawsze pobiera blok, a ponadto wymaga jedynie przekazania jej informacji o tym, jak ma traktować element w kolekcji. Metoda ta stwierdza, że chcesz zrealizowania tego samego działania dla obu elementów każdorazowo, gdy porównuje parę obiektów.

W przypadku zastosowania metody `sort_by` poprzedni przykład sortowania tablicy może mieć następującą postać:

```
>> ["2",1,5,"3",4,"6"].sort_by {|a| a.to_i }  ←——— Lub sort_by(&:to_i)
=> [1, "2", "3", 4, 5, "6"]
```

W obrębie bloku konieczne jest jedynie jednokrotne przekazanie informacji o tym, jakie działanie musi zostać wykonane w celu przygotowania każdego obiektu do operacji sortowania. Nie ma potrzeby wywoływania metody `to_i` dla dwóch obiektów ani jawnego używania metody `<=>`.

Oprócz modułu `Enumerable` — gdy w dalszym ciągu poruszamy się w sferze możliwości wyliczania — język Ruby oferuje klasę o nazwie `Enumerator`. Enumeratory zapewniają językowi Ruby cały wymiar możliwości związanych z modyfikowaniem kolekcji. Przyjrzymy się im teraz dokładnie.

## 10.9. Enumeratory i następny wymiar możliwości wyliczania

Enumeratory są blisko powiązane z iteratorami, ale nie są tym samym. Iterator to metoda przekazująca blokowi kodu jedną lub więcej wartości. Enumerator jest *obiektem*, a nie metodą.

W zasadzie enumerator to prosty obiekt wyliczeniowy. Zawiera metodę `each` oraz wykorzystuje moduł `Enumerable` do definiowania wszystkich zwykłych metod (`select`, `inject`, `map` i innych podobnych) bezpośrednio na bazie swojej metody `each`.

Elementem zaskoczenia jest tu jednak sposób definiowania metody `each` enumeratora.

Enumerator nie jest obiektem kontenera. Nie dysponuje „naturalną” podstawą operacji wykonywanej przez metodę `each`, tak jak ma to miejsce w przypadku tablicy (rozpoczęcie od elementu `0`, przekazanie go, przejście do elementu `1`, przekazanie go itd.). Logika iteracji metody `each` każdego enumeratora musi zostać jawnie określona. Po poinformowaniu enumeratora, jak wykonywać operację metody `each`, przejmuje on inicjatywę, „stwierdzając”, w jaki sposób używać metod `map`, `find`, `take`, `drop` i całej reszty.

Enumerator jest jak mózg w filmie science fiction umieszczony na stole bez żadnego połączenia z ciałem, ale nadal mający możliwość myślenia. Enumerator wymaga jedynie algorytmu metody `each`, aby mógł rozpocząć realizowanie rzeczy, dla których sposób osiągnięcia jest mu już znany. Enumerator może uzyskać takie informacje na jeden z dwóch sposobów. Pierwszym jest wywołanie metody `Enumerator.new` z blokiem kodu zawierającym logikę metody `each`, na której ma bazować enumerator. Drugi

sposób to utworzenie enumeratora na bazie istniejącego obiektu wyliczeniowego (tablica, tablica asocjacyjna itp.) tak, aby metoda `each` enumeratora pobierała jego elementy na potrzeby iteracji z konkretnej metody tego obiektu wyliczeniowego.

Przyjrzymy się najpierw wariantowi tworzenia enumeratorów opartemu na bloku kodu. W pozostałej części omówienia enumeratorów skoncentrujemy się jednak na drugim rozwiązaniu, w przypadku którego enumerator jest „doczepiany" do iteratora innego obiektu (jeśli technikę opartą na bloku uznasz za trudną do zrozumienia, nic się wielkiego nie stanie, gdy na razie pominiesz punkt 10.9.1 i skoncentrujesz się na punkcie 10.9.2). To, jakich technik używasz, a także to, jak je połączysz, ostatecznie będzie zależeć od dokładnych potrzeb w danej sytuacji.

### 10.9.1. Tworzenie enumeratorów z blokiem kodu

Oto prosty przykład tworzenia instancji enumeratora z blokiem kodu:

```
e = Enumerator.new do |y|
  y << 1
  y << 2
  y << 3
end
```

Na początek podstawowa sprawa: czym jest `y`?

`y` to *obiekt przekazujący*, czyli instancja klasy `Enumerator::Yielder`, która jest automatycznie przekazywana blokowi. W obiektach przekazujących zawarty jest opis scenariusza przekazywania, zgodnie z którym ma postępować enumerator. W tym przykładzie określamy, że *gdy enumerator uzyska wywołanie metody* `each`, *oczekuje się od niego, że powinien przekazać wartość 1, a następnie wartość 2 i wartość 3*. Metoda `<<` (jak zwykle na pozycji operatora wrostka) ma za zadanie poinstruowanie obiektu przekazującego o tym, co powinien przekazać (możesz też utworzyć wywołanie `y.yield(1)` itd., choć podobieństwo metody `yield` do słowa kluczowego `yield` może spowodować więcej niejasności niż korzyści). W momencie zażądania iteracji od enumeratora komunikuje się on z obiektem przekazującym i wykonuje następny krok (przekazanie) na podstawie instrukcji zapisanych przez obiekt przekazujący.

Co ma miejsce w przypadku użycia enumeratora `e`? Oto sesja narzędzia irb, w której enumerator poddano testom (przy założeniu, że kod w przykładzie został już wykonany):

```
>> e.to_a                  ←—— Reprezentacja tablicowa przekazanych elementów
=> [1, 2, 3]
>> e.map {|x| x * 10 }     ←—— Odwzorowywanie oparte na metodzie each
=> [10, 20, 30]
>> e.select {|x| x > 1 }   ←—— Wybieranie oparte na metodzie each
=> [2, 3]
>> e.take(2)               ←—— Pobiera pierwsze dwa przekazane elementy
   => [1, 2]
```

Enumerator `e` jest mechanizmem wyliczeniowym. Nie zawiera obiektów. Z enumeratorem powiązany jest oryginalny blok kodu, który przekazuje mu informacje o działaniach, jakie ma wykonać, gdy komunikacja z nim odbywa się na warunkach, jakie rozpoznaje jako pochodzące z modułu `Enumerable`.

Enumerator dokonuje jednorazowej iteracji każdorazowo po wywołaniu metody << (lub metody `yield`) dla obiektu przekazującego. Jeśli wywołania metody << umieścisz wewnątrz pętli lub innego iteratora w obrębie bloku kodu, możesz zastosować niemal dowolną żądaną logikę iteracji. Oto modyfikacja poprzedniego przykładu, w której wewnątrz bloku użyto iteratora:

```
e = Enumerator.new do |y|
  (1..3).each {|i| y << i }
end
```

Biorąc pod uwagę taką definicję, zachowanie enumeratora `e` będzie takie samo jak w poprzednich przykładach. Określono trzykrotne wywołanie metody <<. Oznacza to, że metoda `e.each` dokona trzech iteracji. I tym razem zachowanie enumeratora może być śledzone ostatecznie do miejsca wywołań metody << wewnątrz bloku kodu, za pomocą którego został on zainicjowany.

W szczególności zauważ, że nie ma miejsca przekazywanie z bloku. Oznacza to, że nie używasz następującego kodu:

```
e = Enumerator.new do   <------ Niepoprawnie!
  yield 1               |
  yield 2               | Tego właśnie nie stosujesz!
  yield 3               |
end
```

Zamiast tego obiekt przekazujący (`y` w pierwszych przykładach) jest zapełniany specyfikacjami określającymi, w jaki sposób ma przebiegać iteracja w takiej sytuacji jak wywoływanie metody iteracyjnej dla enumeratora.

Za każdym razem przy wywoływaniu metody iteracyjnej dla enumeratora jednorazowo wykonywany jest blok kodu. Wszelkie zmienne inicjowane w bloku są inicjalizowane jednokrotnie na początku każdego takiego wywołania metody. Sekwencję wykonywania możesz śledzić, dodając szczegółowe komentarze i wywołując wiele metod:

```
e = Enumerator.new do |y|
  puts "Rozpoczynanie wykonywania bloku!"
  (1..3).each {|i| y << i }
  puts "Zakończenie bloku!"
end
p e.to_a
p e.select {|x| x > 2 }
```

Oto dane wyjściowe tego kodu:

```
Rozpoczynanie wykonywania bloku!  <------ Wywołanie metody to_a
Zakończenie bloku!
[1, 2, 3]
Rozpoczynanie wykonywania bloku!  <------ Wywołanie metody select
Zakończenie bloku!
[3]
```

Możesz zauważyć, że blok jest wykonywany jednokrotnie dla każdego iteratora wywołanego dla enumeratora `e`.

Możliwe jest również uwzględnienie innych obiektów w bloku kodu enumeratora. Poniżej zaprezentowano trochę abstrakcyjny przykład, w którym enumerator wykonuje obliczenie obejmujące elementy tablicy, gdy są one trwale usuwane z tablicy.

```
a = [1,2,3,4,5]
e = Enumerator.new do |y|
  total = 0
  until a.empty?
    total += a.pop
    y << total
  end
end
```

Przyjrzyjmy się teraz w narzędziu irb temu, co pozostało z „nieszczęsnej" tablicy `a`:

```
>> e.take(2)
=> [5, 9]
>> a
=> [1, 2, 3]
>> e.to_a
=> [3, 5, 6]
>> a
=> []
```

Operacja metody `take` generuje tablicę wynikową z dwoma elementami (wartość zmiennej `total` dla dwóch kolejnych iteracji), a ponadto w tablicy `a` pozostawia trzy elementy. Wywołanie w tym przypadku metody `to_a` dla tablicy `a` powoduje ponowne wykonanie oryginalnego bloku kodu, ponieważ wywołanie metody `to_a` nie jest częścią tej samej iteracji co wywołanie metody `take`. A zatem zmienna `total` ponownie zaczyna się od pozycji `0`, a pętla `until` jest wykonywana, czego wynikiem są przekazane trzy wartości i pusta tablica `a`.

Nie w porządku jest „robienie zasadzki" na osobny obiekt przez usuwanie jego elementów jako efektu ubocznego wywoływania enumeratora. Przykład prezentuje jednak mechanizm, a także zapewnia sensowne przejście do drugiej części zagadnienia tworzenia enumeratorów, których metody `each` są powiązane z konkretnymi metodami istniejących obiektów wyliczeniowych.

### 10.9.2. Powiązanie enumeratorów z innymi obiektami

Innym sposobem zapewnienia enumeratorowi logiki metody `each` jest „podpięcie" go do innego obiektu, a dokładniej rzecz biorąc, do iteratora (często jest to metoda `each`, ale potencjalnie może to być dowolna metoda przekazująca co najmniej jedną wartość) innego obiektu. Zapewnia to enumeratorowi bazę dla jego własnej iteracji: gdy musi coś przekazać, otrzymuje niezbędną wartość przez wyzwolenie za pośrednictwem wyznaczonej metody następnego przekazania z obiektu, z którym enumerator został powiązany. A zatem enumerator pełni po części rolę „pełnomocnika", a po części „pasożyta", definiując własną metodę `each` z uwzględnieniem iteracji innego obiektu.

W przypadku tego rozwiązania enumerator możesz utworzyć przez wywołanie metody `enum_for` (znana też pod nazwą `to_enum`) dla obiektu, z którego enumerator ma uzyskiwać swoje iteracje. Jako pierwszy argument podajesz nazwę metody, do której

enumerator dołączy swoją metodę `each`. Choć domyślnie argument ma wartość `:each`, częstą sytuacją jest dołączanie enumeratora do innej metody, tak jak w następującym przykładzie:

```
names = %w{ Dawid Nowak Adam Kowalski }
e = names.enum_for(:select)
```

Określenie `:select` jako argumentu oznacza, że dany enumerator ma zostać powiązany z metodą `select` tablicy `array`. A zatem metoda `each` enumeratora będzie pełnić coś w rodzaju interfejsu metody `select` tablicy:

```
e.each {|n| n.include?('a') } ←——— Dane wyjściowe: ["Dawid", "Nowak", "Kowalski"]
```

Metodzie `enum_for` można też udostępnić dodatkowe argumenty. Wszystkie one są przekazywane metodzie, do której dołączany jest enumerator. Poniżej pokazano na przykład, jak utworzyć enumerator dla metody `inject` tak, aby w momencie wywołania jej w celu przekazania wartości metodzie `each` enumeratora była używana wartość początkowa argumentu `"Imiona: "`.

```
>> e = names.enum_for(:inject, "Imiona: ")
=> #<Enumerator: ["Dawid", "Nowak", "Adam", "Kowalski"]:inject("Imiona: ")>
>> e.each {|string, name| string << "#{name}..." }
=> "Imiona: Dawid...Nowak...Adam...Kowalski..."
```

Bądź jednak ostrożny! Choć do tego łańcucha początkowego `"Imiona: "` zostało dodanych kilka imion, nadal pozostaje on wewnątrz enumeratora. Oznacza to, że jeśli ponownie wykonasz taką samą operację metody `inject`, spowoduje ona dodanie imienia do tego samego łańcucha (w celu dopasowania do szerokości strony wiersz danych wyjściowych dla poniższego wiersza kodu podzielono na dwa wiersze):

```
>> e.each {|string, name| string << "#{name}..." }
=> "Imiona: Dawid...Nowak...Adam...Kowalski...
Dawid...Nowak...Adam...Kowalski..."
```

Argumenty udostępniane enumeratorowi podczas jego tworzenia w celu zapewnienia argumentów jego metodzie pośredniczącej są argumentami (obiektami), których enumerator będzie ciągle używać. Z tego powodu uważaj na efekty uboczne. W tym konkretnym przypadku możesz uniknąć takiego efektu przez dodanie łańcuchów (`string + "#{name}..."`), a nie dołączenie do łańcucha za pomocą metody `<<`, ponieważ operacja dodawania powoduje utworzenie nowego obiektu łańcucha. Niemniej jednak taka przestroga zwykle jest przydatna.

> **UWAGA** Metodę `Enumerator.new(`*`obiekt, nazwa_metody, arg1, arg2...`*`)` możesz wywołać jako odpowiednik metody `obj.enum_for(`*`nazwa_metody, arg1, arg2...`*`)`. Użycie takiej postaci `Enumerator.new` nie jest jednak zalecane. Z metody `enum_for` skorzystaj w przypadku dołączania metody, a metody `Enumerator.new` użyj na potrzeby opisanego w punkcie 10.9.1 scenariusza bazującego na bloku kodu.

Wiesz już, jak tworzyć enumeratory obu rodzajów. W przypadku pierwszego z nich informacje o sposobie iteracji są przekazywane w bloku kodu. Z kolei w przypadku

drugiego rodzaju informacje te pochodzą z innego obiektu. Enumeratory są też tworzone niejawnie, gdy tworzone są wywołania bez bloku kodu dotyczące określonych metod iteratora.

### 10.9.3. Niejawne tworzenie enumeratorów za pomocą wywołań iteratora bez użycia bloku kodu

Z definicji iterator jest metodą przekazującą blokowi jedną lub więcej wartości. Co jednak będzie, gdy blok nie jest dostępny?

Odpowiedzią na powyższe pytanie jest to, że większość wbudowanych iteratorów zwraca enumerator, gdy są one wywoływane bez bloku. Poniżej zamieszczono przykład z klasy `String`, czyli metodę `each_byte` (więcej informacji zawarto w podrozdziale 10.7). Najpierw należy zaprezentować klasyczne użycie metody iteratora bez enumeratora, lecz z blokiem:

```
>> str = "Witaj"
=> "Witaj"
>> str.each_byte {|b| puts b }
72
101
108
108
111
=> "Witaj"
```

Metoda `each_byte` dokonuje iteracji bajtów w łańcuchu i zwraca swojego odbiorcę (łańcuch). Jeśli jednak metodę tę wywołasz bez bloku, uzyskasz enumerator:

```
>> str.each_byte
=> #<Enumerator: "Witaj":each_byte>
```

Otrzymywany enumerator odpowiada temu, co zostałoby uzyskane w przypadku użycia następującego wiersza kodu:

```
>> str.enum_for(:each_byte)
```

Stwierdzisz, że wiele metod modułu `Enumerable` zwraca enumeratory po wywołaniu tych metod bez bloku (dotyczy to również metod `each`, `map`, `select`, `inject` oraz innych). Podstawowym przypadkiem użycia tych automatycznie zwracanych enumeratorów jest *tworzenia łańcucha*, czyli wywoływanie następnej metody bezpośrednio w enumeratorze. Tworzeniu łańcucha przyjrzymy się w ramach omawiania semantyki enumeratorów w następnym podrozdziale.

## 10.10. Semantyka enumeratorów i ich zastosowanie

Gdy już wiesz, jak enumeratory są podłączane i tworzone, warto się dowiedzieć, w jaki sposób są one wykorzystywane, a także dlaczego są przydatne.

Być może najtrudniejszą kwestią w przypadku enumeratorów, z powodu dużych trudności związanych z wizualną interpretacją, jest objaśnienie, jakie działania mają miejsce przy wywołaniu metody `each`. Zaczniemy od tego, a później omówimy funkcjonalność enumeratorów, a w szczególności możliwości ochrony przez nie obiektów

przed zmianą. Wyjaśnimy też, jak za pomocą enumeratora można przeprowadzać szczegółowe, kontrolowane iteracje. W dalszej kolejności przyjrzymy się temu, jak ogólnie enumeratory dopasowują się do łańcuchów metod, a ponadto zaprezentujemy kilka ważnych konkretnych przypadków ich użycia.

### *10.10.1. Sposób użycia metody each enumeratora*

Metoda `each` enumeratora jest podłączana do metody w innym obiekcie, która może być inną metodą niż `each`. Jeśli metody `each` enumeratora użyjesz bezpośrednio, będzie ona zachowywać się tak jak inna metoda, również jeśli chodzi o jej wartość zwracaną.

Następujący kod może wygenerować dziwnie wyglądające wyniki, w przypadku których wywołania metody `each` zwracają kolekcje poddane filtrowaniu, sortowaniu lub odwzorowywaniu:

```
>> array = %w{ kot pies królik }
=> ["kot", "pies", "królik"]
>> e = array.map
=> #<Enumerator: ["kot", "pies", "królik"]:map>
>> e.each {|animal| animal.capitalize }      ←——— Zwraca odwzorowanie
=> ["Kot", "Pies", "Królik"]
```

Nie ma w tym nic tajemniczego. Enumerator nie jest tym samym obiektem co tablica. Enumerator samodzielnie określa znaczenie metody `each`. Niemniej jednak ogólnym efektem połączenia enumeratora z metodą `map` tablicy jest uzyskanie operacji metody `each` z odwzorowaniem tablicy jako jej wartością zwracaną. Jak już pokazano, zwykła iteracja metody `each` tablicy istnieje głównie ze względu na jej efekty uboczne, a ponadto zwraca swojego odbiorcę (tablica). Metoda `each` enumeratora jest jednak czymś w rodzaju kanału prowadzącego do metody, z której pobiera swoje wartości, a ponadto zachowuje się tak samo wobec wartości zwracanej.

Inna właściwość enumeratorów, o której trzeba wiedzieć, dotyczy tego, że wykonują one operację będącą czymś w rodzaju *odwrotności przesłaniania* metod w module `Enumerable`.

#### ZJAWISKO ODWROTNOŚCI PRZESŁANIANIA

Jeśli klasa definiuje metodę `each` i dołącza moduł `Enumerable`, jej instancje automatycznie uzyskują metody `map`, `select`, `inject` oraz całą resztę metod tego modułu. Wszystkie te metody są definiowane na warunkach metody `each`.

Czasami jednak dana klasa dokonała już przesłonięcia metody w wersji modułu `Enumerable` własną metodą. Dobrym przykładem jest metoda `Hash#select`. Standardowa metoda `select` z modułu `Enumerable` zawsze zwraca tablicę, niezależnie od tego, jaka może być klasa obiektu, który używa tej metody. Z kolei operacja metody `select` wykonana dla tablicy asocjacyjnej zwraca tablicę asocjacyjną:

```
>> h = { "kot" => "feline", "pies" => "canine", "krowa" => "bovine" }
=> {"kot"=>"feline", "pies"=>"canine", "krowa"=>"bovine"}
>> h.select {|key,value| key =~ /k/ }
=> {"kot"=>"feline", "krowa"=>"bovine"}
```

Na razie wygląda to dobrze (i nie ma nic nowego). Jeśli podłączymy enumerator do metody `select`, uzyskamy metodę `each` działającą jak następująca metoda:

```
>> e = h.enum_for(:select)
=> #<Enumerator: {"kot"=>"feline", "pies"=>"canine", "krowa"=>"bovine"}:select>
>> e.each {|key,value| key =~ /k/ }
=> {"kot"=>"feline", "krowa"=>"bovine"}
```

Co będzie jednak w sytuacji, gdy enumerator nie zostanie podłączony do metody `select` tablicy asocjacyjnej, lecz do jej metody `each`? Enumerator można uzyskać za pomocą metody `to_enum`, zezwalając na to, aby docelową metodą domyślną była metoda `each`:

```
>> e = h.to_enum
=> #<Enumerator: {"kot"=>"feline", "pies"=>"canine", "krowa"=>"bovine"}:each>
```

Metoda `Hash#each` wywołana z blokiem zwraca tablicę asocjacyjną. To samo dotyczy metody `each` enumeratora, ponieważ jest to tylko interfejs metody `each` tablicy asocjacyjnej. Bloki w tych przykładach są puste, gdyż interesują nas wyłącznie wartości zwracane:

```
>> h.each { }
=> {"kot"=>"feline", "pies"=>"canine", "krowa"=>"bovine"}
>> e.each { }
=> {"kot"=>"feline", "pies"=>"canine", "krowa"=>"bovine"}
```

Jak dotąd wygląda na to, że metoda `each` enumeratora zastępuje metodę `each` tablicy asocjacyjnej. Co będzie jednak, gdy do wykonania operacji wybierania zostanie użyta metoda `each` enumeratora?

```
>> e.select {|key,value| key =~ /k/ }
=> [["kot", "feline"], ["krowa", "bovine"]]
```

Jak widać, odpowiedź jest taka, że jako wynik uzyskujemy tablicę, a nie tablicę asocjacyjną.

Dlaczego? Jeśli metoda `e.each` jest „podpięta" do metody `h.each`, w jaki sposób wartość zwracana metody `e.select` zostanie „odpięta" od wartości zwracanej metody `h.each`?

Kluczem jest to, że wywołanie metody `select` w ostatnim przykładzie jest wywołaniem metody `select` *enumeratora*, a nie tablicy asocjacyjnej. Ponadto metoda `select` enumeratora bazuje bezpośrednio na metodzie `each` enumeratora. Okazuje się, że metoda `select` enumeratora jest metodą `Enumerable#select`, która zawsze zwraca tablicę. To, że metoda `Hash#select` nie zwraca tablicy, nie ma żadnego znaczenia dla enumeratora.

W tym sensie enumerator zapewnia tablicy asocjacyjnej możliwość wyliczania nawet pomimo tego, że tablica ta jest już obiektem wyliczeniowym. Enumerator powoduje również odwracanie przesłonięcia metody `Enumerable#select`. Metoda `select` zapewniana przez enumerator jest metodą `Enumerable#select` nawet wtedy, gdy nie była nią metoda `select` tablicy asocjacyjnej (z technicznego punktu widzenia nie jest to odwracanie przesłonięcia, ale powoduje wrażenie, że enumerator przesłania logikę wybierania oryginalnej tablicy asocjacyjnej).

Wniosek z tego taki, że ważne jest pamiętanie o tym, iż enumerator to obiekt różniący się od kolekcji, z której uzyskuje swoje iterowane obiekty. Choć ta różnica między obiektami może być przyczyną dziwnych rezultatów, takich jak przekierowywanie metody `select` za pośrednictwem modułu `Enumerable`, ostatecznie zapewnia korzyści przynajmniej na jeden istotny sposób. Mowa o uzyskiwaniu dostępu do kolekcji za pośrednictwem enumeratora, a nie samej kolekcji, co chroni obiekt kolekcji przed modyfikacją.

### *10.10.2. Ochrona obiektów za pomocą enumeratorów*

Rozważ metodę oczekującą na przykład, że jej argumentem będzie tablica (zgadza się, zwracanie uwagi na klasę obiektu nie jest zbytnio w stylu języka Ruby, ale zauważysz, że w tym miejscu nie jest to główna kwestia).

```
def give_me_an_array(array)
```

Jeśli metodzie tej przekażesz obiekt tablicy, może go ona zmodyfikować:

```
array << "nowy element"
```

Aby ochronić oryginalną tablicę przed modyfikacją, możesz ją zduplikować i przekazać duplikat albo zamiast niego enumerator:

```
give_me_an_array(array.to_enum)
```

Choć enumerator bez żadnych problemów zezwoli na iteracje tablicy, nie uwzględni zmian (próba wywołania metody << dla tablicy spowoduje wygenerowanie błędu krytycznego). Inaczej mówiąc, enumerator może pełnić rolę czegoś w rodzaju bramy do obiektu kolekcji, która zezwala na iterację i sprawdzanie elementów, ale nie umożliwia wykonywania operacji destrukcyjnych.

Kod talii kart z punktu 10.5.4 zapewnia możliwość zapewnienia pewnej ochrony obiektów. W tym kodzie klasa `Deck` zawiera atrybut odczytujący `cards`. Podczas tworzenia talii kart jej zmienna instancji `@cards` jest inicjowana przy użyciu tablicy zawierającej wszystkie karty. Pojawia się tutaj niepewność: co będzie, jeśli ktoś uzyska dostęp do tablicy zmiennej `@cards` za pośrednictwem atrybutu odczytującego `cards` i zmieni go?

```
deck = PlayingCard::Deck.new
deck.cards << "JOKER!!"
```

W idealnej sytuacji chcielibyśmy mieć możliwość odczytu z tablicy zmiennej `@cards`, lecz bez modyfikowania jej (można ją zablokować za pomocą metody `freeze`, która zapobiega dalszym zmianom w obiektach, ale konieczna będzie modyfikacja talii wewnątrz klasy `Deck`, gdy z talii zostaną rozdane karty). Enumeratory zapewniają rozwiązanie. Zamiast używać atrybutu odczytującego, spowodujmy zwrócenie enumeratora przez metodę `cards`:

```
class PlayingCard
  SUITS = %w{ trefle kara kiery piki }
  RANKS = %w{ 2 3 4 5 6 7 8 9 10 J Q K A }
  class Deck
    def cards
      @cards.to_enum
```

```
    end
    def initialize(n=1)
      @cards = []
      SUITS.cycle(n) do |s|
        RANKS.cycle(1) do |r|
          @cards << "#{r} of #{s}"
        end
      end
    end
  end
end
```

Zdeterminowana osoba nadal może uzyskać dostęp do tablicy zmiennej `@cards` i wprowadzić w niej zmiany. Enumerator zapewnia jednak znaczny poziom ochrony:

```
deck = PlayingCard::Deck.new
deck.cards << "Joker!!" ←——— NoMethodError: undefined method '<<' for #<Enumerator:0x000001020643b8>
```

Oczywiście jeśli kod wywołujący ma mieć możliwość adresowania obiektów kart jako tablicy, zwracanie enumeratora może przynieść efekt przeciwny do zamierzonego (przynajmniej jedna inna technika chroni obiekty w takich okolicznościach, czyli instrukcja `return @cards.dup`). Jeśli jednak enumerator okaże się odpowiedni, zapewniane przez niego możliwości ochrony mogą być wystarczające.

Ponieważ enumeratory są obiektami, mają stan. Co więcej, używają one stanu do śledzenia własnego postępu, dlatego możesz zatrzymywać i rozpoczynać ich iteracje. Przyjrzymy się teraz technikom kontrolowania enumeratorów w ten sposób.

### 10.10.3. Rozbudowana iteracja z wykorzystaniem enumeratorów

Enumeratory zachowują stan: śledzą one, na jakim znajdują się etapie wyliczania. Z tych informacji korzysta bezpośrednio kilka metod. Przeanalizuj następujący przykład:

```
names = %w{ Dawid Nowak }
e = names.to_enum
puts e.next
puts e.next
e.rewind
puts e.next
```

Oto dane wyjściowe tych poleceń:

```
Dawid
Nowak
Dawid
```

Enumerator umożliwia przeprowadzanie wyliczania tablicy w zwolnionym tempie, jak to można ująć, zatrzymując i ponawiając proces na żądanie. Pod tym względem przypomina to jeden z tych stołów edycyjnych, na których montażysta filmu składa go ręcznie. W przeciwieństwie do projektora, który po włączeniu nie wymaga dalszej obsługi, stół edycyjny pozwala wpływać na postać odtwarzanego filmu.

Kwestia ta „rzuca też światło” na różnicę między enumeratorem i iteratorem. Enumerator to obiekt, dlatego może utrzymywać stan. Zapamiętuje aktualny etap wyliczania. Iterator jest metodą. Gdy go wywołujesz, wywołanie jest niepodzielne. Po wystąpieniu całego wywołania dobiega ono końca. Dzięki blokom kodu w wywołaniach

metod języka Ruby pojawia się oczywiście określona, przydatna złożoność: metoda może wywołać blok, a ponadto mogą być podejmowane decyzje wpływające na wynik. Nadal jest to jednak metoda. Iterator nie ma stanu. Enumerator jest obiektem wyliczeniowym.

Co interesujące, enumeratora możesz użyć jako obiektu, który nie jest wyliczeniowy. Konieczne jest jedynie, aby obiekt zawierał metodę przekazującą coś, co pozwala enumeratorowi na zastosowanie tej metody jako bazy dla swojej własnej metody `each`. W rezultacie obiekt niewyliczeniowy staje się wyliczeniowym.

### 10.10.4. Zapewnianie możliwości wyliczania za pomocą enumeratora

Enumerator może zapewnić możliwość wyliczania obiektom, które są jej pozbawione. Jest to kwestia powiązania: jeśli do dowolnego iteratora podłączysz metodę `each` enumeratora, możesz go użyć do wykonywania operacji wyliczania dla obiektu, który jest w posiadaniu iteratora, niezależnie od tego, czy ten obiekt identyfikuje się jako wyliczeniowy, czy nie.

Podłączenie enumeratora do metody `String#bytes` skutkuje zapewnieniem obiektowi (łańcuchowi) możliwości wyliczania, której jest on pozbawiony w tym sensie, że klasa `String` nie dołącza modułu `Enumerable`. Ten sam efekt możesz osiągnąć w przypadku własnych klas. Przyjrzyj się następującej klasie, która nie dołącza modułu `Enumerable`, ale zawiera jedną metodę iteratora:

```
module Music
  class Scale
    NOTES = %w{ c c# d d# e f f# g a a# b }
    def play
      NOTES.each {|note| yield note }
    end
  end
end
```

Dla danej klasy możliwe jest dokonanie iteracji dla nut skali:

```
scale = Music::Scale.new
scale.play {|note| puts "Następna nuta to #{note}" }
```

Oto wynik:

```
Następna nuta to c
Następna nuta to c#
Następna nuta to d
```

I tak dalej. Z technicznego punktu widzenia obiekt skali nie jest obiektem wyliczeniowym. Standardowe metody modułu `Enumerable` nie działają, ponieważ klasa `Music::Scale` nie dołącza tego modułu i nie definiuje metody `each`:

```
scale.map {|note| note.upcase }
```

Wynik jest następujący:

```
NoMethodError: unknown method `map' for #<Music::Scale:0x3b0aec>
```

Jeśli w praktyce obiekty skali miałyby być w pełni obiektami wyliczeniowymi, prawie na pewno zostałby dołączony moduł `Enumerable` i zmieniona nazwa metody z `play` na `each`. Możliwe jest też jednak zapewnienie obiektowi skali możliwości wyliczania przez podłączenie go do enumeratora.

Oto sposób tworzenia enumeratora dla obiektu skali powiązanego z metodą `play`:

```
enum = scale.enum_for(:play) ←——— Lub scale.to_enum(:play)
```

Enumerator `enum` zawiera metodę `each`, która wykonuje taką samą iterację co metoda `play` obiektu skali. Co więcej, w przeciwieństwie do obiektu skali enumerator *jest* obiektem wyliczeniowym, który zawiera metody `map`, `select` i `inject` oraz wszystkie standardowe metody z modułu `Enumerable`. Jeśli użyjesz enumeratora, zostaną wykonane operacje wyliczeniowe dla obiektu, który zasadniczo nie jest wyliczeniowy:

```
p enum.map {|note| note.upcase }
p enum.select {|note| note.include?('f') }
```

Oto dane wyjściowe pierwszego wiersza:

```
["C", "C#", "D", "D#", "E", "F", "F#", "G", "A", "A#", "B"]
```

Dane wyjściowe drugiego wiersza są następujące:

```
["f", "f#"]
```

A zatem enumerator dołącza się do określonej metody konkretnego obiektu, a ponadto używa tej metody jako metody podstawowej (`each`) dla całego zestawu narzędzi wyliczeniowych.

Dołączanie enumeratora do obiektu niewyliczeniowego, takiego jak obiekt skali, stanowi dobre ćwiczenie, ponieważ ilustruje ono bardzo wyraźnie różnicę między oryginalnym obiektem i enumeratorem. Jednak w zdecydowanej większości przypadków obiekty, dla których tworzone są enumeratory, same są obiektami wyliczeniowymi, takimi jak tablice, tablice asocjacyjne itd. W większości dalszych przykładów uwzględniono obiekty wyliczeniowe (z wyjątkiem łańcuchów). Oprócz poznania sfery najczęstszych zastosowań umożliwi to nam szersze przyjrzenie się możliwym korzyściom wynikającym z użycia enumeratorów.

Pamiętaj cały czas o wniosku sformułowanym przy prezentowaniu przykładu z obiektem `Music::Scale` i jego enumeratorem: enumerator to obiekt wyliczeniowy, którego metoda `each` działa jak coś w rodzaju „syfonu" pobierającego wartości z iteratora zdefiniowanego w innym obiekcie.

Omówienie enumeratorów zakończymy prezentacją technik, które obejmują tworzenie łańcucha enumeratorów i wywołania metod.

## 10.11. Tworzenie łańcucha metod enumeratora

Tworzenie łańcucha metod to powszechna technika w języku Ruby. Po części wynika to z dużej łatwości jej stosowania. Chcesz wyświetlić listę oddzielonych przecinkami imion złożonych z dużych liter i zaczynających się od litery z przedziału od `A` do `N`? Po prostu połącz kilka metod w ramach jednego łańcucha:

```
puts names.select {|n| n[0] < 'M' }.map(&:upcase).join(", ")
```

**Dane wyjściowe (dla poprzedniej listy imion): DAWID, NOWAK**

Styl przetwarzania danych od lewej do prawej strony przypominający przenośnik taśmowy zapewnia duże możliwości, a ponadto w dużej mierze prostotę. Kryje się jednak za tym koszt w postaci tworzenia obiektów pośrednich. Tworzenie łańcucha metod powoduje zwykle definiowanie nowego obiektu dla każdego łącza w łańcuchu. Założenie w poprzednim kodzie, że `names` jest tablicą łańcuchów, kończy się utworzeniem przez interpreter języka Ruby kolejnych dwóch tablic (jednej w postaci danych wyjściowych metody `select`, a drugiej zwróconej przez metodę `map`) i łańcucha (generowany przez metodę `join`).

Enumeratory nie rozwiązują wszystkich problemów związanych z tworzeniem łańcucha metod. W niektórych przypadkach zmniejszają one jednak problem w postaci tworzenia obiektów pośrednich. Tworzenie łańcucha oparte na enumeratorach zawiera w sobie semantykę, która zasługuje na poznanie.

### *10.11.1. Zmniejszanie liczby obiektów pośrednich*

Pamiętaj, że wiele metod z modułu `Enumerable` zwraca enumerator w przypadku wywołania ich bez bloku. W większości takich sytuacji nie ma powodu, aby w ramach łańcucha dołączać enumerator bezpośrednio do innej metody. Na przykład wywołanie `names.each.inject` równie dobrze może mieć postać `names.inject`. Podobnie jest w przypadku wywołania `names.map.select`, które nie zapewnia niczego więcej niż wywołanie `names.select`. Enumerator metody `map` nie ma żadnych informacji o tym, do jakiej funkcji ma zostać wykonane odwzorowanie. A zatem enumerator nie może zrealizować niczego więcej poza przekazaniem oryginalnej tablicy wartości w dół łańcucha metod.

Rozważ jednak wywołanie `names.each_slice(2)`. Enumerator generowany przez to wyrażenie zawiera w sobie wartościowe informacje o tym, że oczekiwane jest utworzenie porcji tablicy `names` złożonych z dwóch elementów. Umieszczenie tej metody wewnątrz łańcucha metod powoduje następujący efekt:

```
>> names = %w{ Dawid Nowak Adam Kowalski }
=> ["Dawid", "Nowak", "Adam", "Kowalski"]
>> names.each_slice(2).map do |first, last|
     "Imię: #{first}, nazwisko: #{last}\n"
   end
=> ["Imię: Dawid, nazwisko: Nowak\n",
    "Imię: Adam, nazwisko: Kowalski\n"]
```

Z powodu enumeratora metody `each_slice(2)` blok kodu dołączony do operacji odwzorowania uzyskuje po dwa na raz elementy tablicy imion. Enumerator może działać w sposób „leniwy": zamiast tworzyć w pamięci całą tablicę dwuelementowych porcji, może on generować porcje w momencie, gdy będą wymagane przez operację odwzorowania.

Metody modułu `Enumerable` pobierające argumenty i zwracające enumeratory, takie jak `each_slice`, są kandydatami do przeprowadzenia tego rodzaju kompresji lub optymalizacji. Jeśli nawet metoda obiektu wyliczeniowego nie zwraca enumeratora,

**Umiejętność posługiwania się enumeratorami**

Konsekwencją sposobu działania enumeratorów oraz automatycznego zwracania ich z wywołań iteratora bez bloku jest to, że do poprawnego czytania kodu z enumeratorem wymagane jest trochę praktyki. Przyjrzyj się następującemu fragmentowi kodu, który zwraca tablicę liczb całkowitych:

```
string = "Dowolny łańcuch"
string.each_byte.map {|b| b + 1 }
```

Prawdopodobnie nie jest to przydatna logika biznesowa, ale rzecz w tym, że bardzo przypomina wywołanie `string.each_byte`, które zwraca tablicę. Obecność metody `map` jako następnej operacji, choć nie jest ostatecznym dowodem występowania tablicy, z pewnością wskazuje na obecność kolekcji po lewej stronie tej metody.

Spójrzmy na to z innej perspektywy. W przypadku określania na podstawie obecności możesz oczekiwać, że po odłączeniu całego wywołania metody `map` pozostanie kolekcja.

Okazuje się, że wywołanie `string.each_byte` zwraca enumerator. Kluczem jest to, że enumerator *to* kolekcja. Jest to obiekt wyliczeniowy w takim samym stopniu jak tablica lub tablica asocjacyjna. Oswojenie się z tym faktem może wymagać trochę czasu.

możesz go dla niej utworzyć, uwzględniając argument, aby został zapamiętany przez enumerator. W punkcie 10.9.2 zaprezentowano już przykład tej techniki, lecz z trochę innej perspektywy:

```
e = names.enum_for(:inject, "Imiona: ")
```

Enumerator zapamiętuje nie tylko to, że został dołączony do metody `inject` tablicy `names`, ale też to, że reprezentuje wywołanie tej metody z argumentem `"Imiona"`.

Oprócz ogólnej praktyki dołączania enumeratorów do łańcuchów metod istnieje specjalizowana metoda `with_index` (jedna z kilku metod, jakie klasa `Enumerator` implementuje niezależnie od metod modułu `Enumerable`), która w znacznym stopniu zwiększa wartość operacji wyliczania.

### 10.11.2. Indeksowanie obiektów wyliczeniowych za pomocą metody with_index

W czasach, gdy programiści używający języka Ruby korzystali z metody `each_with_index`, część z nas dążyła do stworzenia odpowiadającej jej metody `map_with_index`. Choć nigdy do tego nie doszło, ostatecznie uzyskaliśmy nawet coś lepszego. Enumeratory dysponują metodą `with_index`, która dodaje do dowolnej operacji wyliczania indeksowanie liczbowe jako drugi parametr bloku. Oto przykład użycia metody `with_index` do odwzorowania liter na liczby:

```
('a'..'z').map.with_index {|letter,i| [letter, i] }
```

**Dane wyjściowe: [["a", 0], ["b", 1] itd.]**

Zauważ, że jest to wywołanie `map.with_index` (dwie metody tworzące łańcuch), a nie `map_with_index` (złożona nazwa metody). Metoda `with_index` może być w ramach łańcucha łączona z dowolnym enumeratorem. Czy pamiętasz skalę muzyczną z punktu 10.10.4? Załóżmy, że dla metody `play` utworzono enumerator:

```
def play
  NOTES.to_enum
end
```

Oryginalny przykład przetwarzania obiektów nut tak zmodyfikowano, że obecnie pominięto w nim tworzenie enumeratora pośredniego:

```
scale.play {|note| puts "Następna nuta: #{note}" }
```

Następujący wiersz kodu również zadziała:

**Podaj 1 jako pierwszą wartość indeksu**

```
scale.play.with_index(1) {|note,i| puts "Nuta #{i}: #{note}" }
```

Oto dane wyjściowe w postaci numerowanej listy nut:

```
Nuta 1: c
Nuta 2: c#
Nuta 3: d
# itd.
```

A zatem metoda `with_index` uogólnia to, co w przeciwnym razie byłoby ograniczoną funkcjonalnością.

Przyjrzymy się jeszcze jednemu przykładowi tworzenia łańcucha metod enumeratora, w którym w odpowiedni sposób zebrano kilka technik iteracji i technik opartych na enumeratorach, a także wprowadzono kilka nowych metod (możesz uznać je za przydatne).

### 10.11.3. Operacje alternatywy wykluczającej na łańcuchach z wykorzystaniem enumeratorów

Wykonywanie operacji alternatywy wykluczającej (lub *XOR*) dla łańcucha oznacza użycie w ramach tej operacji określonej wartości dla każdego bajta. Jest to operacja bitowa: każdy bajt reprezentowany jest przez liczbę całkowitą, a wynikiem operacji XOR jest przetworzenie tej liczby za pomocą innej liczby.

Na przykład łańcuchem jest łańcuch `"a"`; zawiera on jeden bajt z wartością `97`. Reprezentacja binarna tej wartości to `1100001`. Załóżmy, że ma dla niej zostać wykonana operacja XOR ze znakiem #, który ma wartość ASCII wynoszącą `35` lub `100011` w zapisie binarnym. Gdy potraktuje się to w sposób czysto numeryczny, a nie w postaci łańcuchów, zostanie wykonana operacja `97 ^ 35` lub `1100001 ^ 100011` w wariancie binarnym.

Operacja XOR generuje wynik, który w przypadku reprezentacji binarnej (czyli dotyczącej bitów) zawiera liczbę `1`, gdy dowolna z liczb źródłowych, *lecz nie obie*, zawierała liczbę `1`. Z kolei wynik uwzględnia liczbę `0` wtedy, gdy obie liczby źródłowe zawierają taką samą liczbę (`0` lub `1`). W przypadku dwóch przykładowych liczb operacja XOR generuje wartość `1000010` lub `66`.

Cechą rozpoznawczą operacji bitowych XOR jest to, że w przypadku dwukrotnego wykonania tej samej operacji uzyskujesz pierwotną wartość. Inaczej mówiąc: `(a ^ b) ^ b == a`. A zatem jeśli dla liczby `66` zostanie wykonana operacja XOR z wykorzystaniem liczby `35`, zostanie uzyskana liczba `97`. Takie działanie operacji XOR w przypadku łańcuchów sprawia, że jest to przydatna technika kodująca, zwłaszcza wtedy, gdy za pomocą drugiego łańcucha wykonujesz operację XOR dla długiego łańcucha bajt po bajcie. Załóżmy, że łańcuch ma postać `"To jest łańcuch."`. Jeśli wykonasz dla niego operację XOR znak po znaku z wykorzystaniem na przykład łańcucha `#%.3u`, powtarzając w razie potrzeby ten łańcuch w celu osiągnięcia długości oryginalnego łańcucha,

uzyskasz raczej zniechęcający wynik `wMG@UJV\x0ERUPQ\\Z\eD\v`. Jeśli wykonasz operację XOR dla tego „monstrum” ponownie z użyciem łańcucha `#%.3u`, w rezultacie otrzymasz łańcuch `"To jest łańcuch."`.

Utwórzmy metodę, która zrealizuje to działanie. Metoda zostanie dodana do klasy `String`. Niekoniecznie jest to najlepszy sposób modyfikowania funkcjonalności podstawowych obiektów języka Ruby (jak się okaże w rozdziale 13.), ale odpowiedni na potrzeby zilustrowania zagadnienia. Poniższy listing 10.2 prezentuje metodę instancji `String#^`.

**Listing 10.2. Metoda operacji alternatywy wykluczającej używana dla łańcuchów**

```
class String
  def ^(key)                                  ←❶
    kenum = key.each_byte.cycle               ←❷
    each_byte.map {|byte| byte ^ kenum.next }.pack("C*") ←❸
  end
end
```

Metoda pobiera jeden argument, czyli łańcuch, który będzie używany jako baza operacji XOR (*klucz*) ❶. Konieczne jest poradzenie sobie z przypadkami, w których klucz jest krótszy niż oryginalny łańcuch. W tym celu będzie wykonywana pętla dla klucza liczbę razy wymaganą do zapewnienia wystarczającej liczby znaków dla całej operacji. Pomocne okażą się tutaj enumeratory.

Zmienna `kenum` jest powiązana z enumeratorem w oparciu o tworzenie łańcucha dwóch metod przy użyciu łańcucha klucza. Jest to metoda `each_byte`, która zwraca enumerator, przechodząc łańcuch bajt po bajcie, oraz metoda `cycle` wielokrotnie iterująca kolekcję, wznawiająca działanie od początku po osiągnięciu końca łańcucha ❷. Enumerator zmiennej `kenum` obejmuje operacje obu tych metod: każda iteracja łańcucha zapewnia kolejny jego bajt. Po zakończeniu udostępniania wszystkich bajtów enumerator wraca do początku łańcucha i ponownie iteruje bajty. Dokładnie takie zachowanie jest pożądane do zapewnienia, że uzyskamy liczbę bajtów wystarczającą do dopasowania dowolnego łańcucha, dla którego wykonywana jest operacja XOR, nawet wtedy, gdy jest to łańcuch dłuższy niż klucz. W rezultacie łańcuch klucza stał się nieskończenie długi.

W dalszej kolejności ma miejsce faktyczna operacja XOR ❸. Metoda `each_byte` używana jest w niej do iteracji bajtów łańcucha poddawanych tej operacji. Enumerator zwracany przez tę metodę jest dołączany do metody `map` w ramach łańcucha metod. Wewnątrz bloku metody `map` każdy bajt oryginalnego łańcucha jest przetwarzany przez operację XOR za pomocą „następnego” bajta z enumeratora, który bez końca wykonuje cykle dla bajtów łańcucha klucza. Cała operacja metody `map` generuje następnie tablicę bajtów poddanych operacji XOR. Pozostaje jedynie umieszczenie tych bajtów z powrotem w łańcuchu wynikowym.

Dalej pojawia się metoda `pack`. Przekształca ona tablicę w łańcuch, interpretując każdy element tablicy w sposób określony przez argument. W tym przypadku argumentem jest łańcuch `"C*"`, który oznacza: *potraktuj każdy element tablicy jako liczbę całkowitą bez znaku reprezentującą pojedynczy znak* (czyli `C`) *i przetwórz je wszystkie*

(czyli znak *). A zatem przekształcenie tablicy w łańcuch znaków jest odpowiednikiem transformacji każdego elementu tablicy do postaci znaku, a następnie wykonania operacji łączenia dla całej tablicy.

Możemy teraz wykonać operację XOR dla łańcuchów. Proces ten wygląda następująco:

```
>> str = "Ładny, krótki łańcuch."
=> "Ładny, krótki łańcuch."
>> key = "sekret!"
=> "sekret!"
>> x = str ^ key
   => "=\f\x00\x17E\x18H\a\x11\x0F\x17E\aU\x01\f\r\x15K"
>> orig = x ^ key
=> "Ładny, krótki łańcuch."
```

Jak widać, dwukrotne wykonanie operacji XOR przy użyciu tego samego klucza powoduje powrót do oryginalnego łańcucha. Wszystko to zawdzięczamy dwuwierszowej metodzie, która używa trzech enumeratorów!

**Wymuszanie kodowania**

Metoda instancji `String#^` w postaci zaimplementowanej we wcześniejszym fragmencie kodu jest podatna na problemy z kodowaniem: jeśli na przykład dwukrotnie wykonasz operację XOR dla łańcucha w formacie UTF-8 przy użyciu łańcucha w formacie ASCII, uzyskasz łańcuch zakodowany w formacie ASCII-8BIT. Aby się przed tym uchronić, dodaj wywołanie metody `force_encoding`:

```
each_byte.map {|byte| byte ^ kenum.next }.pack("C*").
                        force_encoding(self.encoding)
```

Zapewni to, że sekwencja bajtów generowana przez odwzorowanie zostanie zakodowana z wykorzystaniem kodowania oryginalnego łańcucha.

Enumeratory zapewniają całkowicie nowe narzędzie w już bogatym zestawie narzędziowym języka Ruby, które służy do iteracji oraz zarządzania. Pod względem koncepcyjnym i technicznym enumeratory różnią się od iteratorów, ale jeśli wypróbujesz je na własnych warunkach, z pewnością znajdziesz zastosowania dla nich, a także dla innych zaprezentowanych technik powiązanych z kolekcjami.

Omówienie enumeratorów podsumujemy ich wariantem określanym mianem „leniwego” enumeratora.

## 10.12. „Leniwe” enumeratory

„Leniwe” enumeratory ułatwiają selektywne wyliczanie nieskończenie dużych kolekcji. Aby zilustrować, co to oznacza, zacznijmy od sytuacji, w której operacja próbuje wyliczać nieskończenie dużą kolekcję i zawiesza się. Co będzie, gdy chcesz poznać 10 pierwszych wielokrotności liczby `3`? Aby użyć nieskończonej kolekcji, utworzymy zakres od liczby `1` do specjalnej wartości `Float::INFINITY`. W przypadku korzystania z takiego zakresu pierwszy wariant realizowania zadania może mieć następującą postać:

```
(1..Float::INFINITY).select {|n| n % 3 == 0 }.first(10)
```

Ten wiersz kodu będzie jednak wykonywany bez końca. Operacja metody `select` nigdy się nie zakończy, dlatego znajdująca się za nią w łańcuchu metoda `first` nigdy nie zostanie wykonana.

Użycie „leniwego" enumeratora pozwala uzyskać skończony wynik dla nieskończonej kolekcji. Wywołanie metody `lazy` bezpośrednio dla obiektu zakresu spowoduje utworzenie „leniwego" enumeratora dla tego zakresu:

```
>> (1..Float::INFINITY).lazy
=> #<Enumerator::Lazy: 1..Infinity>
```

Możesz następnie połączyć ten „leniwy" enumerator z metodą `select`, tworząc kaskadę „leniwych" enumeratorów:

```
>>(1..Float::INFINITY).lazy.select {|n| n % 3 == 0 }
=> #<Enumerator::Lazy: #<Enumerator::Lazy: 1..Infinity>:select>
```

Ponieważ używane są teraz „leniwe" enumeratory, możliwe jest uzyskanie zestawów wynikowych z operacji bez czekania na zakończenie nieskończonych zadań. Dokładniej rzecz biorąc, możemy zażądać 10 pierwszych wyników testu wybierania wykonywanego dla nieskończonej listy, a bez problemów nastąpi wyliczenie tylko takiej ilości danych, jaka jest wymagana do wygenerowania tych 10 wyników:

```
>> (1..Float::INFINITY).lazy.select {|n| n % 3 == 0 }.first(10)
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

W ramach wariantu tego samego tematu możemy utworzyć „leniwy" enumerator metody `select`, a następnie użyć dla niego metody `take`. Umożliwia to określenie, ile wielokrotności liczby `3` ma zostać wyświetlonych, bez wprowadzania ich liczby na stałe do kodu. Zauważ, że musisz wywołać metodę `force` dla wyniku metody `take`. W przeciwnym razie zamiast faktycznego zestawu wynikowego zostanie uzyskany jeszcze jeden „leniwy" enumerator:

```
>> my_enum = (1..Float::INFINITY).lazy.select {|n| n % 3 == 0 }
=> #<Enumerator::Lazy: #<Enumerator::Lazy: 1..Infinity>:select>
>> my_enum.take(5).force
=> [3, 6, 9, 12, 15]
>> my_enum.take(10).force
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

„Leniwe" enumeratory to w pewnym sensie specjalizowane narzędzie, którego prawdopodobnie nie będziesz potrzebować zbyt często. Enumeratory te są jednak bardzo pomocne, gdy istnieje nieskończenie duża kolekcja, a ponadto z operacji wykonywanych dla takiej kolekcji chcesz uzyskać tylko skończony zestaw wynikowy.

### 10.12.1. Problem FizzBuzz w przypadku „leniwego" enumeratora

W swojej klasycznej postaci problem `FizzBuzz` dotyczy wyświetlania liczb całkowitych od 1 do 100, z tym że obowiązują następujące reguły:

- Jeśli liczba jest podzielna przez 15, wyświetl łańcuch `"FizzBuzz"`.
- Jeśli natomiast liczba jest podzielna przez 3, wyświetl łańcuch `"Fizz"`.
- Jeśli natomiast liczba jest podzielna przez 5, wyświetl łańcuch `"Buzz"`.
- W przeciwnym razie wyświetl liczbę.

„Leniwego” enumeratora możesz użyć do stworzenia wersji problemu FizzBuzz, która może obsługiwać dowolny zakres liczb. Może to wyglądać następująco:

```
def fb_calc(i)
  case 0
  when i % 15
    "FizzBuzz"
  when i % 3
    "Fizz"
  when i % 5
    "Buzz"
  else
    i.to_s
  end
end
def fb(n)
  (1..Float::INFINITY).lazy.map {|i| fb_calc(i) }.first(n)
end
```

Możesz teraz na przykład sprawdzić dane wyjściowe metody fb dla 15 pierwszych, dodatnich liczb całkowitych:

```
p fb(15)
```

Oto dane wyjściowe:

```
["1", "2", "Fizz", "4", "Buzz", "Fizz", "7", "8", "Fizz", "Buzz", "11", "Fizz", "13", "14",
↪"FizzBuzz"]
```

Bez tworzenia „leniwego” enumeratora dla zakresu operacja odwzorowania byłaby wykonywana bez końca. „Leniwy” enumerator zapewnia, że cały proces zostanie zatrzymany po uzyskaniu żądanego wyniku.

## 10.13. Podsumowanie

W rozdziale zaprezentowano następujące zagadnienia:

- Moduł Enumerable i jego metody instancji.
- Użycie modułu Enumerable we własnych klasach.
- Podstawowe informacje o enumeratorach.
- Tworzenie enumeratorów.
- Iteracja łańcuchów.
- „Leniwe” enumeratory.

W rozdziale skupiono się na module Enumerable i klasie Enumerator, czyli dwóch blisko powiązanych ze sobą obiektach. Po pierwsze, omówiono metody instancji modułu Enumerable, które są definiowane na warunkach metody each, a ponadto są dostępne dla obiektów, dopóki odpowiadają one na metodę each, a używana klasa ma dołączony moduł Enumerable. Po drugie, przyjrzeliśmy się enumeratorom. Są to obiekty obejmujące proces iteracji innego obiektu, które same wiążą się (a dokładniej rzecz biorąc, ich metody each) z wyznaczoną metodą innego obiektu, a ponadto wykorzystują to „pasożytnicze” powiązanie z metodą each do zapewnienia pełnego zestawu funkcjonalności związanych z wyliczaniem.

Enumeratory mogą być złożone. W całości bazują na module `Enumerable`. W sytuacji, gdy enumerator jest podłączany do obiektu, który przesłonił niektóre metody modułu `Enumerable`, ważne jest pamiętanie o tym, że enumerator będzie dysponować własnymi informacjami o tym, jakie są to metody. Enumerator nie pełni roli uniwersalnego pośrednika dla innego obiektu. Enumerator działa jak coś w rodzaju „syfonu" pobierającego wartości z jednej metody innego obiektu.

Niezależnie od sposobu, czy za pośrednictwem enumeratorów, czy przez użycie bardziej klasycznego stylu iteracji w języku Ruby oraz zarządzania kolekcjami, prawie na pewno będziesz korzystać z narzędzi języka powiązanych z operacją wyliczania niemal za każdym razem podczas pisania programu Ruby. Warto dobrze zaznajomić się z modułem `Enumerable`. Stanowi on jednostkę funkcjonalną, a poziom jego możliwości potwierdza wszechobecność modułu w języku Ruby.

W dalszej kolejności zajmiemy się zagadnieniem wyrażeń regularnych i dopasowywaniem wzorców. Jak się okaże, z omawianiem obiektów łańcuchów i kolekcji przed wyrażeniami regularnymi związana jest następująca korzyść: kilka metod dopasowujących wzorzec wykonywanych dla łańcuchów zwraca swoje wyniki w postaci kolekcji, dlatego też nadają się one do iteracji. Objaśnienie wyrażeń regularnych ułatwi tworzenie w pełni funkcjonalnego zestawu narzędziowego służącego do przetwarzania łańcuchów i treści tekstowej.

# 11 Wyrażenie regularne i operacje na łańcuchach oparte na wyrażeniach regularnych

**W tym rozdziale:**

- Składnia wyrażeń regularnych.
- Operacje dopasowujące wzorce.
- Klasa `MatchData`.
- Metody wbudowane oparte na dopasowywaniu wzorców.

W tym rozdziale omówimy dostępne w języku Ruby rozwiązania, które służą do przetwarzania tekstu i dopasowywania wzorców na bazie wyrażeń regularnych. *Wyrażenie regularne* w języku Ruby spełnia takie same zadania jak w innych językach, czyli określa wzorzec znaków, który może, ale niekoniecznie, poprawnie przewidzieć (czyli dopasować) dany łańcuch. Operacje dopasowywania wzorców są wykorzystywane na potrzeby rozgałęziania warunkowego (dopasowanie/brak dopasowania), określania podłańcuchów (części łańcucha zgodne z częściami wzorca) oraz różnych technik filtrowania tekstu.

W języku Ruby wyrażenia regularne są obiektami. Komunikaty są wysyłane *do* wyrażenia regularnego. Wyrażenia regularne wzbogacają „krajobraz" języka Ruby, ale też, tak jak inne obiekty, dobrze się w niego wpasowują.

Zaczniemy od dokonania przeglądu wyrażeń regularnych. W dalszej części rozdziału zajmiemy się szczegółami dotyczącymi sposobów ich tworzenia i oczywiście korzystania z nich. W ramach omówienia użycia wyrażeń regularnych przybliżymy zarówno proste operacje dopasowywania, jak i metody, w których wyrażenia pełnią rolę w większym procesie, takim jak filtrowanie kolekcji lub powtarzane skanowanie łańcucha.

## 11.1. Czym są wyrażenia regularne?

Wyrażenia regularne pojawiają się w wielu językach programowania; występują w nich w wielu wariantach niewiele różniących się od siebie. Celem tych wyrażeń jest określanie wzorców znakowych, które następnie służą do dopasowywania (lub wykonywania odwrotnej operacji) łańcuchów. Z kolei dopasowywanie wzorca stanowi bazę takich operacji jak analizowanie plików dziennika, testowanie danych wprowadzonych za pomocą klawiatury pod kątem poprawności oraz izolowanie podłańcuchów. Inaczej mówiąc, są to operacje często stosowane przez każdego, kto musi przetwarzać łańcuchy i tekst.

Wyrażenia regularne mają dziwną reputację. Są one „skoncentrowaną" techniką zapewniającą duże możliwości. Wyrażenia regularne „wżerają się" w duży podzbiór problemów związanych z przetwarzaniem tekstu jak kwas w kłódkę. Ponadto w opinii wielu osób (w tym tych, które je dobrze rozumieją) wyrażenia regularne są trudne w użyciu i analizowaniu, a także niejasne, niemożliwe do obsługi i skrajnie nieefektywne.

Musisz sam to ocenić. Z pewnością *nie* należy wzbraniać się przed poznaniem przynajmniej podstaw dotyczących sposobu działania wyrażeń regularnych, a także tego, jak używać korzystających z nich metod języka Ruby. Jeśli nawet uznasz, że nie należysz do osób preferujących wyrażenia regularne, niezbędna będzie podstawowa wiedza na ich temat. W żadnym razie nie będziesz osamotniony, jeśli okaże się, że ostatecznie wyrażeń regularnych będziesz używać w swoich programach częściej, niż przewidywałeś.

Wiele metod wbudowanych języka Ruby pobiera wyrażenia regularne jako argumenty, a ponadto wykonuje operację wybierania lub modyfikacji dla co najmniej jednego obiektu łańcucha. Wyrażenia regularne są na przykład stosowane do *skanowania* łańcucha pod kątem wielu wystąpień wzorca, *zastępowania* podłańcucha łańcuchem zastępującym i *dzielenia* łańcucha na wiele podłańcuchów na podstawie separatora dopasowującego.

Jeśli jesteś zaznajomiony z wyrażeniami regularnymi z języka Perl, narzędzi sed, vi i Emacs lub dowolnego innego źródła, możesz pominąć lub przejrzeć pobieżnie zawarty tutaj materiał objaśniający i przejść do podrozdziału 11.5, w którym omawiane są metody języka Ruby korzystające z wyrażeń regularnych. Zauważ jednak, że wyrażenia regularne tego języka nie są takie same jak stosowane w dowolnym innym języku.

Prawie na pewno będziesz w stanie je odczytać, ale w przypadku tworzenia ich może być niezbędne poznanie różnic (np. dotyczących tego, czy nawiasy okrągłe są specjalnie traktowane domyślnie, czy po zastosowaniu znaku zmiany znaczenia).

Utwórzmy teraz kilka wyrażeń regularnych.

## 11.2. Tworzenie wyrażeń regularnych

Wyrażenia regularne są oczywiście tworzone za pomocą znanych znaków, ale konieczne jest opanowanie umiejętności czytania i zapisywania ich jako całości. Nie są one łańcuchami, a ich znaczenie nie zawsze jest tak oczywiste jak łańcuchów. Wyrażenia regularne są reprezentacjami *wzorców*.

### 11.2.1. Wyświetlanie wzorców

Wyrażenie regularne (ang. *regex*) określa wzorzec. W przypadku każdego takiego wzorca dowolny łańcuch świata jest z nim zgodny albo nie. Metody języka Ruby, które używają wyrażeń regularnych, korzystają z nich, aby określić, czy dany łańcuch pasuje do danego wzorca, albo aby po wykonaniu takiej operacji na podstawie przekazanej przez nią odpowiedzi zastosować wybrane działanie.

Wzorce określane przez wyrażenie regularne są początkowo najprostsze do zrozumienia jako wyrażone za pomocą zwykłego języka. Oto kilka przykładów wzorców wyrażonych w ten sposób:

- Litera `a`, po której następuje cyfra.
- Dowolna duża litera, po której następuje co najmniej jedna mała litera.
- Trzy cyfry, po których następuje łącznik oraz cztery cyfry.

Wzorzec może też obejmować składniki i ograniczenia powiązane z pozycjonowaniem wewnątrz łańcucha:

- Początek wiersza, po którym następuje co najmniej jeden biały znak.
- Znak kropki (`.`) na końcu łańcucha.
- Duża litera na początku słowa.

Składniki wzorca, takie jak „początek wiersza”, które dopasowują warunek, a nie znak w łańcuchu, są jednak wyrażane za pomocą znaków lub sekwencji znaków w wyrażeniu regularnym.

Wyrażenia regularne zapewniają język służący do wyrażania wzorców. Uczenie się ich tworzenia uwzględnia głównie opanowanie różnych elementów wyrażanych w obrębie wyrażenia regularnego. Najczęściej stosowane reguły konstrukcji wyrażenia regularnego są dość proste do opanowania. Trzeba jedynie pamiętać, że wyrażenie regularne, choć zawiera znaki, nie jest łańcuchem. Jest to specjalna notacja służąca do wyrażania wzorca, który może, ale niekoniecznie, poprawnie opisywać część lub całość dowolnego danego łańcucha.

### *11.2.2. Proste dopasowywanie za pomocą wyrażeń regularnych literału*

Wyrażenia regularne są instancjami klasy `Regexp`, która jest jedną z klas języka Ruby. Klasa ta zawiera konstruktor literału służący do prostego tworzenia instancji. Konstruktor literału wyrażenia regularnego ma postać pary ukośników:

```
//
```

Jakkolwiek dziwnie by to wyglądało, to naprawdę jest wyrażenie regularne, choćby w formie podstawowej. W narzędziu irb możesz sprawdzić, że wyrażenie to zapewnia instancję klasy `Regexp`:

```
>> //.class
=> Regexp
```

Szczegóły wyrażenia regularnego znajdują się między ukośnikami. Tworzenie kilku prostych wyrażeń regularnych rozpoczniemy po przyjrzeniu się podstawom procesu dopasowywania.

Dowolna operacja dopasowywania wzorca obejmuje dwóch głównych „graczy": wyrażenie regularne i łańcuch. Wyrażenie regularne określa przewidywania związane z łańcuchem, które zostaną przez niego spełnione (dopasowanie wzorca) albo nie.

Najprostszym sposobem stwierdzenia, czy występuje dopasowanie między wzorcem i łańcuchem, jest użycie metody `match`. Przy wyborze jednego z dwóch dostępnych wariantów obowiązuje dowolność. Obiekty wyrażeń regularnych i obiekty łańcuchów odpowiadają metodzie `match`, dlatego oba następujące przykładowe wiersze zostaną pomyślnie wykonane, powodując wyświetlenie komunikatu `"Dopasowanie!"`:

```
puts "Dopasowanie!" if /abc/.match("Alfabet zaczyna się od łańcucha abc.")
puts "Dopasowanie!" if "Alfabet zaczyna się od łańcucha abc.".match(/abc/)
```

Wersja metody `match` z obiektem łańcucha (drugi z powyższych wierszy) różni się od wersji z obiektem wyrażenia regularnego tym, że przekształca argument łańcucha *w* wyrażenie regularne (zajmiemy się tym trochę później). W przykładzie argument jest już wyrażeniem regularnym (`/abc/`), dlatego nie jest wymagane przekształcenie.

Oprócz metody `match` w języku Ruby dostępny jest też operator dopasowujący wzorzec =~ (znak równości i tylda), który trafia między łańcuch i wyrażenie regularne:

```
puts "Dopasowanie!" if /abc/ =~ "Alfabet zaczyna się od łańcucha abc."
puts "Dopasowanie!" if "Alfabet zaczyna się od łańcucha abc." =~ /abc/
```

Jak można przypuszczać, taki operator dopasowujący wzorzec jest metodą instancji klas `String` i `Regexp`. Jest to jedna z wielu metod języka Ruby, która zapewnia „lukier" składniowy w stylu związanym z użyciem operatora wrostkowego.

Metoda `match` i operator =~ są jednakowo przydatne, gdy udzielono prostej odpowiedzi (tak/nie) na pytanie dotyczące tego, czy występuje zgodność między łańcuchem i wzorcem. W przypadku braku zgodności otrzymasz wartość `nil`. Jest to pomocne przy instrukcjach warunkowych. We wszystkich czterech powyższych przykładach wyniki ich operacji dopasowania sprawdzano za pomocą testu instrukcji `if`. Metoda `match`

i operator =~ różnią się głównie tym, co zwracają w przypadku *wystąpienia* dopasowania: operator =~ zwraca indeks liczbowy znaku w łańcuchu, od którego zaczyna się dopasowanie, natomiast metoda `match` zwraca instancję klasy `MatchData`:

```
>> "Alfabet zaczyna się od łańcucha abc" =~ /abc/
=> 25
>> /abc/.match("Alfabet zaczyna się od łańcucha abc.")
=> #<MatchData "abc">
```

W pierwszym przykładzie dopasowanie określane jest w pozycji 25. łańcucha. W drugim przykładzie utworzenie obiektu `MatchData` oznacza, że znaleziono dopasowanie.

Obiekty `MatchData` zostaną omówione w dalszej części rozdziału. W tym miejscu zajmiemy się głównie uzyskiwaniem odpowiedzi (tak/nie) na próbę dopasowania, dlatego sprawdzi się dowolna z dotychczas zaprezentowanych technik. W celu zachowania spójności, a także dlatego, że bardziej będą nas interesować obiekty `MatchData` niż indeksy liczbowe podłańcuchów, większość przykładów zamieszczonych w rozdziale będzie dotyczyć metody `Regexp#match`.

Pora przyjrzeć się dokładniej tworzeniu wyrażenia regularnego.

## 11.3. Tworzenie wzorca w wyrażeniu regularnym

Tworząc wyrażenie regularne, definicję wzorca umieszczasz między ukośnikami. Pamiętaj, że to, co wstawiasz, nie jest łańcuchem, lecz zestawem przewidywań i ograniczeń, które mają być szukane w łańcuchu.

Oto możliwe składniki wyrażenia regularnego:

- *Znaki literału* oznaczające: „dopasuj ten znak".
- *Znak wieloznaczny kropki* (`.`) oznaczający: „dopasuj dowolny znak" (z wyjątkiem `\n`, czyli znaku nowego wiersza).
- *Klasy znaków* oznaczające: „dopasuj jeden z tych znaków".

Kolejno zostaną omówione wymienione składniki. Później zdobytą wiedzę wykorzystamy do bliższego omówienia operacji dopasowywania.

### 11.3.1. Znaki literału we wzorcach

Dowolny znak literału umieszczony w wyrażeniu regularnym dopasowuje się *sam* w łańcuchu. A zatem wyrażenie regularne:

```
/a/
```

dopasowuje dowolny znak zawierający literę `a`.

Niektóre znaki mają specjalne znaczenie dla analizatora wyrażeń regularnych (o czym wkrótce dokładnie się dowiesz). W celu dopasowania jednego z tych znaków specjalnych jako jego samego musisz zmienić jego znaczenie za pomocą lewego ukośnika `\`. Aby na przykład dopasować znak `?` (pytajnik), konieczne jest użycie następującego kodu:

```
/\?/
```

Znak \ oznacza: „nie traktuj następnego znaku jako specjalnego, lecz jako samego siebie".

Znaki specjalne obejmują podane w nawiasach okrągłych: (^ $ ? . / \ [ ] { } ( ) + *). Jak widać, wśród nich znajduje się kropka, która w wyrażeniach regularnych jest znakiem specjalnym.

### 11.3.2. Znak wieloznaczny kropki (.)

Czasem niezbędne jest dopasowanie *dowolnego znaku* w określonym miejscu wzorca. Umożliwia to specjalny znak wieloznaczny kropki (.). Kropka dopasowuje każdy znak z wyjątkiem znaku nowego wiersza (istnieje też możliwość dopasowywania znaków nowego wiersza, o czym będzie mowa trochę później).

Wzorzec w następującym wyrażeniu regularnym dopasowuje łańcuchy „bolorowy" i „kolorowy":

```
/.olorowy/
```

Wzorzec dopasowuje też łańcuchy „%olorowy" i „8olorowy":

```
puts "Dopasowanie!" if /.olorowy /.match("%olorowy")
```

Znak wieloznaczny kropki jest przydatny, ale czasami zapewnia więcej dopasowań, niż żądano. Przy wykorzystaniu klas znaków możliwe jest wymuszenie ograniczeń dla dopasowań przy jednoczesnym zezwoleniu na wiele łańcuchów.

### 11.3.3. Klasy znaków

*Klasa znaków* jest jawną listą znaków umieszczonych wewnątrz wyrażenia regularnego w nawiasach kwadratowych:

```
/[bk]olorowy/
```

Oznacza to: „dopasuj literę *b* lub *k*, po której następuje łańcuch *olorowy*". Ten nowy wzorzec dopasowuje łańcuch „bolorowy" lub „kolorowy", lecz nie „%olorowy". Klasa znaków jest czymś w rodzaju częściowego lub ograniczonego znaku wieloznacznego: zezwala na wiele możliwych znaków, ale na tylko ich ograniczoną liczbę.

Wewnątrz klasy znaków możesz też wstawić *zakres* znaków. Typowy przypadek dotyczy następujących czterech znaków:

```
/[a-z]/
```

Aby dopasować cyfrę szesnastkową, możesz użyć kilku zakresów wewnątrz klasy znaków:

```
/[A-Fa-f0-9]/
```

Wiersz powoduje dopasowanie dowolnego znaku z zakresu od litery *a* do litery *f* (dużej lub małej) lub dowolnej cyfry.

Czasami konieczne jest dopasowanie dowolnego znaku *z wyjątkiem* tych, które są na specjalnej liście. Na przykład możesz szukać pierwszego znaku w łańcuchu, który *nie* jest poprawną cyfrą szesnastkową.

**Klasy znaków są dłuższe niż to, co dopasowują**

Nawet krótka klasa znaków, taka jak `[a]`, zajmuje więcej niż jedno miejsce w wyrażeniu regularnym. Pamiętaj jednak, że każda klasa znaków dopasowuje *jeden* znak łańcucha. Gdy przyjrzysz się klasie znaków, takiej jak `/[dr]/`, możesz uznać, że dopasuje ona podłańcuch `dr`. Tak jednak nie jest: klasa ta dopasuje literę `d` lub `r`.

Tego rodzaju wyszukiwanie negatywne jest wykonywane przez negację klasy znaków. W tym celu na początku klasy umieszczasz znak `^`. Na przykład następująca klasa znaków dopasowuje dowolny znak z wyjątkiem poprawnej cyfry szesnastkowej:

```
/[^A-Fa-f0-9]/
```

Następujące wiersze prezentują, w jaki sposób możesz znaleźć indeks pierwszego wystąpienia w łańcuchu znaku, który nie jest szesnastkowy:

```
>> string = "ABC3934 to liczba szesnastkowa."
=> "ABC3934 to liczba szesnastkowa."
>> string =~ /[^A-Fa-f0-9]/
=> 7
```

Klasa znaków, czy to pozytywna, czy negatywna, może zawierać dowolne znaki. Niektóre klasy znaków są tak powszechne, że mają specjalne skróty.

### SPECJALNE SEKWENCJE ZMIANY ZNACZENIA DLA TYPOWYCH KLAS ZNAKÓW

Aby dopasować dowolną cyfrę, możesz zastosować następujący wiersz:

```
/[0-9]/
```

Ten sam efekt możesz też uzyskać w bardziej spójny sposób za pomocą specjalnej sekwencji zmiany znaczenia `\d`:

```
/\d/
```

Zauważ, że w tym przypadku nie występują nawiasy kwadratowe, a jedynie `\d`. Oto dwie inne przydatne sekwencje przeznaczone dla predefiniowanych klas znaków:

- Sekwencja `\w` dopasowuje dowolną cyfrę, znak alfabetu lub znak podkreślenia (_).
- Sekwencja `\s` dopasowuje dowolny biały znak (spacja, tabulator, znak nowego wiersza).

Każda z tych predefiniowanych klas znaków ma też postać negatywną. Możliwe jest dopasowanie w następujący sposób dowolnego znaku, który nie jest cyfrą:

```
/\D/
```

Sekwencja `\W` dopasowuje dowolny znak inny niż znak alfanumeryczny lub znak podkreślenia, a sekwencja `\S` dopasowuje każdy znak inny niż biały znak.

Pomyślne wywołanie metody `match` powoduje zwrócenie obiektu `MatchData`. Przyjrzyjmy się bliżej tym obiektom i ich możliwościom.

## 11.4. Dopasowywanie i przechwytywanie podłańcuchów oraz obiekt MatchData

Dotychczas omówiliśmy następujące podstawowe operacje dopasowywania:

```
regex.match(string)
string.match(regex)
```

Zasadniczo są to testy prawdy/fałszu, czyli wystąpienia dopasowania lub jego braku. Przeanalizujemy, co ma miejsce w przypadku pomyślnych i niepomyślnych operacji dopasowania, a także co takie operacje mogą zapewnić poza odpowiedzią tak/nie.

### 11.4.1. Przechwytywanie dopasowań podrzędnych za pomocą nawiasów okrągłych

Jedną z najważniejszych technik tworzenia wyrażeń regularnych jest użycie nawiasów okrągłych do określania *przechwyceń*.

Najpierw należy przybliżyć ideę tej techniki. Sprawdzanie dopasowania między łańcuchem (np. wiersz z pliku) i wzorcem wynika przeważnie z potrzeby przetworzenia w jakiś sposób łańcucha lub, co jest częstsze, jego części. Notacja przechwytywania umożliwia wyizolowanie i zapisanie podłańcuchów łańcucha zgodnych z konkretnymi podwzorcami.

Dla przykładu załóżmy, że istnieje łańcuch zawierający informacje o osobie:

```
Nowak,Tomasz,prof.,utalentowany naukowiec
```

Niezbędne jest wyodrębnienie z tego łańcucha nazwiska i tytułu osoby. Wiemy, że pola są oddzielone przecinkami, a ponadto znana jest ich kolejność: nazwisko, imię, tytuł i stanowisko.

Aby utworzyć wzorzec dopasowujący taki łańcuch, w przypadku następujących wierszy możesz posługiwać się językiem mówionym:

> *Najpierw* `kilka znaków alfabetu,`
> *a następnie* `przecinek,`
> *a następnie* `kilka znaków alfabetu,`
> *a następnie* `przecinek,`
> *a następnie* `albo 'prof.', albo 'dr'`

Nie komplikujemy niczego: żadnych nazw z łącznikami ani zwrotów grzecznościowych. W związku z tym wyrażenie regularne może mieć następującą postać:

```
/[A-Za-z]+,[A-Za-z]+,prof?\./
```

Pytajnik występujący po literze `f` oznacza: *dopasuj jedną literę f albo nic*. Wyrażenie tego w ten sposób umożliwia zwięzłe dopasowanie łańcuchów „prof." i „doc.". Wzorzec dopasowuje łańcuch, co potwierdzają wiersze w narzędziu irb:

```
>> /[A-Za-z]+,[A-Za-z]+,prof?\./.match("Nowak,Tomasz,prof.,utalentowany naukowiec")
=> #<MatchData "Nowak,Tomasz,prof.">
```

Zamiast obiektu `nil` otrzymano obiekt `MatchData`, ponieważ wystąpiło dopasowanie.

Ale co teraz? W jaki sposób wyizolujemy interesujące nas podłańcuchy (`"Nowak"` i `"prof."`)?

W tym przypadku przydatne okażą się grupowania w nawiasach okrągłych. Wymagane będą dwa takie grupowania: pierwsze dla podwzorca dopasowującego nazwisko, a drugie dla podwzorca dopasowującego tytuł:

```
/([A-Za-z]+),[A-Za-z]+,(prof?\.)/
```

Gdy wykonamy operację dopasowania:

```
/([A-Za-z]+),[A-Za-z]+,(prof?\.)/.match("Nowak,Tomasz,prof.,utalentowany naukowiec")
```

Mają miejsce następujące dwa działania:

- Uzyskiwany jest obiekt `MatchData`, który zapewnia dostęp do dopasowań podrzędnych (poniżej zamieszczono więcej informacji o tym).
- Interpreter języka Ruby automatycznie zapełnia serię zmiennych, które również zapewniają dostęp do tych dopasowań podrzędnych.

Zapełniane zmienne to zmienne globalne, których nazwy są oparte na liczbach: `$1`, `$2` itd. Zmienna `$1` zawiera podłańcuch dopasowany przez podwzorzec wewnątrz *pierwszego* od lewej strony zestawu nawiasów okrągłych w wyrażeniu regularnym. Sprawdzenie zmiennej `$1` po poprzednim dopasowaniu (na przykład za pomocą instrukcji `puts $1`) powoduje wyświetlenie łańcucha `Nowak`. W zmiennej `$2` znajduje się podłańcuch dopasowany przez *drugi* podwzorzec itd. Ogólnie rzecz biorąc, reguła jest następująca: po pomyślnej operacji dopasowania zmienna `$`*n* (gdzie *n* to liczba) zawiera podłańcuch dopasowany przez podwzorzec wewnątrz *n*-tego zestawu nawiasów okrągłych, licząc od lewej strony w wyrażeniu regularnym.

> **UWAGA** Jeśli używałeś języka Perl, być może spotkałeś się ze zmienną `$0`, która nie reprezentuje konkretnego przechwyconego podwzorca, lecz cały pomyślnie dopasowany podłańcuch. W języku Ruby zmienna `$0` używana jest do czegoś innego: zawiera ona nazwę pliku programu Ruby, z którego początkowo uruchomiono bieżący program lub skrypt. Zamiast stosowania zmiennej `$0` na potrzeby dopasowań wzorców, język Ruby zapewnia metodę `string`, która jest wywoływana w obiekcie `MatchData` zwracanym przez dopasowanie. W punkcie 11.4.2 zamieszczono przykład użycia tej metody.

Techniki te można połączyć z interpolacją łańcuchów w celu wygenerowania zwrotu grzecznościowego w liście, bazując na operacji dopasowania oraz pobierania zmiennych `$1` i `$2`:

```
line_from_file = "Nowak,Tomasz,prof.,utalentowany naukowiec"
/([A-Za-z]+),[A-Za-z]+,(prof?\.)/.match(line_from_file)
puts "Szanowny #{$2} #{$1},"  ←——— Dane wyjściowe: Szanowny prof. Nowak,
```

Zmienne w stylu `$n` są pomocne przy pobieraniu dopasowań podrzędnych. To samo możesz jednak uzyskać w sposób programowy cechujący się większym poziomem uporządkowania, odpytując obiekt `MatchData` zwracany przez operację dopasowania.

### 11.4.2. Dopasowanie pomyślne i zakończone niepowodzeniem

Każda operacja dopasowania kończy się pomyślnie lub niepowodzeniem. Zacznijmy od prostszego przypadku, czyli niepowodzenia. Gdy próbujesz dopasować łańcuch do wzorca, a łańcuch okazuje się niezgodny, wynikiem zawsze jest obiekt `nil`:

```
>> /a/.match("b")
=> nil
```

W przeciwieństwie do obiektu `nil` obiekt `MatchData` zwracany przez pomyślną operację dopasowania ma wartość boolowską prawda, co jest pomocne w przypadku prostych testów dopasowanie/brak dopasowania. Poza tym obiekt ten przechowuje informacje o dopasowaniu, które możesz wyodrębnić za pomocą odpowiednich metod: informacje dotyczą początkowego miejsca dopasowania (pozycja, w której występuje znak w łańcuchu), liczby dopasowanych znaków łańcucha, danych przechwyconych w grupach ograniczonych nawiasami okrągłymi itd.

Aby użyć obiektu `MatchData`, musisz go najpierw zapisać. Przeanalizuj przykład, w którym z łańcucha chcesz wyodrębnić numer telefonu i różne jego części (kierunkowy i sam numer) zapisać w grupach. Listing 11.1 prezentuje, jak to można zrealizować. Listing ma też na celu objaśnienie tego, jak korzystać z niektórych popularniejszych metod obiektu `MatchData`.

**Listing 11.1. Dopasowanie numeru telefonu i odpytywanie wynikowego obiektu MatchData**

```
string = "Mój numer telefonu to (123) 555-1234."
phone_re = /\((\d{3})\)\s+(\d{3})-(\d{4})/
m = phone_re.match(string)
unless m
  puts "Niestety, nie wystąpiło dopasowanie."
  exit                                        ← Koniec działania programu
end
print "Cały łańcuch początkowy: "
puts m.string                                 ← ❶
print "Cała dopasowana część łańcucha: "
puts m[0]                                     ← ❷
puts "Trzy przechwycenia: "
3.times do |index|                            ← ❸
  puts "Przechwycenie ##{index + 1}: #{m.captures[index]}"
end
puts "Oto inny sposób uzyskania dostępu do pierwszego przechwycenia:"
print "Przechwycenie #1: "
puts m[1]                                     ← ❹
```

W tym kodzie użyto metody `string` obiektu `MatchData` ❶ zwracającej cały łańcuch, dla której wykonano operację dopasowania. W celu uzyskania części łańcucha zgodnej z wzorcem adresujemy obiekt `MatchData` za pomocą nawiasów kwadratowych z indeksem `0` ❷. Używamy też zmyślnej metody `times` ❸, aby dokładnie trzy razy wykonać

iterację bloku kodu i kolejno wyświetlić dopasowania podrzędne (przechwycenia w nawiasach okrągłych). Wewnątrz bloku kodu metoda o nazwie `captures` wyodrębnia podłańcuchy zgodne z częściami wzorca zawartymi w nawiasach okrągłych. Na końcu analizujemy pierwsze przechwycenie, tym razem za pomocą innej techniki ❹, czyli indeksowania obiektu `MatchData` bezpośrednio przy użyciu nawiasów kwadratowych i dodatnich liczb całkowitych, z których każda odpowiada przechwyceniu.

Oto dane wyjściowe kodu z listingu:

```
Cały łańcuch początkowy: Mój numer telefonu to (123) 555-1234.
Cała dopasowana część łańcucha: (123) 555-1234
Trzy przechwycenia:
Przechwycenie #1: 123
Przechwycenie #2: 555
Przechwycenie #3: 1234
Oto inny sposób uzyskania dostępu do pierwszego przechwycenia:
Przechwycenie #1: 123
```

W ten sposób zorientowałeś się w rodzajach dopasowywanych danych, które mogą być wyodrębniane z obiektu `MatchData`. Możesz zauważyć, że istnieją dwa sposoby pobierania przechwyceń. Przyjrzyjmy się bliżej tym technikom.

### 11.4.3. Dwa sposoby uzyskiwania przechwyceń

Jeden ze sposobów uzyskiwania przechwyceń w nawiasach okrągłych z obiektu `MatchData` polega na bezpośrednim indeksowaniu obiektu w stylu tablic:

```
m[1]
m[2]
#itp.
```

W pierwszym wierszu zostanie wyświetlone pierwsze przechwycenie (pierwszy od lewej zestaw nawiasów okrągłych), w drugim wierszu drugie przechwycenie itd.

Jak pokazuje listing 11.1, indeks `0` zapewnia cały dopasowany łańcuch. Począwszy od indeksu `1`, indeks *n* zapewnia *n*-te przechwycenie, bazując na liczeniu od lewej strony otwierających nawiasów okrągłych (*n*, gdzie $n > 0$, zawsze odpowiada liczbie w zmiennej globalnej `$n`).

Inną techniką uzyskiwania przechwyceń w nawiasach okrągłych z obiektu `MatchData` jest użycie metody `captures`, która w pojedynczej tablicy zwraca wszystkie przechwycone podłańcuchy. Ponieważ jest to zwykła tablica, jej pierwszy element, który zasadniczo jest taki sam jak zmienna globalna `$1`, to element `0`, a nie element `1`. Inaczej mówiąc, zastosowanie mają następujące równoznaczne wiersze:

```
m[1] == m.captures[0]
m[2] == m.captures[1]
```

itd.

Należy coś wspomnieć o powtarzającym się stwierdzeniu dotyczącym liczenia od lewej strony nawiasów okrągłych. Jeśli nie poznasz reguły, niektóre wyrażenia regularne mogą być niejasne w kontekście swoich nawiasów okrągłych przechwyceń. Dla przykładu użyj następującego wiersza:

```
/((a)((b)c))/.match("abc")
```

Co będzie zawarte w różnych przechwyceniach? Cóż, wystarczy policzyć od lewej strony otwierające nawiasy okrągłe. Dla każdego takiego nawiasu znajdź jego odpowiednik po prawej stronie. Wszystko, co znajduje się wewnątrz takiej pary nawiasów, będzie przechwyceniem o numerze *n* (dla dowolnego maksymalnego *n*, jakie zostanie określone).

Oznacza to, że pierwszym przechwyceniem będzie łańcuch `"abc"`, ponieważ jest to część łańcucha dopasowana do wzorca między nawiasami okrągłymi położonymi najbardziej na zewnątrz. Następne nawiasy okrągłe otaczają łańcuch `"a"`, który będzie drugim przechwyceniem. W dalszej kolejności występuje łańcuch `"bc"`, a po nim łańcuch `"b"`. Z tym łańcuchem powiązany jest ostatni otwierający nawias okrągły.

Reprezentacja łańcuchowa obiektu `MatchData` uzyskana z tego dopasowania zapewni przechwycenia:

```
>> /((a)((b)c))/.match("abc")
=> #<MatchData "abc" 1:"abc" 2:"a" 3:"bc" 4:"b">
```

Z całą pewnością przechwycenia dokładnie odpowiadają temu, co zostało dopasowane między parami nawiasów okrągłych liczonych od lewej strony.

### PRZECHWYCENIA NAZWANE

Przechwytywanie podwyrażeń indeksowanych za pomocą liczby z pewnością jest pomocne, ale dostępny jest inny (czasami bardziej przyjazny dla czytelnika) sposób, który umożliwia nadawanie etykiet podwyrażeniom. Mowa o przechwyceniach nazwanych.

Poniżej zaprezentowano przykład. Następujące wyrażenie regularne dopasuje nazwę w postaci „Dawid A. Nowak" lub odpowiadającą jej „Dawid Nowak" (bez inicjału w środku):

```
>> re = /(?<first>\w+)\s+((?<middle>\w\.)\s+)?(?<last>\w+)/
```

Czym są tutaj słowa `first`, `middle` i `last`? Zapewniają one przechwycenia nazwane, czyli ujęte w nawiasy okrągłe przechwycenia, które możesz uzyskać z obiektu `MatchData` za pomocą słów, a nie liczb.

Jeśli wykonujesz operację dopasowania za pomocą tego wyrażenia regularnego, potwierdzenie przechwyceń nazwanych znajdziesz w wyświetlonych na ekranie danych wyjściowych reprezentujących obiekt `MatchData`:

```
>> m = re.match("Dawid A. Nowak")
=> #<MatchData "Dawid A. Nowak" first:"Dawid" middle:"A." last:"Nowak">
```

Możesz teraz odpytać obiekt o jego przechwycenia nazwane:

```
>> m[:first]
=> "Dawid"
```

Przechwycenia nazwane mogą powiększyć wyrażenia regularne, ale w zamian semantyka związana z pobieraniem przechwyceń z dopasowania będzie bazować na słowach, a nie na liczbach. Dzięki temu wyrażenia będą potencjalnie bardziej przejrzyste,

a ponadto w mniejszym stopniu będą wymagać dokumentacji. Ponadto nie jest wymagane liczenie par nawiasów okrągłych w celu uzyskania odwołania do przechwyconych podłańcuchów.

Obiekty `MatchData` zapewniają informacje o przechwyceniach w nawiasach okrągłych, które w razie potrzeby mogą zostać pobrane i wykorzystane.

### 11.4.4. Inne informacje zawarte w obiekcie MatchData

Kod zaprezentowany w listingu 11.2, który przewidziano jako wariant kodu z listingu 11.1, zapewnia krótkie przykłady kilku dodatkowych metod obiektu `MatchData`.

**Listing 11.2. Uzupełniający kod dla operacji dopasowania numeru telefonu**

```
print "Część łańcucha przed dopasowaną częścią: "
puts m.pre_match
print "Część łańcucha po dopasowanej części: "
puts m.post_match
print "Drugie przechwycenie rozpoczęło się od znaku "
puts m.begin(2)
print "Trzecie przechwycenie zakończyło się na znaku "
puts m.end(3)
```

Oto dane wyjściowe tego dodatkowego kodu:

```
Część łańcucha przed dopasowaną częścią: Mój numer telefonu to
Część łańcucha po dopasowanej części: .
Drugie przechwycenie rozpoczęło się od znaku 25
Trzecie przechwycenie zakończyło się na znaku 33
```

Metody `pre_match` i `post_match` widoczne w tym listingu zależą od tego, czy w przypadku pomyślnego dopasowania łańcucha może on następnie być traktowany jako złożony z trzech części: części przed częścią dopasowaną do wzorca, części dopasowanej do wzorca oraz części występującej po części zgodnej ze wzorcem. Dowolna z tych części lub wszystkie mogą być łańcuchem pustym. W przypadku tego listingu tak nie jest: łańcuchy metod `pre_match` i `post_match` zawierają znaki (choć w przypadku drugiej metody jest to tylko jeden znak).

W powyższym listingu znajdują się również metody `begin` i `end`. Zapewniają one informacje o tym, gdzie rozpoczynają się i kończą różne przechwycenia (jeśli występują) w nawiasach okrągłych. Aby uzyskać informacje o przechwyceniu *n*, metodzie `begin` i/lub `end`, przekaż *n* jako argument.

Obiekt `MatchData` to swego rodzaju biuro z informacjami o tym, co się stało w momencie napotkania łańcucha przez wzorzec. Dysponując taką wiedzą, kontynuujmy omawianie technik, które mogą zostać użyte do tworzenia wyrażeń regularnych i ich stosowania. Zaczniemy od zestawu ważnych składników wyrażeń regularnych: kwantyfikatorów, zakotwiczeń i modyfikatorów. Poznanie tych składników ułatwi zarówno tworzenie własnych wyrażeń regularnych, jak i ich czytanie. Jeśli dopasowanie `/abc/` jest już dla Ciebie zrozumiałe, dopasowanie `/^x?[yz]{2}.*\z/i` również wkrótce takie będzie.

**Zmienna globalna $~ obiektu MatchData**

Każdorazowo w przypadku wykonania pomyślnej operacji dopasowania za pomocą metody `match` lub `=~` interpreter języka Ruby ustawia zmienną globalną `$~` na obiekt `MatchData` reprezentujący dopasowanie. Gdy dopasowanie się nie powiedzie, dla zmiennej tej jest ustawiany obiekt `nil`. Oznacza to, że zawsze możesz skorzystać z obiektu `MatchData` do przeprowadzenia analizy, nawet jeśli użyjesz metody `=~`.

## 11.5. Dostrajanie wyrażeń regularnych za pomocą kwantyfikatorów, zakotwiczeń i modyfikatorów

*Kwantyfikatory* umożliwiają określenie, ile razy pod rząd coś ma zostać dopasowane. *Zakotwiczenia* pozwalają ustalić, że dopasowanie występuje w konkretnym miejscu struktury łańcucha (początek łańcucha, koniec wiersza, granica słowa itp.). *Modyfikatory* przypominają przełączniki przestawiane w celu zmiany zachowania mechanizmu wyrażeń regularnych (na przykład przez spowodowanie rozpoznawania wielkości znaków lub zmianę sposobu obsług białych znaków).

Kolejno przyjrzymy się kwantyfikatorom, zakotwiczeniom i modyfikatorom.

### 11.5.1. Ograniczanie dopasowań za pomocą kwantyfikatorów

Składnia wyrażeń regularnych zapewnia sposoby określania nie tylko tego, co jest żądane, ale też w jakiej ilości: dokładnie jednego konkretnego znaku, od pięciu do dziesięciu powtórzeń podwzorca itd.

Wszystkie kwantyfikatory przetwarzają pojedynczy znak (może być reprezentowany przez klasę znaków) lub grupę w nawiasach okrągłych. Gdy określasz, że na przykład chcesz dopasować trzy kolejne wystąpienia określonego podwzorca, może on być tylko jednym znakiem lub dłuższym podwzorcem umieszczonym w nawiasach okrągłych.

#### JEDEN LUB ŻADEN

Zaprezentowano już przykład kwantyfikatora typu „żaden lub jeden”. Omówmy go trochę obszerniej.

Dopasowane mają być anglojęzyczne tytuły grzecznościowe „Mr” lub „Mrs”. Aby było to bardziej interesujące, zostaną uwzględnione zarówno amerykańskie wersje zwrotów zakończone kropką, jak i wersje brytyjskie bez kropki. Wzorzec może zostać opisany w następujący sposób:

```
litera M, po której występuje litera r, a następnie
żadna lub jedna litera s, po której następuje
żaden lub jeden znak '.'
```

Notacja wyrażeń regularnych zawiera specjalny znak służący do reprezentowania sytuacji, w której występuje jeden znak albo żaden. Znakiem tym jest pytajnik (?). Opisany powyżej wzorzec zostałby za pomocą tej notacji wyrażony w następujący sposób:

```
/Mrs?\.?/
```

Pytajnik występujący po literze *s* oznacza, że łańcuch zawierający ją na takiej pozycji dopasuje wzorzec, jak również będzie tak w przypadku łańcucha bez litery *s*. Dla

kropki literału (zwróć uwagę na lewy ukośnik wskazujący, że jest to rzeczywista kropka, a nie specjalny znak symbolu wieloznacznego), po której następuje pytajnik, obowiązuje taka sama zasada. A zatem cały wzorzec dopasuje łańcuchy „Mr", „Mrs", „Mr." lub „Mrs." (choć zostaną też dopasowane łańcuchy „ABCMr." i „Mrs!"; w punkcie 11.5.3 przy omawianiu zakotwiczeń dowiesz się, jak dokładniej ograniczać dopasowanie).

Pytajnik jest często używany z klasami znaków, aby wskazać jeden spośród kilku dowolnych znaków albo żaden. Jeśli na przykład szukasz jednej lub dwóch cyfr pod rząd, taką część wzorca możesz wyrazić w następujący sposób:

```
\d\d?
```

Sekwencja ta dopasuje łańcuchy „1", „55", „03" itd.

Oprócz kwantyfikatora „jeden lub żaden" istnieje kwantyfikator „więcej niż jeden lub żaden".

### WIĘCEJ LUB ŻADEN

Dość częstym przypadkiem jest sytuacja, w której łańcuch do dopasowania zawiera białe znaki, ale nie ma pewności, ile ich jest. Załóżmy, że w dokumencie XML próbujesz dopasować znaczniki domykające `</poem>`. Taki znacznik może zawierać biały znak. Wszystkie następujące znaczniki są równoważne:

```
</poem>
< /poem>
</     poem>
</poem
>
```

Aby dopasować znacznik, konieczne jest zezwolenie we wzorcu na nieprzewidywalną liczbę białych znaków (z ich brakiem włącznie).

Coś takiego umożliwia kwantyfikator *więcej niż jeden lub żaden* w postaci gwiazdki (*):

```
/<\s*\/\s*poem\s*>/
```

Każdorazowe pojawienie się sekwencji `\s*` oznacza, że dopasowywany łańcuch może w danym miejscu dopasowania zawierać więcej niż jeden biały znak albo nie zawierać żadnego (zwróć uwagę na konieczność zmiany znaczenia ukośnika we wzorcu za pomocą lewego ukośnika, gdyż w przeciwnym razie zostałby zinterpretowany jako ukośnik sygnalizujący koniec wyrażenia regularnego).

Należy zauważyć, że wyrażenia regularne nie pozwalają na wszystko. W szczególności częstą i właściwą obserwacją jest brak możliwości analizowania dowolnych danych XML z wykorzystaniem wyrażeń regularnych. Ma to związek z zagnieżdżaniem elementów, a także sposobami reprezentowania danych znakowych. Jeśli jednak skanujesz dokument w celu ustalenia przybliżonej liczby zawartych w nim poematów, a ponadto dopasowujesz i liczysz znaczniki poematów, wysokie jest prawdopodobieństwo uzyskania szukanych informacji.

Kolejny wśród kwantyfikatorów to kwantyfikator „jeden lub więcej".

**JEDEN LUB WIĘCEJ**

Kwantyfikator „jeden lub więcej" jest znakiem plus (+) wstawianym po znaku lub grupie w nawiasach okrągłych, dla których żądane jest co najmniej jedno dopasowanie. Dopasowanie ma miejsce, gdy łańcuch zawiera co najmniej jedno wystąpienie określonego podwzorca w odpowiednim miejscu. Na przykład następujący wzorzec:

```
/\d+/
```

dopasowuje dowolną sekwencję złożoną z jednej lub większej liczby kolejnych cyfr:

```
/\d+/.match("Gdzieś znajduje się cy3fra...") ←—— Powodzenie
/\d+/.match("Brak tutaj cyfr. Kontynuuj.")   ←—— Niepowodzenie
/\d+/.match("Cyfry-To-My 2345")              ←—— Powodzenie
```

Oczywiście jeśli zastosujesz nawiasy okrągłe, możesz dowiedzieć się, co zostało dopasowane:

```
/(\d+)/.match("Cyfry-To-My 2345")
puts $1
```

W tym przypadku dane wyjściowe to 2345.

Pojawia się jednak następująca kwestia. Zadaniem wzorca \d+ jest dopasowanie co najmniej jednej cyfry. Oznacza to, że od razu po stwierdzeniu przez mechanizm wyrażeń regularnych (część interpretera w całości odpowiedzialna za obsługę dopasowywania wzorców), iż łańcuch zawiera cyfrę 2, uzyska on informacje wystarczające do wywnioskowania wystąpienia dopasowania. Oczywiście mechanizm kontynuuje działanie. Nie przerywa operacji dopasowania łańcucha do momentu dotarcia do cyfry 5. Możesz to wywnioskować na podstawie wartości zmiennej $1: to, że ma ona wartość 2345, oznacza, że podwyrażenie \d+, które jest tym, co znajduje się w pierwszym zestawie nawiasów okrągłych, uważane jest za odpowiedzialne za dopasowanie tego łańcucha złożonego z czterech cyfr.

Dlaczego jednak mają być dopasowywane cztery cyfry, gdy do potwierdzenia poprawności rozumowania wymagana jest jedna cyfra? Jak w życiu, tak w przypadku analizy wyrażeń regularnych odpowiedzią na to pytanie jest zachłanność.

### *11.5.2. „Zachłanne" (i inne) kwantyfikatory*

Kwantyfikatory * (jeden lub więcej albo żaden) i + (jeden lub więcej) są *zachłanne*. Oznacza to, że dopasowują one jak największą możliwą liczbę znaków, co odpowiada zezwoleniu na dopasowanie reszty wzorca.

Przyjrzyj się, co w następującym fragmencie kodu jest dopasowywane przez wzorzec .*:

```
string = "abc!def!ghi!"
match = /.+!/.match(string)
puts match[0]          ←—— Dane wyjściowe: abc!def!ghi!
```

Zażądano co najmniej jednego znaku (używając kropki symbolu wieloznacznego), po którym następuje wykrzyknik. Możesz oczekiwać uzyskania podłańcucha "abc!", który odpowiada temu opisowi.

Zamiast niego otrzymujemy łańcuch `"abc!def!ghi!"`. Kwantyfikator + zachłannie „zjada" jak największą możliwą część łańcucha i kończy działanie dopiero na *ostatnim* wykrzykniku, a nie na pierwszym.

Aby wyeliminować zachłanność kwantyfikatorów + i *, należy wstawić za nimi pytajnik. Zaobserwuj, co się stanie, gdy takie działanie zostanie wykonane dla ostatniego przykładu:

```
string = "abc!def!ghi!"
match = /.+?!/.match(string)
puts match[0]          ←——— Dane wyjściowe: abc!
```

Ta wersja kodu przekazuje następującą informację: „Zapewnij mi jeden lub więcej znaków wieloznacznych, ale tylko taką ich liczbę, jaka jest widoczna do miejsca wystąpienia *pierwszego* wykrzyknika, który również powinien zostać uwzględniony". Z całą pewnością tym razem uzyskamy łańcuch `"abc!"`.

Jeśli w przykładzie z cyframi do kwantyfikatora dodamy pytajnik, działanie zakończy się po napotkaniu cyfry 2:

```
/(\d+?)/.match("Cyfry-To-My 2345")
puts $1
```

W tym przypadku dane wyjściowe to cyfra `2`.

Co oznacza stwierdzenie, że zachłanne kwantyfikatory zapewniają jak największą możliwą liczbę znaków, co „odpowiada zezwoleniu na dopasowanie reszty wzorca"?

Rozważ następujące dopasowanie:

```
/\d+5/.match("Cyfry-To-My 2345")
```

Jeśli zachłanność kwantyfikatora „jeden lub więcej" byłaby absolutna, wzorzec `\d+` dopasowałby wszystkie cztery cyfry. Później cyfra `5` we wzorcu nie dopasowałaby niczego, dlatego całe dopasowanie zakończyłoby się niepowodzeniem. Jednak zachłanność zawsze podporządkowuje się zapewnieniu pomyślnego dopasowania. W tym przypadku po niepowodzeniu dopasowania mechanizm wyrażeń regularnych wycofuje się: eliminuje dopasowanie cyfry 5 i ponownie wypróbowuje wzorzec. Tym razem kończy się to powodzeniem: zostało spełnione zarówno wymaganie wzorca `\d+` (w postaci cyfr `234`), jak i wymaganie określające, że cyfra 5 następuje po cyfrach dopasowanych przez wzorzec `\d+`.

Ponownie uzyskujesz bogaty w informacje zapis przebiegu działań przez przechwytywanie części dopasowanego łańcucha i sprawdzanie tego, co zostało przechwycone. Narzędzie irb i obiekt `MatchData` umożliwiają uzyskanie następujących informacji o odpowiednich przechwyceniach:

```
>> /(\d+)(5)/.match("Cyfry-To-My 2345")
=> #<MatchData "2345" 1:"234" 2:"5">
```

Pierwsze przechwycenie to `"234"`, a drugie to `"5"`. Choć jest „zachłanny", kwantyfikatorowi „jeden lub więcej" wystarczyło uzyskanie tylko trzech cyfr, a nie czterech, co wynika z umożliwiania mechanizmowi wyrażeń regularnych znalezienia sposobu na dopasowanie łańcucha przez cały wzorzec.

Oprócz użycia modyfikatorów „jeden lub więcej albo żaden" lub „jeden lub więcej" możliwe jest też zażądanie dokładnej liczby lub zakresu z liczbami powtórzeń danego podwzorca.

### KONKRETNA LICZBA POWTÓRZEŃ

Aby dokładnie określić, ile ma zostać dopasowanych powtórzeń części wzorca, liczbę umieść w nawiasach klamrowych ({}) tuż za odpowiednim podwyrażeniem. Prezentuje to następujący przykład:

```
/\d{3}-\d{4}/
```

W przykładzie dopasowywane są dokładnie trzy cyfry, łącznik, a następnie cztery cyfry, czyli sekwencja 555-1212 oraz inne przypominające numer telefonu.

Możliwe jest też podanie zakresu w nawiasach klamrowych:

```
/\d{1,10}/
```

W tym przykładzie dopasowywany jest każdy łańcuch zawierający od 1 do 10 kolejnych cyfr. Pojedyncza liczba, po której następuje przecinek, interpretowana jest jako minimum (*n* lub więcej powtórzeń). Oznacza to, że możesz dopasować „trzy lub więcej cyfr" w następujący sposób:

```
/\d{3,}/
```

Mechanizm wyrażeń regularnych języka Ruby jest wystarczająco „inteligentny", aby poinformować, gdy użyty zakres jest niemożliwy. Jeśli na przykład spróbujesz dopasować liczbę wystąpień podwzorca reprezentowanych przez zakres `{10,2}` (co najmniej 10, lecz nie więcej niż 2), zostanie wygenerowany błąd krytyczny.

Liczbę powtórzeń możesz określić nie tylko dla pojedynczych znaków lub klas znaków, ale też dla dowolnego *atomu* (bardziej techniczny termin odnoszący się do „części wzorca") wyrażenia regularnego. Atomy obejmują podwzorce w nawiasach okrągłych i klasy znaków, a także pojedyncze znaki. Dlatego możesz użyć następującego wiersza, aby dopasować pięć kolejnych dużych liter:

```
/([A-Z]){5}/.match("Dawid NOWAK")
```

W tego typu sytuacjach trzeba być świadomym istotnej, potencjalnej pułapki.

### OGRANICZENIE DOTYCZĄCE NAWIASÓW OKRĄGŁYCH

Jeśli wykonasz ostatni wiersz powyższego kodu i przyjrzysz się przekazywanym przez obiekt `MatchData` informacjom o pierwszym przechwyceniu, możesz oczekiwać, że ujrzysz łańcuch `"NOWAK"`. Tak jednak nie jest:

```
>> /([A-Z]){5}/.match("Dawid NOWAK")
=> #<MatchData "NOWAK" 1:"K">
```

Pojawia się jedynie łańcuch `"K"`. Dlaczego w całości nie jest przechwytywany łańcuch `"NOWAK"`?

Wynika to stąd, że nawiasy okrągłe nie „wiedzą", że są powtarzane pięciokrotnie. „Wiedzą" jedynie, że są pierwszymi nawiasami od lewej strony (w tym konkretnym przypadku) i że to, co zostało przez nie przechwycone, powinno być umieszczone w pozycji pierwszego przechwycenia (zmienna `$1` lub metoda `captures[1]` obiektu

`MatchData`). Wyrażenie `[A-Z]` wewnątrz nawiasów okrągłych może dopasować tylko jeden znak. Jeśli wyrażenie to dopasuje jeden znak pięć razy pod rząd, nadal w danym momencie zostanie przez nie dopasowany tylko jeden znak, a ponadto „zapamięta" ono jedynie ostatni znak.

Inaczej mówiąc, pięciokrotne dopasowywanie jednego znaku nie jest tym samym co jednoczesne dopasowanie pięciu znaków.

W celu przechwycenia wszystkich pięciu znaków musisz tak przesunąć nawiasy okrągłe, aby obejmowały całe pięcioczęściowe dopasowanie:

```
>> /([A-Z]{5})/.match("Dawid NOWAK")
=> #<MatchData "NOWAK" 1:"NOWAK">
```

Przy stwierdzaniu, co zostanie przechwycone, zachowaj ostrożność i nie daj się ponieść wyobraźni.

Jako następnym przyjrzymy się sposobom określania żądanych warunków występowania dopasowań zamiast oczekiwanej zawartości łańcucha.

### 11.5.3. Zakotwiczenia i asercje wyrażeń regularnych

Asercje i zakotwiczenia to różne typy konstrukcji uzyskiwanych ze znaków. W przypadku dopasowywania znaku (nawet na podstawie klasy znaków lub symbolu wieloznacznego) mówi się, że zostanie *wykorzystany* znak w dopasowywanym łańcuchu. Z kolei asercja lub zakotwiczenie nie powoduje wykorzystania żadnych znaków. Zamiast tego wyrażają one *ograniczenie*, czyli warunek, który musi zostać spełniony przed zezwoleniem na kontynuowanie operacji dopasowywania znaków.

Najczęściej spotykane zakotwiczenia to *początek wiersza* (`^`) i *koniec wiersza* (`$`). Zakotwiczenia początku wiersza możesz użyć do wykonania takiego zadania jak usuwanie z pliku programu Ruby wszystkich wierszy komentarza. Zadanie to może zostać zrealizowane przez przetworzenie wszystkich wierszy w pliku i wyświetlenie tylko tych, które *nie* zaczynały się znakiem # lub białym znakiem z następującym po nim znakiem #. Aby określić, jakie wiersze są wierszami komentarza, możesz zastosować następujące wyrażenie regularne:

```
/^\s*#/
```

Znak ^ w tym wzorcu *zakotwicza* dopasowanie na początku wiersza. Jeśli reszta wzorca zostanie dopasowana, lecz *nie* na początku wiersza, nie zostanie to uwzględnione, co potwierdza kilka następujących testów:

```
>> comment_regexp = /^\s*#/
=> /^\s*#/
>> comment_regexp.match("  # Sam komentarz!")
=> #<MatchData "  #">
>> comment_regexp.match("  x = 1  # Kod oraz komentarz!")
=> nil
```

Dopasowaniem wzorca komentarza jest tylko wiersz rozpoczynający się białym znakiem i znakiem #. Drugi z powyższych wierszy nie będzie zgodny z wzorcem, dlatego nie zostałby usunięty, jeśli takie wyrażenie regularne zostałoby użyte do filtrowania komentarzy w pliku.

W tabeli 11.1 pokazano kilka zakotwiczeń, w tym początek i koniec wiersza oraz początek i koniec łańcucha.

**Tabela 11.1.** Zakotwiczenia wyrażeń regularnych

| Notacja | Opis | Przykład | Przykładowy dopasowany łańcuch |
|---|---|---|---|
| ^ | Początek wiersza | /^\s*#/ | " # Wiersz komentarza języka Ruby z początkowymi spacjami" |
| $ | Koniec wiersza | /\.$/ | "jeden\ndwa\ntrzy.\ncztery" |
| \A | Początek łańcucha | /\AOcena Cztery/ | "Ocena Cztery" |
| \z | Koniec łańcucha | /z ziemi.\z/ | "z ziemi." |
| \Z | Koniec łańcucha (z wyjątkiem ostatniego znaku nowego wiersza) | /z ziemi.\Z/ | "z ziemi\n" |
| \b | Granica słowa | /\b\w+\b/ | "!!!wyraz***" (dopasowuje łańcuch wyraz) |

Zauważ, że zakotwiczenie `\z` dopasowuje bezwzględny koniec łańcucha, natomiast zakotwiczenie `\Z` dopasowuje koniec łańcucha, z wyjątkiem opcjonalnego, końcowego znaku nowego wiersza. Zakotwiczenie `\Z` przydaje się w sytuacjach, w których nie masz pewności, czy łańcuch zawiera na końcu znak nowego wiersza (być może jest to ostatni wiersz wczytywany z pliku tekstowego), a ponadto nie chcesz się tym zajmować.

Zakotwiczeniom towarzyszą *asercje*, które podobnie jak one informują procesor wyrażeń regularnych o tym, że dopasowanie ma zostać uwzględnione tylko w przypadku wystąpienia określonych warunków.

### ASERCJE WYPRZEDZAJĄCE

Załóżmy, że chcesz dopasować sekwencję liczb tylko wtedy, gdy jest ona zakończona kropką. Nie chcesz jednak, aby sama kropka była traktowana jako część dopasowania.

Jednym ze sposobów umożliwiających to jest użycie *asercji wyprzedzającej* lub, posługując się kompletnym terminem, pozytywnej asercji wyprzedzającej o szerokości zerowej. Sposób zostanie poniżej szerzej objaśniony, a oto prezentujący go przykład:

```
str = "123 456. 789"
m = /\d+(?=\.)/.match(str)
```

W tym przypadku `m[0]` (reprezentuje cały łańcuch dopasowany przez wzorzec) zawiera `456`, czyli jedną sekwencję liczb, po których następuje kropka.

Oto trochę obszerniejszy komentarz użytej terminologii:

- *Szerokość zerowa* oznacza, że nie są wykorzystywane dowolne znaki łańcucha. Choć stwierdzana jest obecność kropki, w dalszym ciągu możliwe jest dopasowanie jej, jeśli wzorzec jest kontynuowany.
- *Pozytywna* oznacza, że wymagane jest przewidzenie obecności kropki. Istnieją też *negatywne* asercje wyprzedzające, w których zamiast konstrukcji `(?=...)` używana jest konstrukcja `(?!...)`.

- *Asercja wyprzedzająca* oznacza, że możesz określić, co *byłoby* następne, bez dopasowywania tego.

W przypadku stosowania asercji wyprzedzającej nie są uwzględniane nawiasy okrągłe, w których umieszczono część dopasowania związaną z asercją wyprzedzającą. W przykładzie zmienna `$1` nie zostanie ustawiona przez operację dopasowania. Ponadto kropka po liczbie 6 nie zostanie wykorzystana przez dopasowanie. Miej na uwadze tę ostatnią kwestię, jeśli kiedykolwiek poczujesz się zaintrygowany działaniem asercji wyprzedzającej. Konsternacja może często wynikać z zapomnienia, że wyprzedzanie nie jest tym samym co przechodzenie dalej.

### ASERCJE WSTECZNE

Asercje wyprzedzające mają odpowiedniki w wariancie wstecznym. Oto wyrażenie regularne dopasowujące łańcuch `NOWAK` tylko wtedy, gdy jest on poprzedzony łańcuchem `Dawid`:

```
re = /(?<=Dawid )NOWAK/
```

W odwrotnym wariancie zostałoby zastosowane wyrażenie regularne dopasowujące łańcuch `NOWAK` tylko wtedy, gdy nie jest on poprzedzony łańcuchem `Dawid`:

```
re = /(?<!Dawid )NOWAK/
```

I tym razem pamiętaj o tym, że są to asercje o szerokości zerowej. Reprezentują one ograniczenia dotyczące łańcucha (*łańcuch* `Dawid` *musi występować przed łańcuchem* `NOWAK`*, który w przeciwnym razie nie zostanie uznany za dopasowanie*), ale nie dopasowują ani nie wykorzystują żadnych znaków.

**Nawiasy okrągłe bez przechwytywania**

W celu dopasowania czegoś (nie tylko stwierdzenia, że występuje to jako następne, ale faktycznego dopasowania) za pomocą nawiasów okrągłych, lecz bez traktowania tego jako jednego z numerowanych przechwyceń w nawiasach okrągłych wynikających z dopasowania, użyj konstrukcji `(?:...)`. Wszystko, co znajduje się wewnątrz konstrukcji grupującej `(?:)`, będzie dopasowywane na podstawie grupy, lecz nie zapisywane w przechwyceniu. Zauważ, że obiekt `MatchData`, będący wynikiem poniższej operacji dopasowania, zawiera tylko dwa przechwycenia. Grupa `def` nie jest uwzględniana z powodu notacji `?:`:

```
>> str = "abc def ghi"
=> "abc def ghi"
>> m = /(abc) (?:def) (ghi)/.match(str)
=> #<MatchData "abc def ghi" 1:"abc" 2:"ghi">
```

W przeciwieństwie do asercji o szerokości zerowej grupa `(?:)` wykorzystuje znaki. Po prostu nie zapisuje ona ich jako przechwycenia.

Istnieje też coś takiego jak *dopasowanie warunkowe*.

### DOPASOWANIA WARUNKOWE

Choć prawdopodobnie nie będą one stanowić części codziennych działań związanych z wyrażeniami regularnymi, godne uwagi jest istnienie dopasowań warunkowych w mechanizmie wyrażeń regularnych języka Ruby 2.0 (projekt o nazwie Onigmo).

Dopasowanie warunkowe sprawdza konkretne przechwycenie (za pomocą liczby lub nazwy) i dopasowuje jedno z dwóch podwyrażeń na podstawie tego, czy zostało znalezione przechwycenie.

Poniżej zaprezentowano prosty przykład. Wyrażenie warunkowe `(?(1)b|c)` dopasowuje literę `b` w przypadku dopasowania przechwycenia o numerze `1`. W przeciwnym razie dopasowywana jest litera `c`:

```
>> re = /(a)?(?(1)b|c)/
=> /(a)?(?(1)b|c)/
>> re.match("ab")
=> #<MatchData "ab" 1:"a">  ←——❶
>> re.match("b")
=> nil        ←——❷
>> re.match("c")
=> #<MatchData "c" 1:nil>   ←——❸
```

Wyrażenie regularne `re` dopasowuje łańcuch `"ab"` ❶ z łańcuchem `"a"` jako pierwszym przechwyceniem w nawiasach okrągłych i podwyrażeniem warunkowym dopasowującym łańcuch `"a"`. Jednak wyrażenie regularne `re` nie dopasowuje łańcucha `"b"` ❷. Ponieważ nie ma pierwszego przechwycenia w nawiasach okrągłych, podwyrażenie warunkowe próbuje dopasować łańcuch `"c"`, co kończy się niepowodzeniem ❸. Z tego też powodu wyrażenie regularne `re` *dopasowuje* łańcuch `"c"`: warunek `(?(1)...)` nie jest spełniony, dlatego wyrażenie próbuje dopasować część samego siebie (odpowiadającą instrukcji `else`), którą jest podwyrażenie `/c/`.

Możliwe jest też tworzenie warunkowych wyrażeń regularnych za pomocą przechwyceń nazwanych. Wyrażenie z poprzedniego przykładu miałoby następującą postać:

```
/(?<first>a)?(?(<first>)b|c)/
```

Wyniki różnych dopasowań byłyby takie same.

Zakotwiczenia, asercje i dopasowania warunkowe zapewniają językowi wzorców bogactwo i szczegółowość. Język ten umożliwia wyrażanie szukanych dopasowań. Do kategorii elementów wzbogacających język zaliczają się też modyfikatory wyrażeń regularnych.

### *11.5.4. Modyfikatory*

Modyfikator wyrażenia regularnego jest literą umieszczoną po końcowym, domykającym ukośniku literału wyrażenia regularnego:

```
/abc/i
```

Zaprezentowany tutaj modyfikator `i` powoduje, że operacje dopasowania obejmujące to wyrażenie regularne rozróżniają wielkość znaków. Inny najczęściej spotykany modyfikator to `m` (ang. *multiline*). Sprawia on, że znak kropki wieloznacznej, który standardowo dopasowuje *każdy znak, z wyjątkiem znaku nowego wiersza*, dopasuje *dowolny znak, w tym znak nowego wiersza*. Jest to przydatne, gdy ma zostać przechwycone wszystko, co znajduje się na przykład między nawiasem otwierającym i zamykającym, a ponadto nie wiesz (lub nie przejmujesz się tym), czy są one w tym samym wierszu.

W następującym przykładzie zwróć uwagę na znaki nowego wiersza (`\n`) osadzone w łańcuchu:

```
str = "Pobiera (w tym to,\nco jest zawarte w nawiasach okrągłych\n) maksymalnie trzy wiersze."
m = /\(.*?\)/m.match(str)
```

„Niezachłanny" podwzorzec wieloznaczny `.*?` dopasowuje:

```
(w tym to,\nco jest zawarte w nawiasach okrągłych\n)
```

Bez modyfikatora `m` kropka w podwzorcu nie dopasowałaby znaków nowego wiersza. Po napotkaniu pierwszego znaku nowego wiersza i nieznalezieniu w tym przypadku znaku `)` operacja dopasowania zakończyłaby się niepowodzeniem.

Inny często używany modyfikator wyrażeń regularnych to `x`. Zmienia on sposób, w jaki analizator wyrażeń regularnych traktuje biały znak. Zamiast uwzględniać go w takiej postaci we wzorcu, modyfikator ignoruje biały znak, chyba że jego znaczenie zostanie zmienione za pomocą lewego ukośnika. Celem modyfikatora `x` jest umożliwienie dodawania komentarzy do wyrażeń regularnych:

```
/
 \((\d{3})\)  # 3 cyfry wewnątrz dosłownych nawiasów okrągłych (kierunkowy)
   \s         # Jeden znak spacji
 (\d{3})      # 3 cyfry (kierunkowy)
    -         # Łącznik
 (\d{4})      # 4 cyfry (druga część numeru
/x
```

Poprzednie wyrażenie regularne jest dokładnie takie samo jak to, z tym że z rozszerzoną składnią i komentarzami:

```
/\((\d{3})\)\s(\d{3})-(\d{4})/
```

Zachowaj ostrożność w przypadku modyfikatora `x`. Gdy odkryjesz go po raz pierwszy, pojawi się pokusa, aby wszystkie wzorce miały naprawdę sporą szerokość:

```
/  (?<=   Dawid\ )    NOWAK  /x
```

Zwróć uwagę na znak spacji — jego znaczenie zmieniono za pomocą lewego ukośnika. Jest to jedyny taki znak, który będzie uważany za część wzorca. Pamiętaj jednak, że wielu programistów nauczyło się analizować wyrażenia regularne bez umieszczonych w nich dodatkowych białych znaków, które rzekomo są przyjazne dla użytkownika. Jeśli przywykłeś do standardowej składni, nie będzie łatwe wyeliminowanie modyfikatora `x` z analizowanego wyrażenia regularnego.

Przeważnie modyfikator `x` najlepiej zastosować w sytuacjach, w których wyrażenie regularne ma zostać podzielone na wiele wierszy z myślą o dodaniu komentarzy, tak jak w przykładzie numeru telefonu. Nie zakładaj, że biały znak automatycznie zwiększy czytelność wyrażeń regularnych.

W dalszej kolejności przyjrzymy się technikom dokonywania konwersji między dwoma różnymi, lecz blisko ze sobą związanymi klasami: `String` i `Regexp`.

## 11.6. Wzajemna konwersja łańcuchów i wyrażeń regularnych

To, że wyrażenia regularne nie są łańcuchami, można z łatwością od razu stwierdzić w przypadku wyrażeń regularnych podobnych do następującego:

```
/[a-c]{3}/
```

Biorąc pod uwagę jego specjalną składnię powtórzeń i klasy znaków, powyższy wzorzec nie przypomina zbytnio żadnego z dopasowywanych przez niego łańcuchów (`"aaa"`, `"aab"`, `"aac"` itd.).

W przypadku wyrażenia regularnego podobnego do następującego trochę trudniejsze staje się *niedostrzeżenie* bezpośredniego powiązania między wyrażeniem regularnym i łańcuchem:

```
/abc/
```

To wyrażenie regularne nie jest łańcuchem `"abc"`. Co więcej, dopasowuje ono nie tylko ten łańcuch, ale dowolny łańcuch ze znajdującym się gdzieś w jego obrębie podłańcuchem `"abc"` (np. w zdaniu „Teraz znam moje abcs"). Nie występuje żadna unikatowa relacja między łańcuchem i podobnie wyglądającym wyrażeniem regularnym.

Choć wizualne podobieństwo między niektórymi łańcuchami i wyrażeniami regularnymi nie oznacza, że są tym samym, to jednak zachodzi między nimi interakcja na różne znaczące sposoby. Przyjrzyjmy się przepływowi odbywającemu się w kierunku od łańcucha do wyrażenia regularnego, a następnie przepływowi w przeciwnym kierunku.

### 11.6.1. Idiomy związane z przepływem od łańcucha do wyrażenia regularnego

Najpierw możesz przeprowadzić interpolację łańcucha (lub w stylu łańcuchowym) wewnątrz wyrażenia regularnego. Umożliwia to znajoma już technika interpolacji oparta na konstrukcji `#{...}`:

```
>> str = "def"
=> "def"
>> /abc#{str}/
=> /abcdef/
```

Wartość zmiennej `str` jest umieszczana w wyrażeniu regularnym i ustanawiana jego częścią, tak jak miałoby to miejsce w przypadku użycia tej samej techniki do interpolacji wartości do postaci łańcucha.

Technika interpolacji staje się bardziej złożona, gdy interpolowany łańcuch zawiera znaki specjalne wyrażenia regularnego. Dla przykładu rozważ łańcuch zawierający kropkę. Jak wiesz, znak kropki ma specjalne znaczenie w wyrażeniach regularnych: kropka dopasowuje dowolny pojedynczy znak, z wyjątkiem znaku nowego wiersza. W łańcuchu jest to po prostu kropka. W przypadku interpolacji łańcuchów do postaci wyrażeń regularnych może to być przyczyną niejasności:

```
>> str = "a.c"
=> "a.c"
>> re = /#{str}/
=> /a.c/
```

```
>> re.match("a.c")
=> #<MatchData "a.c">
>> re.match("abc")
=> #<MatchData "abc">
```

Obie operacje dopasowania kończą się pomyślnie. Zwracają one obiekty `MatchData`, a nie obiekt `nil`. Kropka we wzorcu dopasowuje kropkę w łańcuchu `"a.c"`. Dopasowuje jednak też literę `b` w łańcuchu `"abc"`. Kropka, która początkowo była jedynie kropką w zmiennej `str`, nabiera specjalnego znaczenia, gdy staje się częścią wyrażenia regularnego.

Możesz jednak *zmienić znaczenie* znaków specjalnych wewnątrz łańcucha przed umieszczeniem go w wyrażeniu regularnym. Nie musisz tego robić ręcznie: klasa `Regexp` zapewnia metodę klasy `Regexp.escape`, która się tym zajmuje. Efekty działania tej metody możesz sprawdzić, uruchamiając ją dla kilku wyizolowanych łańcuchów:

```
>> Regexp.escape("a.c")
=> "a\\.c"
>> Regexp.escape("^abc")
=> "\\^abc"
```

Narzędzie irb podwaja lewe ukośniki, ponieważ zwraca łańcuchy w cudzysłowie. W razie potrzeby dla wyrażeń możesz zastosować instrukcję `puts`, dzięki czemu ujrzysz je w ich rzeczywistej postaci z pojedynczymi lewymi ukośnikami.

W wyniku tego rodzaju operacji zmiany znaczenia możesz ograniczać wyrażenia regularne w celu dopasowania dokładnie takich łańcuchów, jakie są uzyskiwane po przeprowadzeniu interpolacji:

```
>> str = "a.c"
=> "a.c"
>> re = /#{Regexp.escape(str)}/
=> /a\.c/
>> re.match("a.c")
=> #<MatchData "a.c">
>> re.match("abc")
=> nil
```

Tym razem próba użycia kropki jako wieloznacznego znaku dopasowania kończy się niepowodzeniem. Łańcuch `"abc"` nie stanowi dopasowania dla poddanego interpolacji łańcucha o zmienionym znaczeniu.

Możliwe jest również utworzenie instancji wyrażenia regularnego przy użyciu łańcucha przez przekazanie go metodzie `Regexp.new`:

```
>> Regexp.new('(.*)\s+Nowak')
=> /(.*)\s+Nowak/
```

Zastosowanie ma zwykła logika zmiany znaczenia znaku i/lub wyrażenia regularnego:

```
>> Regexp.new('prof\. Dawid Nowak')
=> /prof\. Dawid Nowak/
>> Regexp.new(Regexp.escape("prof. Dawid Nowak"))
=> /prof\.\ Dawid\ Nowak/
```

Zauważ, że za pomocą lewych ukośników zmieniono znaczenie dosłownych znaków spacji, co nie jest bezwzględnie konieczne — chyba że używasz modyfikatora `x` — ale też nie jest szkodliwe.

Możesz też przekazać metodzie `Regexp.new` dosłowne wyrażenie regularne, co w tym przypadku powoduje uzyskanie nowego, identycznego wyrażenia regularnego. Ponieważ zawsze możesz najpierw użyć dosłownego wyrażenia regularnego, metoda ta jest częściej stosowana do przekształcania łańcuchów w wyrażenia regularne.

Zastosowanie łańcuchów w apostrofach eliminuje konieczność podwajania lewych ukośników. Jeśli korzystasz z cudzysłowów (co może być niezbędne, zależnie od rodzajów wymaganych interpolacji), pamiętaj, że musisz utworzyć konstrukcję `prof\\.` tak, aby lewy ukośnik był częścią łańcucha przekazywanego konstruktorowi wyrażenia regularnego. W przeciwnym razie zakończy się to jedynie umieszczeniem w łańcuchu samej kropki (co i tak by się stało), która trafi do wyrażenia regularnego bez ukośnika. W efekcie zostanie ona zinterpretowana jako kropka symbolu wieloznacznego.

Przyjrzyjmy się teraz kilku technikom konwersji przeprowadzanej w przeciwnym kierunku, czyli od wyrażenia regularnego do łańcucha. Coś takiego będzie przeważnie stosowane na potrzeby analizy i debugowania.

### 11.6.2. Przejście od wyrażenia regularnego do łańcucha

Podobnie jak wszystkie obiekty języka Ruby wyrażenia regularne mogą się same reprezentować w postaci łańcucha. Sposób realizowania tego działania może początkowo wyglądać dziwnie:

```
>> puts /abc/
(?-mix:abc)
```

Jest to alternatywna notacja wyrażenia regularnego, która rzadko „widuje światło dzienne". Wyjątkiem są sytuacje, w których jest ona generowana przez metodę instancji `to_s` obiektów wyrażenia regularnego. To, co wygląda jak słowo *mix*, jest listą modyfikatorów (`m`, `i` oraz `x`) z umieszczonym na początku znakiem minus, który wskazuje, że wszystkie modyfikatory są wyłączone.

W narzędziu irb możesz poeksperymentować z użyciem instrukcji `puts` z wyrażeniami regularnymi. Dzięki temu dowiesz się więcej o sposobie działania tej notacji. Nie będziemy się tutaj tym zajmować, po części dlatego, że istnieje inny sposób uzyskiwania reprezentacji łańcuchowej wyrażenia regularnego. Polega on na wywołaniu metody `inspect` lub metody `p` (która z kolei wywołuje metodę `inspect`):

```
>> /abc/.inspect
=> "/abc/"
```

Przejście z wyrażeń regularnych do łańcuchów jest przydatne głównie wtedy, gdy analizujesz wyrażenia regularne i/lub rozwiązujesz związane z nimi problemy. Jest to dobry sposób na upewnienie się, że wyrażenia regularne są tym, czego w ich przypadku oczekujesz.

Na tym etapie „zatoczymy pełne koło" odnośnie do wyrażeń regularnych, omawiając role, jakie pełnią one w przypadku kilku istotnych metod innych klas. Do tej pory używaliśmy prawie wyłącznie metody `match`, która jest dopiero początkiem.

## 11.7. Typowe metody używające wyrażeń regularnych

Korzyścią wynikającą z zapewnienia obsługi wyrażeń regularnych w języku Ruby jest możliwość zastosowania metod, które pobierają takie wyrażenia jako argumenty i je przetwarzają.

Na początek możesz zawsze użyć metody `match` w roli testu, na przykład w wykonywanej dla kolekcji operacji opartej na metodzie `find` lub `find_all`. Aby na przykład w tablicy łańcuchów o nazwie `array` znaleźć wszystkie łańcuchy liczące więcej niż 10 znaków i zawierające co najmniej jedną cyfrę, możesz zastosować następujący kod:

```
array.find_all {|e| e.size > 10 and /\d/.match(e) }
```

Jednak kilka metod, powiązanych głównie z łańcuchami, bardziej bezpośrednio bazuje na wykorzystaniu wyrażeń regularnych. W niniejszym podrozdziale przyjrzyjmy się kilku z nich.

### 11.7.1. Metoda String#scan

Metoda `scan` analizuje łańcuch od lewej do prawej strony, sprawdzając wielokrotnie istnienie dopasowania dla podanego wzorca. Wyniki są zwracane w tablicy.

Aby na przykład uzyskać wszystkie cyfry w łańcuchu, możesz użyć następującego kodu:

```
>> "testowanie 1 2 3 testowanie 4 5 6".scan(/\d/)
=> ["1", "2", "3", "4", "5", "6"]
```

Zauważ, że metoda `scan` pomija to, co jest niezgodne z jej wzorcem, i szuka dopasowania w dalszej części łańcucha. Takie zachowanie jest odmienne niż w przypadku metody `match`, która całkowicie kończy działanie po jednokrotnym ukończeniu pełnego dopasowywania wzorca.

Jeśli w wyrażeniu regularnym przekazywanym metodzie `scan` użyjesz grupowań w nawiasach okrągłych, operacja zwróci tablicę tablic. Każda tablica wewnętrzna zawiera wyniki jednego skanowania łańcucha:

```
>> str = "Leopold Auer był nauczycielem skrzypka Jascha Heifetz."
=> "Leopold Auer był nauczycielem skrzypka Jascha Heifetz."
>> violinists = str.scan(/([A-Z]\w+)\s+([A-Z]\w+)/)
=> [["Leopold", "Auer"], ["Jascha", "Heifetz"]]
```

W tym przykładzie zagnieżdżana jest tablica tablic, w której każda tablica wewnętrzna zawiera imię i nazwisko osoby. Przechowywanie każdego imienia i nazwiska w osobnej tablicy ułatwia wykonywanie iteracji dla całej listy imion i nazwisk, która została umieszczona w zmiennej `violinists`:

```
violinists.each do |fname,lname|
  puts "Imię dla #{lname} to #{fname}."
end
```

Dane wyjściowe dla tego fragmentu kodu są następujące:

```
Imię dla Auer to Leopold.
Imię dla Heifetz to Jascha.
```

Wyrażenie regularne używane dla imion i nazwisk w tym przykładzie jest oczywiście nazbyt uproszczone: pomija ono łączniki, drugie imiona itp. Ilustruje jednak odpowiednio sposób użycia przechwyceń w przypadku metody `scan`.

Metoda `String#scan` może też pobrać blok kodu. Technika ta umożliwia czasem wyeliminowanie kroku. Metoda `scan` przekazuje swoje wyniki do bloku, a szczegóły tego działania zależą od tego, czy używasz przechwyceń w nawiasach okrągłych. Oto modyfikacja poprzedniego kodu bazująca na bloku skanowania:

```
str.scan(/([A-Z]\w+)\s+([A-Z]\w+)/) do |fname, lname|
  puts "#Imię dla {lname} to #{fname}."
end
```

Przy każdym przetwarzaniu łańcucha blok odbiera przechwycenia w tablicy. Jeśli nie jest wykonywana operacja przechwytywania, blok odbiera kolejno dopasowane podłańcuchy. Kod skanowania grup znaków `\w` (jest to klasa znaków złożona z liter, liczb i znaku podkreślenia) może mieć następującą postać:

```
"jeden dwa trzy".scan(/\w+/) {|n| puts "Następna liczba: #{n}" }
```

Kod ten wygeneruje następujące dane wyjściowe:

```
Następna liczba: jeden
Następna liczba: dwa
Następna liczba: trzy
```

Zauważ, że jeśli podajesz blok, metoda `scan` nie przechowuje wyników w tablicy i nie zwraca ich. Każdy wynik jest przez nią wysyłany do bloku, a następnie usuwany. Dzięki temu możesz skanować długie łańcuchy, przetwarzając w ramach tego wyniki w dowolny sposób i unikając zajmowania pamięci podłańcuchami, które zostały już znalezione i użyte.

Metoda `split` reprezentuje kolejną typową operację łańcuchową opartą na wyrażeniach regularnych.

### *11.7.2. Metoda String#split*

Jak wskazuje nazwa, metoda `split` (dziel) dokonuje podziału łańcucha na wiele podłańcuchów, zwracając je w postaci tablicy. Metoda ta może pobrać wyrażenie regularne lub zwykły łańcuch jako separator dla operacji podziału. Metoda jest często używana do uzyskiwania tablicy złożonej z wszystkich znaków łańcucha. W tym celu zastosuj następujące puste wyrażenie regularne:

```
>> "Ruby".split(//)
=> ["R", "u", "b", "y"]
```

Metoda `split` często jest wykorzystywana w ramach przekształcania zwykłych, tekstowych plików konfiguracyjnych w struktury danych języka Ruby. Zwykle obejmuje to przetworzenie pliku wiersz po wierszu oraz skonwertowanie każdego wiersza. Konwersja jednowierszowa może mieć następującą postać:

```
line = "first_name=dawid;last_name=nowak;country=pl"
record = line.split(/=|;/)
```

**Jeszcze więcej skanowania łańcuchów za pomocą klasy StringScanner**

Biblioteka standardowa zawiera rozszerzenie o nazwie `strscan`, które zapewnia klasę `StringScanner`. Obiekty tej klasy rozszerzają dostępny zestaw narzędzi pod kątem skanowania i sprawdzania łańcuchów. Obiekt `StringScanner` utrzymuje wskaźnik do łańcucha, umożliwiając przemieszczanie się w dwóch kierunkach w obrębie łańcucha z użyciem pozycji i semantyki wskaźnika.

Oto kilka przykładów metod klasy `StringScanner`:

```
>> require 'strscan'          ← Ładuje bibliotekę skanera
=> true
>> ss = StringScanner.new("Testowanie skanowania łańcucha")  ← Tworzy skaner
=> #<StringScanner 0/23 @ "Testo...">
>> ss.scan until(/wanie/)     ← Skanuje łańcuch do momentu dopasowania wyrażenia regularnego
=> "Testowanie"
>> ss.pos                     ← Sprawdza nową pozycję wskaźnika
=> 10
>> ss.peek(10)                ← Sprawdza 7 następnych bajtów (nie powoduje jednak przesunięcia wskaźnika)
=> " skanowani"
>> ss.unscan                  ← Wycofuje poprzednie skanowanie
=> #<StringScanner 0/23 @ "Testo...">
>> ss.pos
=> 0
>> ss.skip(/Test/)            ← Przesuwa wskaźnik za wyrażenie regularne
=> 4
>> ss.rest                    ← Sprawdza część łańcucha po prawej stronie wskaźnika
=> "owanie skanowania łańcucha"
```

Dzięki zastosowaniu pojęcia wskaźnika do łańcucha klasa `StringScanner` umożliwia przejście łańcucha, a także sprawdzenie, co zostało już dopasowane, a co pozostało. Klasa ta stanowi przydatne uzupełnienie wbudowanych rozwiązań służących do skanowania łańcuchów.

Operacja powoduje, że zmienna `record` zawiera następującą tablicę:

```
["first_name", "dawid", "last_name", "nowak", "country", "pl"]
```

Przy niewiele większym nakładzie pracy możesz zapełnić tablicę asocjacyjną wpisami następującego rodzaju:

```
data = []
record = Hash[*line.split(/=|;/)]   ← Użyj znaku * w celu przekształcenia tablicy w podstawową listę, która zostanie przekazana do Hash[ ]
data.push(record)
```

Jeśli działanie to wykonasz dla każdego wiersza w pliku, uzyskasz tablicę tablic asocjacyjnych reprezentujących wszystkie rekordy. Z kolei taka tablica może pełnić rolę bazy dla dalszej operacji, jaką być może będzie osadzanie informacji w raporcie lub przekazywanie ich procedurze biblioteki, która może je zapisać w tabeli bazy danych jako sekwencję par złożonych z kolumny i wartości.

Metodzie `split` możesz zapewnić drugi argument, który ogranicza liczbę zwracanych elementów. W następującym przykładzie:

```
>> "a,b,c,d,e".split(/,/,3)
=> ["a", "b", "c,d,e"]
```

metoda `split` kończy podział po uzyskaniu trzech elementów do zwrócenia i w trzecim łańcuchu umieszcza wszystko, co pozostało (przecinki i reszta elementów).

Oprócz dzielenia łańcucha na części przez skanowanie i podział możesz też zmieniać te części w ramach operacji zastępowania. Będzie o tym mowa w następnym punkcie.

### *11.7.3. sub/sub! i gsub/gsub!*

`sub` i `gsub` (wraz z ich odpowiednikami z wykrzyknikiem w nazwie) to najczęściej używane narzędzia służące do modyfikowania zawartości łańcuchów w języku Ruby. Różnica między nimi jest taka, że narzędzie `gsub` (global *sub*stitution) wprowadza zmiany w całym łańcuchu, natomiast narzędzie `sub` dokonuje co najwyżej jednego zastąpienia.

#### POJEDYNCZE ZASTĄPIENIA DOKONYWANE ZA POMOCĄ NARZĘDZIA SUB

Narzędzie `sub` pobiera dwa argumenty: wyrażenie regularne (lub łańcuch) oraz łańcuch zastępujący. Każda część (jeśli występuje) łańcucha dopasowana do wyrażenia regularnego jest usuwana z łańcucha i zastępowana łańcuchem zastępującym:

```
>> "błąd typograficzny".sub(/i/,"o")
=> "błąd typograficzny"
```

*Zamiast* argumentu łańcucha zastępującego możesz użyć bloku kodu. Blok jest wywoływany w przypadku wystąpienia dopasowania. Wywołanie przekazuje zastępowany łańcuch jako argument:

```
>> "pierwszą samogłoskę ustaw jako dużą literę".sub(/[aeiou]/) {|s| s.upcase }
=> "pIerwszą samogłoskę ustaw jako dużą literę"
```

Jeśli użyto jakiegokolwiek grupowania w nawiasach okrągłych, ustawiane są zmienne globalne `$n` i udostępniane są one w celu użycia wewnątrz bloku.

#### ZASTĄPIENIA GLOBALNE DOKONYWANE ZA POMOCĄ NARZĘDZIA GSUB

Narzędzie `gsub` różni się od narzędzia `sub` tylko tym, że zastąpienie utrzymuje do momentu dopasowania wzorca w dowolnym miejscu łańcucha. Na przykład następujący wiersz prezentuje sposób zastąpienia pierwszej litery każdego słowa łańcucha odpowiadającą jej dużą literą:

```
>> "ustaw dużą literę na początku każdego słowa".gsub(/\b\w/) {|s| s.upcase }
=> "Ustaw Dużą Literę Na Początku Każdego Słowa"
```

Podobnie jak narzędzie `sub` narzędzie `gsub` zapewnia dostęp do zmiennych przechwycenia w nawiasach okrągłych w bloku kodu.

#### UŻYCIE PRZECHWYCEŃ W ŁAŃCUCHU ZASTĘPUJĄCYM

Dostęp do przechwyceń w nawiasach okrągłych możesz uzyskać za pomocą specjalnej notacji złożonej z liczb o znaczeniu zmienionym przez lewy ukośnik. Na przykład w następujący sposób możesz poprawić wystąpienie małej litery, po której następuje duża litera (zakładając, że masz do czynienia z sytuacją, w której wystąpiła pomyłka):

```
>> "aDwid".sub(/([a-z])([A-Z])/, '\2\1')
=> "Dawid"
```

Zwróć uwagę na użycie apostrofów dla łańcucha zastępującego. W przypadku cudzysłowów konieczne byłoby podwojenie lewych ukośników w celu zmiany znaczenia znaku lewego ukośnika.

Aby zdublować każde słowo łańcucha, możesz zastosować wiersz podobny do następującego, który jednak zawiera narzędzie `gsub`:

```
>> "zdubluj każde słowo".gsub(/\b(\w+)/, '\1 \1')
=> "zdubluj zdubluj każde każde słowo słowo"
```

**Pułapka związana ze zmienną globalną przechwycenia**

Bądź świadom następującej rzeczy: zmienne globalne przechwycenia (`$1` itd.) mogą być używane w łańcuchu zastępującym, ale mogą one nie realizować tego, czego się po nich spodziewasz. Dokładniej rzecz biorąc, będziesz narażony na wartości tych zmiennych, które są pozostałościami. Przeanalizuj następujący przykład:

```
>> /(abc)/.match("abc")
=> #<MatchData "abc" 1:"abc">
>> "aDwid".sub(/([a-z])([A-Z])/, "#{$2}#{$1}")
=> "abcwid"
```

W tym przypadku zmienna `$1` z poprzedniego dopasowania (`"abc"`) zakończyła infiltrację łańcucha zastąpienia w drugim dopasowaniu. Ogólnie rzecz biorąc, pozostanie przy odwołaniach do przechwyceń w stylu `\1` jest bezpieczniejsze od stosowania zmiennych globalnych przechwycenia w łańcuchach zastąpienia narzędzi `sub` i `gsub`.

Omówienie narzędzi bazujących na wyrażeniach regularnych zakończymy przedstawieniem dwóch technik, których wspólnym elementem jest zależność od operatora równości przypadków (`===`). Pierwszą techniką są instrukcje przypadków (nie są one wywołaniami metody, ale stosują wywołania operatora `===`), a drugą techniką jest metoda `Enumerable#grep`.

### 11.7.4. Równość przypadków i metoda grep

Jak wiesz, wszystkie obiekty języka Ruby rozpoznają komunikat `===`. Jeśli nie zostanie on przesłonięty w danej klasie lub dla danego obiektu, jest on synonimem dla `==`. W przypadku przesłonięcia operator `===` będzie tym, czym ustanowi go nowa wersja.

W przypadku wyrażeń regularnych równość przypadków jest testem dopasowania: dla dowolnego danego wyrażenia regularnego i łańcucha równość *wyrażenie regularne* `===` *łańcuch* jest prawdziwa, gdy *łańcuch* jest zgodny z *wyrażeniem regularnym*. Operatora `===` możesz użyć jawnie jako testu dopasowania:

```
puts "Dopasowanie!" if re.match(string)
puts "Dopasowanie!" if string =~ re
puts "Dopasowanie!" if re === string
```

Oczywiście musisz zastosować taki test, który zapewni to, czego wymagasz: obiekt `nil` lub `MatchData` dla metody `match`, obiekt `nil` lub przesunięcie w postaci liczby całkowitej dla operatora `=~` oraz obiekt `true` lub `false` dla operatora `===`.

W instrukcjach przypadku operator `===` jest stosowany jawnie. Aby w instrukcjach przypadku przetestować różne dopasowania wzorca, użyj następujących wierszy:

```
print "Czy kontynuować? (t/n) "
answer = gets
case answer
when /^t/i
  puts "Znakomicie!"
when /^n/i
  puts "Żegnaj!"
  exit
else
  puts "Co takiego?"
end
```

Każda klauzula `when` jest wywołaniem metody `===`: `/^t/i === answer` itd.

Metoda `Enumerable#grep` to inna technika, która również niejawnie korzysta z metody/operatora `===`. Była o tym mowa w punkcie 10.3.3. W tym miejscu przybliżymy kilka aspektów związanych ze sposobem obsługi przez tę technikę łańcuchów i wyrażeń regularnych.

Metoda `grep` wykonuje operację filtrowania z poziomu obiektu wyliczeniowego na bazie operatora równości przypadków (`===`), zwracając wszystkie elementy w tym obiekcie, które zwracają wartość prawda w przypadku zastosowania operatora `===` dla argumentu metody `grep`. A zatem jeśli argumentem tej metody jest wyrażenie regularne, wybór oparty jest na dopasowaniach wzorca, zgodnie z działaniem metody `Regexp#===`:

```
>> ["USA", "UK", "Francja", "Niemcy"].grep(/[a-z]/)
=> ["Francja", "Niemcy"]
```

Choć to samo możesz uzyskać za pomocą metody `select`, jest to trochę bardziej dziwne:

```
["USA", "UK", "Francja", "Niemcy"].select {|c| /[a-z]/ === c }
```

Metoda `grep` używa uogólnionej techniki opartej na metodzie `===` do tego, aby zwięzłe i wygodne były specjalizowane operacje z wykorzystaniem metody `select`, które uwzględniają operacje dotyczące łańcuchów, lecz nie są do nich ograniczone.

Możliwe jest też przekazanie metodzie `grep` bloku kodu. W tym przypadku realizowana jest operacja łącząca w sobie metody `select` i `map`: wyniki operacji filtrowania są po jednym na raz dostarczane blokowi, a wartość zwracana całego wywołania metody `grep` ma postać skumulowanego wyniku tych operacji dostarczania. Aby na przykład wybrać kraje, a następnie zgromadzić je jako nazwy złożone z dużych liter, możesz użyć następującego wiersza:

```
>> ["USA", "UK", "Francja", "Niemcy"].grep(/[a-z]/) {|c| c.upcase }
=> ["FRANCJA", "NIEMCY"]
```

Pamiętaj o tym, że metoda `grep` dokonuje wyboru na podstawie operatora równości przypadków (`===`), dlatego nie wybierze niczego innego niż łańcuchy, gdy jako argument zostanie jej przekazane wyrażenie regularne. Ponadto nie występuje automatyczna konwersja między liczbami i łańcuchami. Oznacza to, że jeśli użyjesz następującego wiersza:

```
[1,2,3].grep(/1/)
```

to jako wynik uzyskasz pustą tablicę. Nie zawiera ona elementu łańcucha zgodnego z wyrażeniem regularnym `/1/`. W tablicy nie ma żadnego elementu, dla którego prawdziwe jest równanie `/1/ === element`.

W ten oto sposób kończymy przegląd wyrażeń regularnych i niektórych korzystających z nich metod. To nie koniec informacji, jakie należy przyswoić. Dopasowywanie wzorców to rozwlekłe zagadnienie. Jednak w rozdziale zaprezentowano sporo tego, czego prawdopodobnie będziesz wymagać, a także tego, z czym się spotkasz w trakcie kontynuowania nauki języka Ruby i używania go.

## 11.8. Podsumowanie

W rozdziale zaprezentowano następujące zagadnienia:

- Podstawowe zasady związane z dopasowywaniem wzorców wyrażeń regularnych.
- Techniki oparte na metodzie `match` i operatorze `=~`.
- Klasy znaków.
- Przechwycenia w nawiasach okrągłych.
- Kwantyfikatory.
- Zakotwiczenia.
- Obiekty `MatchData`.
- Interpolacja i konwersja łańcuchów/wyrażeń regularnych.
- Metody języka Ruby korzystające z wyrażeń regularnych: `scan`, `split`, `grep`, `sub` i `gsub`.

W rozdziale zaprezentowano fundamenty wyrażeń regularnych w języku Ruby, w tym klasy znaków, przechwycenia w nawiasach okrągłych i zakotwiczenia. Pokazano, że wyrażenia regularne to obiekty, a dokładniej rzecz biorąc, obiekty klasy `Regexp`, które odpowiadają na komunikaty (np. „dopasowania"). Omówiliśmy klasę `MatchData`, której instancje przechowują informacje o wynikach operacji metody `match`. Dowiedziałeś się również, jak interpolować łańcuchy do postaci wyrażeń regularnych (ze zmienionym znaczeniem lub nie, zależnie od tego, czy znaki specjalne w łańcuchu mają być traktowane jako szczególne w wyrażeniu regularnym), w jaki sposób tworzyć instancję wyrażenia regularnego na bazie łańcucha, a także jak wygenerować reprezentację łańcuchową wyrażenia regularnego.

Takie metody jak `String#scan`, `String#split` i `Enumerable#grep`, a także rodzina „podrzędna" metod klasy `String`, korzystają z wyrażeń regularnych i dopasowywania wzorców jako sposobu określania tego, jak powinny być stosowane ich działania. Zdobycie wiedzy na temat wyrażeń regularnych zapewnia dostęp nie tylko do stosunkowo prostych metod dopasowujących, ale też do pakietu narzędzi obsługujących łańcuchy, które w przeciwnym razie nie byłyby przydatne.

Kontynuując omawianie narzędzi wbudowanych języka Ruby, w rozdziale 12. zajmiemy się ogólnie zagadnieniem operacji wejścia-wyjścia, a dokładnie obsługą plików.

# 12 Operacje wejścia-wyjścia i operacje na plikach

**W tym rozdziale:**

- Dane wprowadzane z klawiatury i dane wyświetlane na ekranie.
- Klasy `IO` i `File`.
- Narzędzia plikowe biblioteki standardowej, w tym klasy `FileUtils` i `Pathname`.
- Składniki biblioteki `StringIO` i `open-uri`.

Jak się okaże po zagłębieniu w treść rozdziału, w języku Ruby obiektowością cechują się nawet operacje wejścia-wyjścia i operacje na plikach. Obiektami są strumienie wejściowe i wyjściowe (np. standardowy strumień wejściowy) lub — jeśli już mowa o plikach — dowolny uchwyt pliku. Niektóre polecenia powiązane z operacjami wejścia-wyjścia są bardziej proceduralne: jest to na przykład metoda `puts` lub metoda `system`, która umożliwia wykonanie polecenia systemowego. Jednak metoda `puts` jest proceduralna tylko wtedy, gdy przetwarza standardowy strumień wyjściowy. W przypadku umieszczenia wiersza w pliku za pomocą tej metody jawnie wysyłasz komunikat `puts` obiektowi `File`.

Obszar pamięci programu Ruby jest czymś w rodzaju wyidealizowanej przestrzeni, w której obiekty powstają i komunikują się ze sobą. Biorąc pod uwagę to, że operacje wejścia-wyjścia i wykonywanie poleceń systemowych uwzględniają wyjście poza tę wyidealizowaną przestrzeń, w języku Ruby ma miejsce wiele działań mających na celu utrzymania obiektów razem.

W rozdziale pakiety (w przeciwieństwie do rdzenia) biblioteki standardowej zostaną omówione w szerszym zakresie niż gdziekolwiek indziej w reszcie książki. Wynika to stąd, że narzędzia obsługi plików w bibliotece standardowej (zapewniane przede wszystkim przez pakiety `FileUtils`, `Pathname` i `StringIO`) oferują tak duże możliwości i wszechstronność, że uzyskały coś w rodzaju statusu quasi-rdzenia. Dziwne jest to, że jeśli zajmiesz się dowolnego rodzaju programowaniem w języku Ruby z intensywnie wykorzystywanymi plikami, osiągniesz etap, na którym pakiety będą ładowane niemal bez świadomości tego.

## 12.1. Struktura systemu operacji wejścia-wyjścia w języku Ruby

Klasa `IO` obsługuje wszystkie strumienie wejściowe i wyjściowe samodzielnie lub za pośrednictwem swoich klas potomnych, a zwłaszcza klasy `File`. Interfejs API klasy `IO` w dużej mierze składa się z elementów opakowujących wywołania biblioteki standardowej oraz pewnych rozszerzeń i modyfikacji. Im bardziej jesteś zaznajomiony z biblioteką standardową języka C, tym szybciej opanujesz takie metody jak `seek`, `getc` i `eof?`. Jeśli korzystałeś z innego języka wysokiego poziomu, który również jest wyposażony w interfejs API dość dobrze dopasowanego elementu opakowującego takie metody biblioteki, rozpoznasz ich odpowiedniki w języku Ruby. Jeśli nawet nie jesteś programistą używającym języka C ani instrukcji systemowych, szybko nabędziesz umiejętność obsługi tych metod.

### 12.1.1. Klasa IO

Obiekty klasy `IO` reprezentują możliwe do odczytu i/lub zapisu połączenia z plikami dyskowymi, klawiaturami, ekranami oraz innymi urządzeniami. Obiekt `IO` jest traktowany jak dowolny inny obiekt. Wysyłane są do niego komunikaty. Obiekt wykonuje metody i zwraca wyniki.

W momencie uruchamiania program Ruby rozpoznaje standardowe strumienie wejścia, wyjścia i błędów. Wszystkie są hermetyzowane w instancjach klasy `IO`. Możesz z nich skorzystać, aby zorientować się, jak proste są zasady działania obiektu `IO`:

```
>> STDERR.class
=> IO                                  ❶
>> STDERR.puts("Problem!")
Problem!                               ❷
=> nil
>> STDERR.write("Problem!\n")
Problem!                               ❸
=> 9
```

Stałe `STDERR`, `STDIN` i `STDOUT` (wszystkie omówiono szczegółowo w punkcie 12.1.3) są automatycznie ustawiane podczas uruchamiania programu. Stała `STDERR` jest obiektem `IO` ❶. Jeśli obiekt ten jest otwarty do zapisu (tak jest w przypadku stałej `STDERR`, ponieważ celem jest przekazanie jej wyjściowych komunikatów statusu i komunikatów o błędzie), możesz dla niego wywołać metodę `puts`. Cokolwiek, co zostanie przekazane tej metodzie, będzie zapisane w strumieniu wyjściowym tego obiektu `IO` ❷. W przypadku

stałej STDERR (przynajmniej dla domyślnej konfiguracji uruchamiania) jest to wyszukany sposób informowania, że dane wyjściowe zostaną wyświetlone na ekranie.

Oprócz metody puts obiekty IO mają metody print i write. Jeśli za pomocą drugiej metody zapisujesz dane w obiekcie IO, w danych wyjściowych nie jest automatycznie wstawiany znak nowego wiersza (pod tym względem metoda write przypomina bardziej metodę print niż metodę puts), a wartość zwracana jest liczbą zapisanych bajtów ❸.

IO to klasa języka Ruby. Jako taka ma możliwość dołączania modułów, co też robi. W szczególności obiekty IO są wyliczeniowe.

### 12.1.2. Obiekty IO jako obiekty wyliczeniowe

Jak wiesz, obiekt wyliczeniowy musi mieć metodę each, aby mógł dokonywać iteracji. Obiekty IO wykonują iterację na podstawie globalnego separatora rekordów danych wejściowych, który zgodnie z tym, co zaprezentowano w podrozdziale 10.7 w nawiązaniu do łańcuchów i ich metody each_line, jest przechowywany w zmiennej globalnej $/.

W zamieszczonych poniżej przykładach dane wyjściowe kodu Ruby wyróżniono **pogrubioną** czcionką. Zwykła czcionka wskazuje dane wprowadzone za pomocą klawiatury. Kod wykonuje iterację dla obiektu STDIN, czyli standardowego strumienia wejściowego (w następnym punkcie przyjrzymy się bliżej obiektowi STDIN oraz powiązanym z nim elementom). Na początku obiekt STDIN traktuje znak nowego wiersza jako sygnał tego, że jedna iteracja została zakończona. Oznacza to, że zostanie wyświetlony każdy wiersz po jego wprowadzeniu:

```
>> STDIN.each {|line| p line }
To jest wiersz 1
"To jest wiersz 1\n"
To jest wiersz 2
"To jest wiersz 2\n"
Wszystko jest oddzielane za pomocą zmiennej $/, czyli znaku nowego wiersza
"Wszystko jest oddzielane za pomocą zmiennej $/, czyli znaku nowego wiersza\n"
```

Jeśli jednak zmienisz wartość zmiennej $/, zmieni się też to, co w przypadku obiektu STDIN stanowi o iteracji. Zakończ pierwszą iteracją za pomocą kombinacji klawiszy *Ctrl+D* (lub w razie potrzeby przy użyciu kombinacji *Ctrl+C*!) i sprawdź następujący przykład:

```
>> $/ = "DALEJ"
=> "DALEJ"
>> STDIN.each {|line| p line}
Pierwszy wiersz
DALEJ
"Pierwszy wiersz\nDALEJ"
Następny wiersz
gdzie "wiersz" w rzeczywistości oznacza
dopiero się okaże... DALEJ
"\nNastępny wiersz\ngdzie \"wiersz\" w rzeczywistości oznacza\ndopiero się okaże... DALEJ"
```

W tym przypadku interpreter języka Ruby akceptuje dane wprowadzane za pomocą klawiatury do momentu napotkania łańcucha "DALEJ". Gdy to nastąpi, wprowadzanie rekordu uznawane jest za zakończone.

A zatem zmienna `$/` określa znaczenie metody `each` w przypadku obiektu klasy `IO`. Ponieważ obiekty `IO` są wyliczeniowe, możesz dla nich wykonywać zwykłe operacje wyliczeniowe (możesz przyjąć, że w tych przykładach zmiennej `$/` została przywrócona jej oryginalna wartość). Notacja `^D` wskazuje, że osoba wprowadzająca dane zastosowała kombinację klawiszy `Ctrl-D`:

```
>> STDIN.select {|line| line =~ /\A[A-Z]/ }
Interesują nas tylko
wiersze rozpoczynające się od
Dużych liter
^D
=> ["Interesują nas tylko\n", "Dużych liter\n"]
>> STDIN.map {|line| line.reverse }
ezsreiw eT
ąnterkes ąjareiwaz
.ćśomodaiw
^D
=> ["\nTe wiersze", "\nzawierają sekretną", "\nwiadomość."]
```

Zachowaniem o charakterze wyliczeniowym obiektów `IO` zajmiemy się ponownie w kontekście obsługi plików w podrozdziale 12.2. Na razie warto bliżej przyjrzeć się trzem podstawowym obiektom klasy `IO`: `STDIN`, `STDOUT` i `STDERR`.

### *12.1.3. Obiekty STDIN, STDOUT i STDERR*

Jeśli utworzyłeś programy i/lub skrypty powłoki, które korzystają z dowolnego rodzaju potokowania operacji wejścia-wyjścia, prawdopodobnie jesteś zaznajomiony z pojęciem *standardowych* strumieni wejściowych, wyjściowych i błędów. Zasadniczo są one domyślnie stosowane: jeśli nie zostanie to wskazane inaczej, interpreter języka Ruby zakłada, że wszystkie dane wejściowe są wprowadzane za pomocą klawiatury, a wszystkie zwykłe dane wyjściowe trafią do terminala. W tym kontekście *zakłada* oznacza, że proste, proceduralne metody operacji wejścia-wyjścia, takie jak `puts` i `gets`, działają odpowiednio w odniesieniu do obiektów `STDOUT` i `STDIN`.

Komunikaty o błędzie i obiekt `STDERR` są trochę bardziej zawiłe. Jeśli nie zostanie to wyraźnie zażądane, nic nie trafi do tego obiektu. A zatem jeśli dla danych wyjściowych ma zostać użyty obiekt `STDERR`, musisz jawnie podać jego nazwę:

```
if broken?
  STDERR.puts "Wystąpił problem!"
end
```

Oprócz trzech stałych w języku Ruby są też dostępne trzy zmienne globalne: `$stdin`, `$stdout` i `$stderr`.

#### STANDARDOWE ZMIENNE GLOBALNE OPERACJI WEJŚCIA-WYJŚCIA

Podstawową różnicą między stałą `STDIN` i zmienną `$stdin` (dotyczy to także innych podobnych par) jest to, że nie jest zakładane ponowne przypisywanie do stałej, ale możliwe jest wykonanie takiej operacji w odniesieniu do zmiennej. Zmienne zapewniają możliwość modyfikacji domyślnego działania standardowego strumienia operacji wejścia-wyjścia bez utraty oryginalnych strumieni.

Na przykład możesz wymagać, aby wszystkie dane wyjściowe trafiły do pliku łącznie ze standardowym wyjściem i błędami. Możesz to osiągnąć przy użyciu przypisań do zmiennych globalnych. Następujący kod zapisz w pliku *outputs.rb*:

```
record = File.open("/tmp/record", "w")
old_stdout = $stdout
$stdout = record
$stderr = $stdout
puts "To jest rekord"
z = 10/0
```

Pierwszym krokiem jest otwarcie pliku, w którym ma zostać dokonany zapis (jeśli w systemie nie istnieje katalog */tmp*, możesz zmienić nazwę pliku tak, aby wskazywała na inną ścieżkę, pod warunkiem że masz do niej uprawnienia zapisu). Następnie zapisz bieżącą wartość zmiennej `$stdout` w zmiennej, aby później mieć możliwość jej przywrócenia.

Pora zająć się uchwytami operacji wejścia-wyjścia. Najpierw zmienna `$stdout` jest ponownie definiowana jako uchwyt danych wyjściowych `record`. Następnie zmienna `$stderr` jest ustawiana jako równoważna zmiennej `$stdout`. Na tym etapie wszystkie poczciwe instrukcje `puts` powodują zapisanie danych wyjściowych w pliku */tmp/record*, ponieważ te zwykłe instrukcje kierują dane wyjściowe do zmiennej `$stdout`, na które ona teraz wskazuje. Dane wyjściowe zmiennej `$stderr` (np. komunikat o błędzie będący wynikiem dzielenia przez zero) również trafiają do pliku, ponieważ tej zmiennej również ponownie przypisano ten uchwyt pliku.

W efekcie po uruchomieniu programu nic nie pojawi się na ekranie. Plik */tmp/record* będzie jednak zawierać następujące dane:

```
To jest rekord
outputs.rb:6:in `/': divided by 0 (ZeroDivisionError)
    from outputs.rb:6:in `<main>'
```

Oczywiście możesz też standardowe wyjście skierować do jednego pliku, a standardowy błąd do innego pliku. Zmienne globalne umożliwiają przetwarzanie strumieni w dowolny żądany sposób.

Wkrótce zajmiemy się plikami. Ponieważ jednak ogólnie omawiamy operacje wejścia-wyjścia, a w szczególności standardowe strumienie, przyjrzyjmy się bliżej danym wprowadzanym za pomocą klawiatury.

### 12.1.4. Trochę więcej o danych wprowadzanych za pomocą klawiatury

Dane wprowadzane przy użyciu klawiatury są przeważnie obsługiwane z wykorzystaniem metod `gets` i `getc`. Jak pokazano, metoda `gets` zwraca pojedynczy wiersz danych wejściowych, a metoda `getc` zwraca jeden znak.

Różnica między tymi dwiema metodami jest taka, że w przypadku metody `getc` konieczne jest jawne podanie nazwy strumienia wejściowego:

```
line = gets
char = STDIN.getc
```

W obu przypadkach dane wejściowe są buforowane: zanim cokolwiek będzie mieć miejsce, musisz nacisnąć klawisz *Enter*. Możliwe jest sprawienie, że metoda `getc` będzie działać bez buforowania, aby swoje dane wejściowe pobrała od razu po wpisaniu znaku. Nie jest jednak dostępna metoda pozwalająca na przenoszenie tego rozwiązania między platformami z programami Ruby (w systemach uniksowych za pomocą polecenia `stty` możesz ustawić terminal w trybie „bez przetwarzania"; aby to samo osiągnąć w programie Ruby, musisz skorzystać z metody `system` opisanej w rozdziale 14.).

Jeśli z jakiegoś powodu dla zmiennej `$stdin` ustawiono coś innego niż dane wprowadzone za pomocą klawiatury, nadal możliwe jest odczytanie tych danych przez jawne użycie stałej `STDIN` jako odbiorcy metody `gets`:

```
line = STDIN.gets
```

Zakładając, że postąpiłeś zgodnie z radą podaną w poprzednim punkcie i całe przetwarzanie związane ze standardowym strumieniem operacji wejścia-wyjścia zrealizowałeś, używając zmiennych globalnych, a nie stałych, stała `STDIN` nadal będzie strumieniem danych klawiaturowych, nawet jeśli zmienna `$stdin` nim nie jest.

Zajmiemy się teraz narzędziami języka Ruby służącymi do odczytywania, zapisywania i modyfikowania plików.

## 12.2. Podstawowe operacje na plikach

Klasa wbudowana `File` zapewnia narzędzia do modyfikowania plików w kodzie Ruby. `File` to podklasa klasy `IO`, dlatego obiekty `File` i obiekty `IO` mają wspólne, określone właściwości. Jednak klasa `File` dodaje konkretne zachowania i je zmienia.

Przyjrzymy się najpierw podstawowym operacjom na plikach, w tym otwieraniu, odczytywaniu, zapisywaniu i zamykaniu plików w różnych trybach. Później zajmiemy się bardziej charakterystycznym dla języka Ruby sposobem obsługi odczytywania i zapisywania plików, czyli z wykorzystaniem bloków kodu. W dalszej kolejności obszerniej omówimy możliwości wyliczania plików, po czym podrozdział zakończymy przeglądem wybranych typowych wyjątków i komunikatów o błędzie, jakie mogą pojawić się podczas modyfikowania plików.

### 12.2.1. Podstawy operacji odczytywania z plików

Odczytywanie z pliku może być wykonywane po jednym bajcie, po ustalonej liczbie bajtów lub po jednym wierszu na raz (w tym przypadku *wiersz* jest definiowany przez ogranicznik `$/`). Możliwa jest również zmiana pozycji następnej operacji odczytu w pliku przez przesunięcie do przodu lub do tyłu o określoną liczbę bajtów bądź przemieszczenie wskaźnika wewnętrznego obiektu `File` do konkretnego przesunięcia bajtowego w pliku.

Wszystkie te operacje są wykonywane dzięki obiektom `File`. A zatem pierwszym krokiem jest utworzenie obiektu `File`. Najprościej w tym celu użyć metody `File.new`. Przekaż nazwę pliku temu konstruktorowi. Zakładając, że plik istnieje, w odpowiedzi uzyskasz uchwyt pliku otwarty do odczytu. W następującym przykładzie użyto pliku o nazwie *ticket2.rb*, który zawiera kod z listingu 3.2. Plik przechowywany jest w katalogu o nazwie *code*:

```
>> f = File.new("code/ticket2.rb")
=> #<File:code/ticket2.rb>
```

Jeśli plik nie istnieje, zostanie zgłoszony wyjątek. Na tym etapie możesz użyć instancji pliku w celu dokonania odczytu z pliku. Do dyspozycji jest kilka metod. Zdecydowanie najprostszą z nich jest metoda `read`, która wczytuje cały plik jako pojedynczy łańcuch:

```
>> f.read
=> "class Ticket\n  def initialize(venue, date)\n
                @venue = venue\n    @date = date\n  end\n\n itp.
```

Choć w wielu sytuacjach użycie metody `read` jest kuszące i odpowiednie, może okazać się nieefektywne i trochę toporne, gdy przetwarzanie i odczytywanie danych wymaga więcej szczegółowości.

W szerokim zakresie przyjrzymy się tutaj metodom odczytującym pliki Ruby, omawiając je w grupach: najpierw będą to metody bazujące na wierszach, a następnie metody oparte na bajtach.

**Zamknij uchwyty plików**

Po zakończeniu operacji odczytywania z pliku i/lub zapisywania w nim musisz go zamknąć. W tym celu obiekty `File` zapewniają metodę `close` (np. `f.close`). Poznasz sposób otwierania plików, w przypadku którego interpreter języka Ruby automatycznie zamyka plik, obejmując zasięgiem bloku kodu całą operację na plikach. Jeśli jednak operację wykonujesz w tradycyjny sposób, jak to miało miejsce w przykładach z tej części rozdziału dotyczących konstruktora `File.new`, pliki należy zamknąć jawnie (choć zostaną one również zamknięte w momencie zakończenia pracy narzędzia irb, dobrą praktyką jest zamykanie otwartych plików).

### 12.2.2. Odczytywanie plików oparte na wierszach

Najprostszym sposobem odczytania następnego wiersza pliku jest użycie metody `gets`:

```
>> f.gets
=> "class Ticket\n"
>> f.gets
=> "  def initialize(venue, date)\n"
>> f.gets
=> "    @venue = venue\n"
```

Metoda `readline` realizuje sporą część tego, co wykonuje metoda `gets`: wczytuje wiersz z pliku. Różnica tkwi w sposobie działania obu metod przy próbie dokonania odczytu poza końcem pliku: metoda `gets` zwraca obiekt `nil`, a metoda `readline` zgłasza błąd krytyczny. Różnica będzie zauważalna, jeśli użyjesz metody `read` dla obiektu `File` w celu dotarcia do końca pliku, a następnie wypróbujesz obie metody dla obiektu:

```
>> f.read
=> "  def initialize(venue, date)\n    @venue = venue\n
    @date = date\n  end\n\n
    etc.
>> f.gets
=> nil
>> f.readline
EOFError: end of file reached
```

Aby od razu pobrać cały plik jako tablicę wierszy, zastosuj metodę `readlines` (blisko powiązana z metodą `read`). Zwróć też uwagę na metodę `rewind`, która przesuwa wewnętrzny wskaźnik pozycji obiektu `File` do początku pliku:

```
>> f.rewind
=> 0
>> f.readlines
=> ["class Ticket\n", "  def initialize(venue, date)\n",
    "    @venue = venue\n", "    @date = date\n"  itp.
```

Pamiętaj o tym, że obiekty `File` są obiektami wyliczeniowymi. Oznacza to, że zamiast wczytywać cały plik do pamięci, możesz przeprowadzać iterację wierszy po jednym na raz. Metoda `each` obiektów `File` (znana jest także w postaci synonimu `each_line`) spełnia następującą rolę:

```
>> f.each {|line| puts "Następny wiersz: #{line}" }
Następny wiersz: class Ticket
Następny wiersz:   def initialize(venue, date)
Następny wiersz:     @venue = venue
itd.
```

> **UWAGA** W poprzednim przykładzie, a także w kilku kolejnych przyjęto, że użyto metody `rewind` dla obiektu `File`. Jeśli w narzędziu irb kolejno wykonujesz przykłady, wymagane będzie użycie metody `f.rewind`, aby cofnąć się do początku pliku.

Możliwość wyliczenia obiektów `File` zasługuje na osobne omówienie, czemu przyjrzymy się wkrótce. Na razie zajmijmy się prostymi operacjami odczytu opartymi na bajtach.

### *12.2.3. Odczytywanie plików w oparciu o bajty i znaki*

Jeśli cały wiersz to zbyt wiele, jak wygląda to w przypadku jednego znaku? Metoda `getc` wczytuje i zwraca jeden znak z pliku:

```
>> f.getc
=> "c"
```

Możliwe jest też „oddanie” znaku, czyli umieszczenie konkretnego znaku z powrotem w plikowym strumieniu wejściowym, aby przy następnym odczycie był on pierwszym wczytanym znakiem:

```
>> f.getc
=> "c"
>> f.ungetc("X")
=> nil
>> f.gets
=> "Xlass Ticket\n"
```

Każdy znak jest reprezentowany przez jeden lub więcej bajtów. To, jak bajty są odwzorowywane na znaki, zależy od kodowania. Niezależnie od użytego kodowania za pomocą metody `getbyte` możesz przemieszczać się w pliku w oparciu o bajty, a także znaki.

W zależności od typu kodowania liczba bajtów lub znaków w pliku może być równa albo nie, a ponadto metody `getc` i `getbyte` dla danego położenia w pliku mogą nie zwracać tych samych danych wynikowych.

Tak jak metoda `readline` różni się od metody `gets` tym, że zgłasza błąd krytyczny, jeśli zostanie użyta przy końcu pliku, tak metody `readchar` i `readbyte` różnią się w ten sam sposób odpowiednio od metod `getc` i `getbyte`. Zakładając, że dokonano już odczytu do końca obiektu `f` klasy `File`, uzyskasz następujące wyniki:

```
>> f.getc
=> nil
>> f.readchar
EOFError: end of file reached
>> f.getbyte
=> nil
>> f.readbyte
EOFError: end of file reached
```

Podczas wykonywania wszystkich tych operacji obiekt `File` (tak jak dowolny obiekt `IO`) dysponuje informacją o swoim położeniu w strumieniu wejściowym. Jak już wspomniano, z łatwością możesz cofnąć ten wewnętrzny wskaźnik do początku pliku. Możliwe jest też zmodyfikowanie wskaźnika w bardziej precyzyjny sposób.

### 12.2.4. Szukanie pozycji w pliku oraz odpytywanie o nią

Obiekt `File` dysponuje informacją o tym, gdzie w pliku przerwał odczyt. Możliwy jest zarówno jawny odczyt, jak i zmiana tego wewnętrznego wskaźnika przy użyciu atrybutu `pos` (*position* — pozycja) i/lub metody `seek` obiektu `File`.

Za pomocą tego atrybutu możesz określić, na co aktualnie w pliku wskaźnik wskazuje:

```
>> f.rewind
=> 0
>> f.pos
=> 0
>> f.gets
=> "class Ticket\n"
>> f.pos
=> 13
```

W tym przypadku pozycja to `0` po wykonaniu cofnięcia za pomocą metody `rewind` oraz `13` po odczytaniu jednego 13-bajtowego wiersza. Możliwe jest utworzenie przypisania do wartości pozycji, co powoduje przemieszczenie wskaźnika do konkretnego położenia w pliku:

```
>> f.pos = 10
=> 10
>> f.gets
=> "et\n"
```

Zwracany łańcuch jest tym, co obiekt `File` uważa za „wiersz”, począwszy od bajta 10, czyli wszystko od tej pozycji aż do następnego wystąpienia znaku nowego wiersza (lub mówiąc wprost, zmiennej `$/`).

Metoda `seek` umożliwia przemieszczanie w obrębie pliku przez przesuwanie wskaźnika pozycji do nowego położenia. Położeniem może być określone przesunięcie w pliku. Położenie może też być względne w odniesieniu do bieżącej pozycji wskaźnika lub końca pliku. Żądane dane określasz za pomocą specjalnych stałych z klasy `IO`:

```
f.seek(20, IO::SEEK_SET)
f.seek(15, IO::SEEK_CUR)
f.seek(-10, IO::SEEK_END)
```

W tym przykładzie pierwszy wiersz jest przeszukiwany do bajta 20. W drugim wierszu wskaźnik jest przesuwany o 15 bajtów względem jego bieżącej pozycji, w ostatnim wierszu wyszukiwanie odbywa się do dziesięciu bajtów przed końcem pliku. Użycie `IO::SEEK_SET` jest opcjonalne. Zwykłe wywołanie metody `f.seek(20)` realizuje to samo co kod `f.pos = 20`.

Przyjrzeliśmy się kilku sposobom odczytu z plików, zaczynając od odczytu wszystkiego naraz, omawiając następnie metodę polegającą na odczytywaniu wiersz po wierszu i wreszcie kończąc na najdokładniejszych odczytach, które bazują na znaku i pozycji. Wszystkie te techniki odczytywania z pliku uwzględniają obiekty `File`, czyli instancje klasy `File`. Sama klasa też oferuje kilka technik odczytywania.

### 12.2.5. Odczytywanie plików za pomocą metod klasy File

Trochę dalej poznasz więcej rozwiązań dostępnych w postaci metod klasy `File`. Na razie przyjrzymy się dwóm metodom, które obsługują odczytywanie plików na poziomie klasy: `File.read` i `File.readlines`.

Te dwie metody realizują to samo co ich odpowiedniki o takiej samej nazwie w postaci metod instancji. Zamiast jednak tworzyć instancję, używasz klasy `File`, nazwy metody oraz nazwy pliku:

```
full_text = File.read("moj_plik.txt")
lines_of_text = File.readlines("moj_plik.txt")
```

W pierwszym przypadku uzyskujesz łańcuch z całą zawartością pliku. W drugim przypadku otrzymujesz tablicę wierszy.

Te dwie metody klasy istnieją wyłącznie jako udogodnienie. Dbają one o automatyczne otwieranie i zamykanie uchwytu pliku. Nie musisz przeprowadzać żadnych działań porządkujących na poziomie systemu. Przeważnie będziesz wymagać osiągnięcia czegoś bardziej złożonego i/lub efektywnego niż wczytywanie naraz całej zawartości pliku do łańcucha lub tablicy. Biorąc pod uwagę nawet to, że metody instancji `read` i `readlines` są stosunkowo prostymi narzędziami, jeśli zdecydujesz się na wczytanie od razu całego pliku, równie dobrze możesz skorzystać z wersji tych metod w postaci metod klasy.

Dysponujesz teraz odpowiednim zestawem narzędzi do odczytywania plików i przetwarzania wyników. Na tym etapie zajmiemy się drugą stroną równania, czyli zapisywaniem w plikach.

**Metody niskopoziomowych operacji wejścia-wyjścia**

Oprócz różnych metod klas `IO` i `File`, którym się tutaj bliżej przyjrzymy, klasa `IO` zapewnia zestaw narzędzi w postaci metod na poziomie systemowym. Umożliwiają one wykonywanie niskopoziomowych operacji wejścia-wyjścia. Metody te obejmują: `sysseek`, `sysread` i `syswrite`. Metody odpowiadają wywołaniom systemowym, na bazie których zbudowana jest część metod wyższego poziomu.

Metody mające w nazwie na początku łańcuch `sys` wykonują podstawowe operacje na danych bez buforowania i nie powinny być łączone z metodami wyższego poziomu. Oto przykład, czego nie należy robić:

```
File.open("output.txt", "w") do |f|
  f.print("Witaj,")
  f.syswrite(" kolego!")
end
puts File.read("output.txt")
```

Jeśli uruchomisz ten niewielki program, uzyskasz następujące dane wynikowe:

```
syswrite.rb:3: warning: syswrite for buffered IO
 kolego!Witaj,
```

Oprócz ostrzeżenia uzyskujesz drugi łańcuch (zapisany za pomocą metody `syswrite`) wstawiony w pliku przed pierwszym łańcuchem. Jest to spowodowane tym, że metody `syswrite` i `print` nie działają zgodnie z tymi samymi regułami, a ponadto nie współpracują dobrze ze sobą. Najlepiej pozostać przy metodach wyższego poziomu, chyba że z konkretnego powodu musisz użyć innych metod.

### *12.2.6. Zapisywanie w plikach*

Zapisywanie w pliku wiąże się z użyciem metody `puts`, `print` lub `write` dla obiektu `File` otwartego w trybie zapisu lub dołączania. Tryb zapisu jest wskazywany przez literę `w` jako drugi argument metody `new`. W tym trybie tworzony jest plik (przy założeniu, że masz uprawnienie, które na to pozwala). Jeśli plik już istnieje, jego stara wersja jest nadpisywana. W trybie dołączania (wskazywanym przez literę `a`) wszystko, co zostanie zapisane w pliku, jest dołączane do już istniejącej zawartości. Jeśli plik jeszcze nie istnieje, otwarcie w trybie dołączania spowoduje utworzenie go.

W następującym przykładzie wykonywane są proste operacje zapisu i dołączania, które w trakcie są wstrzymywane w celu użycia bogatej w możliwości metody `File.read` sprawdzającej zawartość pliku:

```
>> f = File.new("data.out", "w")
=> #<File:data.out>
>> f.puts "Dawid A. Nowak, programista używający języka Ruby"
=> nil
>> f.close
=> nil
>> puts File.read("data.out")
Dawid A. Nowak, programista używający języka Ruby
=> nil
>> f = File.new("data.out", "a")
=> #<File:data.out>
>> f.puts "Yukihiro Matsumoto, twórca języka Ruby"
=> nil
>> f.close
=> nil
```

```
>> puts File.read("data.out")
Dawid A. Nowak, programista używający języka Ruby
Yukihiro Matsumoto, twórca języka Ruby
```

Wartość zwracana wywołania metody `puts` dla obiektu `File` jest taka sama jak wartość zwracana dowolnego wywołania tej metody, czyli jest to obiekt `nil`. To samo dotyczy metody `print`. Jeśli użyjesz niskopoziomowej metody `write`, która jest metodą instancji klasy `IO` (tym samym dostępna jest dla obiektów `File`, ponieważ klasa `File` dziedziczy z klasy `IO`), wartością zwracaną jest liczba bajtów zapisanych w pliku.

Język Ruby umożliwia ograniczenie jawnego zamykania obiektów `File`, a ponadto pozwala na dokładne hermetyzowanie kodu, zapewniając sposób wykonywania operacji na plikach wewnątrz bloku kodu. W następnym punkcie przyjrzymy się tej eleganckiej i często używanej technice.

### 12.2.7. Użycie bloków do określania zasięgu operacji na plikach

Zastosowanie metody `File.new` do tworzenia obiektu `File` ma tę wadę, że ostatecznie trzeba samemu zamknąć plik. Język Ruby zapewnia alternatywny sposób otwierania plików, w przypadku którego za realizowanie zadania porządkującego polegającego na zamykaniu pliku odpowiada interpreter tego języka, korzystając z metody `File.open` i bloku kodu.

Jeśli wywołasz metodę `File.open` z blokiem kodu, blok otrzyma obiekt `File` jak swój jedyny argument. Obiekt ten jest używany wewnątrz bloku. Po zakończeniu bloku obiekt `File` jest automatycznie zamykany.

Poniżej zamieszczono przykład, w którym po otwarciu pliku na potrzeby przetwarzania wczytywane są kolejne wiersze. Utwórz najpierw plik o nazwie *records.txt*, który w każdym wierszu zawiera jeden rekord:

```
Pablo Casals|katalońska|wiolonczela|1876-1973
Jascha Heifetz|rosyjsko-amerykańska|skrzypce|1901-1988
Emanuel Feuermann|austriacko-amerykańska|wiolonczela|1902-1942
```

Utwórz teraz kod, który wczyta ten plik wiersz po wierszu, a następnie poinformuje o znalezionych danych. W kodzie zastosowano wersję metody `File.open` bazującą na bloku:

```
File.open("records.txt") do |f|
  while record = f.gets
    name, nationality, instrument, dates = record.chomp.split('|')
    puts "#{name} (#{dates}), narodowość: #{nationality}, używany instrument: #{instrument}. "
  end
end
```

Program składa się w całości z wywołania metody `File.open` oraz jego bloku kodu (jeśli wywołasz metodę `File.open` bez bloku, będzie ona działać jak metoda `File.new`). Parametr bloku `f` odbiera obiekt `File`. Wewnątrz bloku plik jest wczytywany po jednym wierszu na raz za pomocą parametru `f`. Test instrukcji `while` kończy się powodzeniem do momentu przekazywania wierszy z pliku. Gdy program napotka koniec pliku wejściowego, metoda `gets` zwraca obiekt `nil`, a warunek instrukcji `while` nie zostaje spełniony.

Wewnątrz pętli `while` bieżący wiersz jest pobierany, aby usunąć końcowy znak nowego wiersza (jeśli występuje) i oddzielić znak w potoku. Wynikowe wartości są przechowywane w czterech zmiennych lokalnych po lewej stronie, które są następnie interpolowane do postaci ładnie wyglądającego raportu z danymi wyjściowymi:

```
Pablo Casals (1876-1973), narodowość: katalońska, używany instrument: wiolonczela.
Jascha Heifetz (1901-1988), narodowość: rosyjsko-amerykańska, używany instrument: skrzypce.
Emanuel Feuermann (1902-1942), narodowość: austriacko-amerykańska, używany instrument:
➥wiolonczela.
```

Częste jest użycie bloku kodu do określania zasięgu operacji metody `File.open`. Czasami powoduje to jednak nieporozumienia. W szczególności pamiętaj o tym, że blok zapewniony obiektowi `File` nie realizuje niczego innego. W tym przypadku nie występuje żadna niejawna pętla. Aby wczytać zawartość pliku, nadal konieczne jest zastosowanie czegoś podobnego do pętli `while` przy użyciu obiektu `File`. Miłe jest po prostu to, że może to być realizowane wewnątrz bloku kodu, a ponadto nie musisz się martwić późniejszym zamykaniem obiektu `File`.

I nie zapomnij o tym, że obiekty `File` są obiektami wyliczeniowymi.

### 12.2.8. Możliwości wyliczeniowe plików

Dzięki temu, że moduł `Enumerable` jest wśród przodków klasy `File`, instrukcję `while` w poprzednim przykładzie możesz zastąpić instrukcją `each`:

```
File.open("records.txt") do |f|
  f.each do |record|
    name, nationality, instrument, dates = record.chomp.split('|')
    puts "#{name} (#{dates}), narodowość: #{nationality}, używany instrument: #{instrument}. "
  end
end
```

Po osiągnięciu końca pliku interpreter języka Ruby w poprawny sposób przerywa iterację.

Jako obiekty wyliczeniowe obiekty `File` mogą wykonywać wiele tych samych funkcji co tablice, tablice asocjacyjne oraz inne kolekcje. Zrozumienie sposobu działania wyliczania plików wymaga trochę odmiennego modelu myślowego: podczas gdy tablica już istnieje i w ramach iteracji przetwarza swoje elementy, obiekty `File` muszą zarządzać w tle operacją wczytywania kolejnych wierszy podczas wykonywania dla nich iteracji. Jednak podobieństwo idiomów (odnoszą się one do powszechnego użycia metod modułu `Enumerable`) języka Ruby oznacza, że nie musisz zbytnio zastanawiać się nad szczegółami procesu wczytywania pliku podczas przeprowadzania dla niego iteracji.

Co najważniejsze, nie zapomnij, że pliki mogą być iterowane i adresowane jako obiekty wyliczeniowe. Kuszące jest wczytanie całego pliku do tablicy, a następnie przetwarzanie tablicy. Dlaczego jednak nie można by po prostu iterować pliku, aby uniknąć utraty przestrzeni wymaganej do przechowywania w pamięci zawartości pliku?

Możesz na przykład wczytać cały plik ze zwykłymi rekordami tekstowymi, po czym wykonać operację wprowadzania dla wynikowej tablicy w celu uzyskania średniej określonego pola:

```
# Przykładowy rekord w pliku members.txt:
# Dawid Nowak, mężczyzna, wiek: 55
count = 0
total_ages = File.readlines("members.txt").inject(0) do |total,line|
  count += 1
  fields = line.split
  age = fields[3].to_i
  total + age
end
puts "Średni wiek grupy: #{total_ages / count}."
```

Możliwe jest też jednak wykonania operacji wprowadzania bezpośrednio w obiekcie `File`:

```
count = 0
total_ages = File.open("members.txt") do |f|
  f.inject(0) do |total,line|
    count += 1
    fields = line.split
    age = fields[3].to_i
    total + age
  end
end
puts "Średni wiek grupy: #{total_ages / count}."
```

W przypadku takiego rozwiązania nie jest tworzona żadna pośrednia tablica. Obiekt `File` realizuje swoje własne zadanie.

W ten czy inny sposób niewątpliwie będziesz mieć do czynienia z sytuacjami, w których coś nie powiedzie się z operacjami na plikach. Interpreter języka Ruby nie pozostawi wątpliwości, że wystąpił problem, ale pomocne będzie wcześniejsze sprawdzenie, jakie są niektóre możliwe problemy, a także jak są zgłaszane.

### 12.2.9. Wyjątki i błędy dotyczące plikowych operacji wejścia-wyjścia

Gdy coś się nie powiedzie z operacjami na plikach, interpreter języka Ruby zgłasza wyjątek. Większość błędów, jakie wystąpią podczas pracy z plikami, można znaleźć w przestrzeni nazw `Errno`: `Errno::EACCES` (brak uprawnienia), `Errno::ENOENT` (niewłaściwy obiekt — plik lub katalog), `Errno:EISDIR` (obiekt jest katalogiem; błąd występuje przy próbie otwarcia katalogu tak, jakby był plikiem) oraz inne. Z wyjątkiem zawsze zostanie przekazany komunikat:

```
>> File.open("no_file_with_this_name")
Errno::ENOENT: No such file or directory - no_file_with_this_name
>> f = File.open("/tmp")
=> #<File:/tmp>
>> f.gets
Errno::EISDIR: Is a directory - /tmp
>> File.open("/var/root")
Errno::EACCES: Permission denied - /var/root
```

Rodzina błędów `Errno` uwzględnia nie tylko błędy powiązane z plikami, ale też inne błędy systemowe. System bazowy odwzorowuje zwykle błędy na liczby całkowite (na przykład w systemie Linux błąd informujący o tym, że obiekt nie jest katalogiem, repre-

zentowany jest przez makro języka C `ENOTDIR`, które definiowane jest jako liczba 20). Klasa `Errno` języka Ruby opakowuje takie odwzorowania błędów na liczby w pakunku klas wyjątków.

Każda klasa wyjątków `Errno` zawiera informacje o liczbie całkowitej, na jaką odwzorowywany jest odpowiadający jej błąd systemowy. Liczby te możesz uzyskać za pośrednictwem stałej `Errno` każdej klasy `Errno`. Jeśli brzmi to niejasno, dzięki następującemu przykładowi stanie się to bardziej zrozumiałe:

```
>> Errno::ENOTDIR::Errno
=> 20
```

Rzadko, jeśli w ogóle, będziesz zmuszony do martwienia się odwzorowaniem klas wyjątków `Errno` języka Ruby na liczby całkowite, na jakie używany system operacyjny odwzorowuje błędy. Należy jednak mieć świadomość tego, że dowolny wyjątek `Errno` to zasadniczo błąd systemowy przenikający do interpretera języka Ruby.

Nie są to błędy specyficzne dla języka Ruby, takie jak błędy składni lub błędy brakujących metod. Dotyczą one rzeczy, które nie powiodły się na poziomie systemowym. W takich sytuacjach interpreter języka Ruby jest jedynie posłańcem.

Powróćmy do działań możliwych, gdy wszystko się powiedzie. W dalszej kolejności zajmiemy się wybranymi sposobami, w przypadku których możesz zażądać od obiektów `IO` i `File` informacji o nich samych oraz ich stanie.

## 12.3. Odpytywanie obiektów IO i File

Obiekty `IO` i `File` mogą być odpytywane przy użyciu wielu kryteriów. Klasa `IO` zawiera kilka metod odpytujących. Klasa `File` dodaje kolejne metody.

Swój udział mają też w tym przypadku jedna klasa i jeden moduł blisko powiązane z klasą `File`. Mowa o klasie `File::Stat` i module `FileTest`. Klasa `File::Stat` zwraca obiekty, których atrybuty odpowiadają polom struktury `stat` definiowanej przez wywołanie `stat(2)` biblioteki języka C. Niektóre z tych pól są specyficzne dla systemu, a ponadto nie są istotne we wszystkich platformach. Moduł `FileTest` oferuje wiele metod służących do uzyskiwania informacji o statusie plików.

Klasa `File` również udostępnia kilka metod odpytujących. W niektórych sytuacjach na kilka sposobów możesz uzyskać te same informacje o pliku:

```
>> File.size("code/ticket2.rb")
=> 219
>> FileTest.size("code/ticket2.rb")
=> 219
>> File::Stat.new("code/ticket2.rb").size
=> 219
```

W dalszej części rozdziału przyjrzymy się dużej grupie metod odpytujących. W niektórych przypadkach są one dostępne na więcej niż jeden sposób.

### 12.3.1. Uzyskiwanie informacji z klasy File i modułu FileTest

Klasa `File` i moduł `FileTest` oferują wiele metod odpytujących, które mogą zapewnić mnóstwo informacji o pliku. Oto podstawowe kategorie zapytania: *Co to jest? Jakie ma możliwości? Jaką ma wielkość?*

Metody dostępne jako metody klasy `File` i modułu `FileTest` są prawie identyczne. Są one niemal wzajemnymi aliasami. Choć w przykładach zostanie użyty tylko moduł `FileTest`, możesz też zastosować klasę `File`.

Poniżej zamieszczono kilka pytań, jakie możesz zadać co do danego pliku, a ponadto wymieniono techniki odpytywania plików. Wszystkie te metody zwracają wartość prawda lub fałsz, z wyjątkiem metody `size`, która zwraca liczbę całkowitą. Pamiętaj o tym, że te metody sprawdzające pliki jako swoje argumenty bez problemów akceptują katalogi, łącza, gniazda oraz inne obiekty plikowe. Metody nie są ograniczone do zwykłych plików:

- *Czy plik istnieje?*

  ```
  FileTest.exist?("/usr/local/src/ruby/README")
  ```

- *Czy plik jest katalogiem? Czy jest zwykłym plikiem? Czy jest łączem symbolicznym?*

  ```
  FileTest.directory?("/home/users/dblack/info")
  FileTest.file?("/home/users/dblack/info")
  FileTest.symlink?("/home/users/dblack/info")
  ```

  Ta rodzina metod odpytujących obejmuje też metody `blockdev?`, `pipe?`, `chardev?` i `socket?`.

- *Czy plik może być odczytany? Czy może być zapisany? Czy może być wykonywany?*

  ```
  FileTest.readable?("/tmp")
  FileTest.writable?("/tmp")
  FileTest.executable?("/home/users/dblack/setup")
  ```

  Ta rodzina metod odpytujących obejmuje metody `world_readable?` i `world_writable?`, które sprawdzają dostępność większych uprawnień. Rodzina uwzględnia też warianty trzech podstawowych metod z dołączonym do ich nazwy łańcuchem `_real`. Metody sprawdzają uprawnienia bieżącego identyfikatora wykonawczego skryptu zamiast uprawnień efektywnego identyfikatora użytkownika skryptu.

- *Jaki jest rozmiar tego pliku? Czy plik jest pusty (zero bajtów)?*

  ```
  FileTest.size("/home/users/dblack/setup")
  FileTest.zero?("/tmp/tempfile")
  ```

Oprócz metod odpytujących i metod boolowskich dostępnych za pośrednictwem modułu `FileTest` (i klasy `File`) w celu uzyskiwania informacji o plikach możesz też zastosować obiekty klasy `File::Stat`.

**Uzyskiwanie informacji o pliku za pomocą metody Kernel#test**

Wśród dostępnych metod najwyższego poziomu (czyli metod prywatnych modułu `Kernel`, które możesz wywołać w dowolnym miejscu bez odbiorcy, tak jak w przypadku metody `puts`) jest metoda o nazwie `test`. W celu użycia tej metody należy przekazać jej dwa argumenty: pierwszy reprezentuje test, a drugi jest plikiem lub katalogiem. Wybór testu jest wskazywany przez znak. Wartość może być reprezentowana za pomocą notacji `?c`, w której `c` to znak lub łańcuch jednoznakowy.

Oto przykład, w którym metoda test sprawdza, czy istnieje katalog `/tmp`:

```
test ?e, "/tmp"
```

Inne typowe znaki metody `test` uwzględniają: `?d` (test jest prawdziwy, jeśli drugi argument jest katalogiem), `?f` (test jest prawdziwy, gdy drugi argument to zwykły plik) i `?z` (test jest prawdziwy, jeśli drugi argument jest plikiem o zerowej długości). W przypadku każdego testu dostępnego za pośrednictwem metody `Kernel#test` istnieje zwykle sposób pozwalający uzyskać wynik przez wywołanie metody jednej z klas omawianych w tym podrozdziale. Notacja metody `Kernel#test` jest jednak krótsza, dlatego może okazać się wygodna.

### *12.3.2. Uzyskiwanie informacji o plikach za pomocą klasy File::Stat*

Obiekty klasy `File::Stat` mają atrybuty odpowiadające strukturze `stat` w standardowej bibliotece języka C. Taki obiekt możesz utworzyć na jeden z dwóch sposobów: za pomocą metody `new` lub metody `stat` istniejącego obiektu `File`.

```
>> File::Stat.new("code/ticket2.rb")
=> #<File::Stat dev=0x1000002, ino=11531534, mode=0100644,
nlink=1, uid=501, gid=20, rdev=0x0, size=219, blksize=4096,
blocks=8, atime=2014-03-23 08:31:49 -0400,
mtime=2014-02-25 06:24:43 -0500, ctime=2014-02-25 06:24:43 -0500>
>> File.open("code/ticket2.rb") {|f| f.stat }  ←——— Takie same dane wyjściowe
```

Wyświetlone na ekranie dane wyjściowe z metody `File::Stat.new` prezentują atrybuty obiektu, w tym jego czasy utworzenia (`ctime`), modyfikacji (`mtime`) i ostatniego dostępu (`atime`).

> **WSKAZÓWKA** Dla bloku kodu `{|f| f.stat }` przekazanego w tym przykładzie metodzie `File.open` określane jest ostatnie wyrażenie, które jest w nim zawarte. Ponieważ ostatnie (tak naprawdę jedyne) wyrażenie to `f.stat`, wartością bloku jest obiekt `File::Stat`. Ogólnie rzecz biorąc, gdy metoda `File.open` używana jest z blokiem kodu, wywołanie tej metody powoduje zwrócenie ostatniej wartości z bloku. W przypadku wywołania bez bloku metoda `File.open` (podobnie jak metoda `File.new`) zwraca nowo utworzony obiekt `File`.

Spora część informacji dostępnych w klasie `File::Stat` jest złożona z metryk przypominających te z systemu Unix, takich jak numer i-węzła, tryb dostępu (uprawnienia) oraz identyfikator użytkownika i grupy. Trafność tych informacji zależy od używanego systemu operacyjnego. Nie będziemy się tutaj zajmować tym szczegółowo, ponieważ nie ma to zastosowania w przypadku wielu platform. Niezależnie jednak od tego, jakie informacje o plikach są utrzymywane przez system, w razie potrzeby skorzystania z nich będą one dostępne.

Modyfikowanie i odpytywanie plików często wiąże się z wykonywaniem podobnych operacji wobec katalogów. W klasie `Dir` język Ruby zapewnia narzędzia do wykonywania operacji na katalogach. Z tego typu operacjami spotkasz się też w niektórych narzędziach biblioteki standardowej, które zostaną omówione trochę później. Przyjrzyjmy się najpierw klasie `Dir`.

## 12.4. Modyfikowanie katalogów za pomocą klasy Dir

Podobnie jak klasa `File` klasa `Dir` zapewnia przydatne metody klasy i instancji. Aby utworzyć instancję klasy `Dir`, metodzie `new` przekaż ścieżkę do katalogu:

```
>> d = Dir.new("/usr/local/src/ruby/lib/minitest")
=> #<Dir:/usr/local/src/ruby/lib/minitest>
```

**Dopasuj ścieżkę odpowiednio do potrzeb używanego systemu**

Najczęściej używaną i najbardziej przydatną techniką związaną z klasą `Dir` jest iteracja wykonywana dla pozycji (pliki, łącza oraz inne katalogi) w katalogu.

### 12.4.1. Wczytywanie pozycji katalogu

Dostęp do pozycji możesz uzyskać na jeden z dwóch sposobów: za pomocą metody `entries` lub z zastosowaniem techniki opartej na metodzie `glob`. Główną różnicą jest to, że użycie metody `glob` dla katalogu nie powoduje zwrócenia ukrytych pozycji, co w przypadku wielu systemów operacyjnych (w tym wszystkich systemów uniksowych) oznacza pozycje o nazwach zaczynających się kropką. Technika oparta na metodzie `glob` umożliwia też dopasowywanie symboli wieloznacznych oraz dopasowywanie rekurencyjne w podkatalogach.

#### METODA ENTRIES

Zarówno sama klasa `Dir`, jak i jej instancje mogą zapewniać pozycje katalogu. Dla wcześniej utworzonej instancji klasy `Dir` możesz wykonać następującą operację:

```
>> d.entries
=> [".", "..", ".document", "autorun.rb", "benchmark.rb", "hell.rb",
"mock.rb", "parallel_each.rb", "pride.rb", "README.txt", "spec.rb",
"unit.rb"]
```

Możliwe jest też skorzystanie z rozwiązania opartego na metodzie klasy:

```
>> Dir.entries("/usr/local/src/ruby/lib/minitest")
=> [".", "..", ".document", "autorun.rb", "benchmark.rb", "hell.rb",
"mock.rb", "parallel_each.rb", "pride.rb", "README.txt", "spec.rb",
"unit.rb"]
```

Zwróć uwagę na obecność pozycji z jedną i dwiema kropkami (reprezentują odpowiednio bieżący katalog i katalog nadrzędny), tak jak w przypadku ukrytej pozycji `.document`. Aby dokonać iteracji pozycji, przetwarzając wyłącznie pliki, musisz zadbać o odfiltrowanie nazw rozpoczynających się kropkami.

Załóżmy, że mają zostać zsumowane wielkości wszystkich nieukrytych, zwykłych plików w katalogu. Oto pierwsza iteracja (później zostanie opracowana jej krótsza wersja):

```
d = Dir.new("/usr/local/src/ruby/lib/minitest")
entries = d.entries
entries.delete_if {|entry| entry =~ /^\./ }
entries.map! {|entry| File.join(d.path, entry) }
entries.delete_if {|entry| !File.file?(entry) }
print "Łączna liczba bajtów: "
puts entries.inject(0) {|total, entry| total + File.size(entry) }
```

Najpierw tworzymy obiekt `Dir` dla katalogu docelowego i pobieramy jego pozycje. W dalszej kolejności dla tablicy pozycji wykonywana jest sekwencja modyfikacji. Używając metody tablicy `delete_if`, usuwamy wszystko, co zaczyna się od kropki. Później wykonywane jest wewnętrzne odwzorowanie tablicy pozycji w taki sposób, że każda pozycja zawiera pełną ścieżkę do pliku. Jest to osiągane za pomocą dwóch przydatnych metod: metody instancji `Dir#path` zwracającej ścieżkę oryginalnego katalogu bazowego dla konkretnej instancji klasy `Dir` (`/usr/local/src/ruby/lib/minitest`) oraz metody `File.join`, która łączy ścieżkę do nazwy pliku z właściwym separatorem (zwykle jest to znak `/`, ale w pewnym stopniu zależy to od systemu).

Gdy już pozycje zostały tak zmodyfikowane, aby reprezentowały pełne nazwy ścieżek, wykonywana jest kolejna operacja metody `delete_if` w celu usunięcia wszystkich pozycji, które nie są zwykłymi plikami. Do identyfikacji takich pozycji stosowana jest metoda testująca `File.file?`. Tablica pozycji zawiera aktualnie pełne nazwy ścieżek wszystkich zwykłych plików w oryginalnym katalogu. Ostatnim krokiem jest zsumowanie wielkości tych plików. Do wykonania tego zadania idealnie nadaje się metoda `inject`.

Wśród innych sposobów skrócenia tego kodu jest użycie dla katalogu obsługi symboli wieloznacznych zamiast metody `entries`.

### OBSŁUGA SYMBOLI WIELOZNACZNYCH W PRZYPADKU KATALOGÓW

Obsługa symboli wieloznacznych w języku Ruby swoją semantykę wywodzi w dużej mierze z obsługi symboli wieloznacznych w powłoce, w przypadku której składnia umożliwia wykonywanie następujących działań:

```
$ ls *.rb
$ rm *.?xt
$ for f in [A-Z]*   # itp.
```

Oczywiście poszczególne powłoki różnią się szczegółami, ale rzecz w tym, że cała ta rodzina technik rozwijających nazwy jest dostępna w postaci składni obsługi symboli wieloznacznych w języku Ruby. Gwiazdka reprezentuje dopasowanie symbolu wieloznacznego dla dowolnej liczby znaków. Pytajnik reprezentuje jeden znak symbolu wieloznacznego. Klasy znaków w stylu wyrażeń regularnych są dostępne na potrzeby dopasowywania.

W celu zapewnienia obsługi symboli wieloznacznych dla katalogu możesz użyć metody `Dir.glob` lub `Dir.[]` (nawiasy kwadratowe). Wersja metody z nawiasami kwadratowymi pozwala zastosować składnię w stylu indeksu, tak jak w przypadku metody z takimi nawiasami zastosowanej dla tablicy lub tablicy asocjacyjnej. W odpowiedzi uzyskujesz tablicę zawierającą zestaw wynikowy:

```
>> Dir["/usr/local/src/ruby/include/ruby/r*.h"]
=> ["/usr/local/src/ruby/include/ruby/re.h", "/usr/local/src/ruby/include/
ruby/regex.h", "/usr/local/src/ruby/include/ruby/ruby.h"]
```

Metoda `glob` w dużej mierze odpowiada metodzie `[]`, ale jest trochę bardziej wszechstronna: metodzie `glob` możesz przekazać nie tylko wzorzec obsługi symboli wieloznacznych, ale też co najmniej jeden argument flagi sterujący zachowaniem metody. Aby na przykład skorzystać z obsługi symboli wieloznacznych bez rozróżniania wielkości znaków, możesz przekazać flagę `File::FNM_CASEFOLD`:

```
Dir.glob("info*")       # []
Dir.glob("info", File::FNM_CASEFOLD   # ["Informacje", "INFORMACJE"]
```

Inną przydatną flagą jest flaga `FNM_DOTMATCH`, która uwzględnia w wynikach ukryte pliki z kropką w nazwie.

W celu zastosowania dwóch flag należy połączyć je za pomocą operatora bitowego `OR`, który ma postać pojedynczego znaku potoku. W tym przykładzie wraz z dodawaniem mniej restrykcyjnych flag znajdowana jest coraz większa liczba plików:

```
>> Dir.glob("*info*")
=> []
>> Dir.glob("*info*", File::FNM_DOTMATCH)
=> [".information"]
>> Dir.glob("*info*", File::FNM_DOTMATCH | File::FNM_CASEFOLD)
=> [".information", ".INFO", "Info"]
```

Flagi są faktycznie liczbami. Na przykład wartość `File::FNM_DOTMATCH` wynosi `4`. Konkretne liczby nie mają znaczenia (ostatecznie są one uzyskiwane z flag w funkcji biblioteki systemowej `fnmatch`). Istotne jest to, że są one reprezentantami dwóch korzyści wynikających z użycia operatora `OR` do łączenia flag.

> **UWAGA** Jak możesz zauważyć na podstawie dwóch pierwszych wierszy poprzedniego przykładu, operacja wykonywana przez metodę `glob` dla katalogu może niczego nie znaleźć, a jednak nie będzie zgłaszać problemu. W tym przypadku uzyskana zostanie pusta tablica. Gdy stosowana jest obsługa symboli wieloznacznych, fakt, że nic nie zostało znalezione, nie jest postrzegany jako niepowodzenie.

Obsługa symboli wieloznacznych z wykorzystaniem nawiasów kwadratowych jest tym samym co obsługa bez zapewniania żadnych flag. Inaczej mówiąc, wykonanie następującej operacji:

```
Dir["*info*"]
```

Odpowiada następującej operacji:

```
Dir.glob("*info*", 0)
```

Ponieważ domyślnie żadna z flag nie obowiązuje, ta operacja odpowiada następującej operacji:

```
Dir.glob("*info*")
```

Metoda obiektu `Dir` w postaci nawiasów kwadratowych zapewnia swego rodzaju skrót na potrzeby najczęstszego przypadku. Jeśli wymagasz większej szczegółowości, użyj metody `Dir.glob`.

Domyślnie obsługa symboli wieloznacznych nie obejmuje nazw plików zaczynających się kropkami. Ponadto — jak możesz zauważyć — operacja obsługi symboli wieloznacznych zwraca pełne nazwy ścieżek, a nie tylko nazwy plików. Fakty te powodują, że w efekcie możliwe jest zmniejszenie ilości kodu w przykładzie sumowania wielkości plików:

```
dir = "/usr/local/src/ruby/lib/minitest"
entries = Dir["#{dir}/*"].select {|entry| File.file?(entry) }
print "Łączna liczba bajtów: "
puts entries.inject(0) {|total, entry| total + File.size(entry) }
```

Wyniki operacji obsługi symboli wieloznacznych, która wyklucza pliki z kropką na początku nazwy oraz uwzględnia pełne ścieżki, często są bardziej zgodne ze sposobami, z jakimi wielu z nas ma do czynienia podczas codziennej pracy z plikami i katalogami, niż wyniki metody `Dir.entries`.

Zarządzanie katalogami nie ogranicza się jedynie do sprawdzenia, co zawierają. W dalszej kolejności przyjrzymy się technikom umożliwiającym zagłębienie się w większym stopniu w ten proces.

### 12.4.2. Modyfikowanie i odpytywanie katalogów

Klasa `Dir` zawiera kilka metod odpytujących służących do uzyskiwania informacji o katalogu lub bieżącym katalogu, a także metody przeznaczone do tworzenia i usuwania katalogów. Podobnie jak wiele innych metod najlepiej zilustrować je za pomocą przykładu.

W tym przypadku zostanie utworzony nowy katalog (`mkdir`), a następnie po przejściu do niego (`chdir`) zostanie dodany i sprawdzony plik oraz usunięty katalog (`rmdir`):

```
newdir = "/tmp/newdir"                          ❶
newfile = "newfile"
Dir.mkdir(newdir)
Dir.chdir(newdir) do                            ❷
  File.open(newfile, "w") do |f|
    f.puts "Przykładowy plik w nowym katalogu"  ❸
  end
  puts "Bieżący katalog: #{Dir.pwd}"            ❹
  puts "Listing katalogu: "
  p Dir.entries(".")
  File.unlink(newfile)                          ❺
end
Dir.rmdir(newdir)                               ❻
print "Czy katalog #{newdir} nadal istnieje? "
if File.exist?(newdir)                          ❼
  puts "Tak"
else
  puts "Nie"
end
```

Po zainicjowaniu pary wygodnych zmiennych ❶ tworzony jest nowy katalog za pomocą metody `mkdir`. Przy użyciu metody `Dir.chdir` przechodzimy do tego katalogu. Użycie bloku wraz z metodą `chdir` oznacza, że po opuszczeniu bloku następuje powrót do poprzedniego katalogu ❷. Zastosowanie metody `chdir` bez bloku powoduje zmianę bieżącego katalogu do momentu, gdy zostanie on jawnie ponownie zmieniony.

W ramach kroku zapełniania katalogu tworzymy pojedynczy plik zawierający jeden wiersz ❸. Używając metody `Dir.pwd`, sprawdzamy następnie nazwę bieżącego katalogu i wyświetlamy listing pozycji katalogu ❹. W dalszej kolejności za pomocą metody `unlink` usuwamy ostatnio utworzony plik ❺. W tym momencie ma miejsce zakończenie bloku metody `chdir`.

Po powrocie do dowolnego katalogu, jaki został ustalony jako początkowy, przy użyciu metody `Dir.rmdir` (może też zostać wywołana jako metoda `unlink` lub `delete`) usuwamy przykładowy katalog ❻. Na końcu sprawdzamy istnienie obiektu `newdir`, oczekując odpowiedzi `Nie` (ponieważ metoda `rmdir` spowodowałaby zgłoszenie błędu krytycznego, jeśli nie znalazłaby katalogu, a ponadto nie usunęłaby go pomyślnie) ❼.

Jak obiecano we wprowadzeniu do rozdziału, przyjrzymy się teraz niektórym narzędziom biblioteki standardowej służącym do modyfikowania i obsługi plików.

## 12.5. Narzędzia plikowe biblioteki standardowej

Obsługa plików to obszar, w którym biblioteka standardowa oferuje szczególnie duże możliwości. W związku z tym zajmiemy się tutaj tą tematyką obszerniej niż gdziekolwiek indziej w książce. Nie oznacza to, że reszta biblioteki standardowej nie jest warta poznania. Jednak rozszerzenia udostępnione na potrzeby modyfikowania plików odgrywają na tyle kluczową rolę odnośnie do sposobu, w jaki większość osób modyfikuje pliki w kodzie Ruby, że bez nich nie jest możliwe zdobycie pewnych podstaw dotyczących procesu modyfikowania.

Najpierw przyjrzymy się wszechstronnemu pakietowi `FileUtils`, a następnie bardziej specjalizowanej, lecz przydatnej klasie `Pathname`. Następnie omówimy klasę `StringIO`, której obiekty to w zasadzie łańcuchy z interfejsem operacji wejścia-wyjścia. Łańcuchy te mogą być przetwarzane za pomocą metody `rewind`, przeszukiwane przy użyciu metody `seek`, pobierane przez metodę `getc` itd. Na końcu przybliżymy pakiet `open-uri`, który umożliwia „otwieranie" identyfikatorów URI i wczytywanie ich do łańcuchów z taką łatwością, jakby były plikami lokalnymi.

### 12.5.1. Moduł FileUtils

Moduł `FileUtils` zapewnia kilka praktycznych i wygodnych metod, które ułatwiają modyfikowanie plików z poziomu kodu Ruby w zwięzły sposób odpowiadający temu, jak są używane znajome polecenia systemowe. Nazwy metod będą szczególnie znane użytkownikom systemu operacyjnego Unix i systemów do niego podobnych. Metody te z łatwością mogą zostać opanowane przez osoby, które jeszcze ich nie znają.

Wiele metod modułu `FileUtils` zostało nazwanych z uwzględnieniem nazw poleceń systemowych z konkretnymi opcjami wiersza poleceń. Na przykład metoda `FileUtils.rm_rf` emuluje polecenie `rm -rf` (wymusza bezwarunkowe, rekurencyjne usuwanie pliku lub katalogu). Za pomocą metody `FileUtils.ln_s(`*`nazwa_pliku, nazwa_łącza`*`)`

możesz utworzyć łącze symboliczne między *nazwą pliku* `nazwa_pliku` i *nazwą łącza* `nazwa_łącza` w sposób bardzo przypominający działanie polecenia `ln -s`.

Jak widać, część metod modułu `FileUtils` jest specyficzna dla systemu operacyjnego. Jeśli używany system nie obsługuje łączy symbolicznych, metoda `ln_s` nie zadziała. Jednak większość metod modułu może być przenoszona. Przyjrzymy się przykładom niektórych spośród najbardziej przydatnych metod.

### KOPIOWANIE, PRZENOSZENIE I USUWANIE PLIKÓW

Moduł `FileUtils` zapewnia kilka zwięzłych metod wysokiego poziomu, które realizują te operacje. Metoda `cp` emuluje tradycyjną metodę systemu Unix o tej samej nazwie. Za pomocą tej metody możesz skopiować jeden plik do drugiego lub kilka plików do katalogu:

```
>> require 'fileutils'
=> true
>> FileUtils.cp("baker.rb", "baker.rb.bak")
=> nil
>> FileUtils.mkdir("backup")          ←❶
=> ["backup"]
>> FileUtils.cp(["ensure.rb", "super.rb"], "backup")
=> ["ensure.rb", "super.rb"]
>> Dir["backup/*"]                    ←❷
=> ["backup/ensure.rb", "backup/super.rb"]
```

W tym przykładzie pokazano też metodę `mkdir` ❶, a także użycie metody `Dir#[]` ❷ w celu zweryfikowania obecności skopiowanych plików w nowym katalogu zapasowych danych.

Tak jak możesz skopiować pliki, tak masz również możliwość przenoszenia ich pojedynczo lub w grupach:

```
>> FileUtils.mv("baker.rb.bak", "backup")
=> 0
>> Dir["backup/*"]
=> ["backup/baker.rb.bak", "backup/ensure.rb", "backup/super.rb"]
```

Z łatwością możesz także usuwać pliki i katalogi:

```
>> File.exist?("backup/super.rb")
=> true
>> FileUtils.rm("./backup/super.rb")
=> ["./backup/super.rb"]
>> File.exist?("backup/super.rb")
=> false
```

Metoda `rm_rf` w sposób rekurencyjny i bezwarunkowy usuwa katalog:

```
>> FileUtils.rm_rf("backup")
=> ["backup"]
>> File.exist?("backup")
=> false
```

Moduł `FileUtils` zapewnia wartościowy zestaw narzędzi przeznaczonych do konserwacji plików w prosty i szybki sposób. To jednak nie wszystko: zestaw ten umożliwia sprawdzanie poleceń bez wykonywania ich.

**MODUŁY DRYRUN I NOWRITE**

Aby dowiedzieć się, co się stanie w przypadku uruchomienia konkretnego polecenia modułu `FileUtils`, możesz wysłać polecenie do modułu `FileUtils::DryRun`. Dane wyjściowe wywołanej metody są reprezentacją polecenia systemu uniksowego, która odpowiada temu, co zostałoby uzyskane po wywołaniu tej samej metody w module `FileUtils`:

```
>> FileUtils::DryRun.rm_rf("backup")
rm -rf backup
=> nil
>> FileUtils::DryRun.ln_s("backup", "backup_link")
ln -s backup backup_link
=> nil
```

Aby upewnić się, że omyłkowo nie usunięto, nie nadpisano ani nie przeniesiono plików, polecenia możesz przekazać do modułu `FileUtils::NoWrite`, który ma ten sam interfejs co moduł `FileUtils`, ale nie wykonuje żadnych operacji dokonujących zapisu na dysku:

```
>> FileUtils::NoWrite.rm("backup/super.rb")
=> nil
>> File.exist?("backup/super.rb")
=> true
```

Prawie na pewno moduł `FileUtils` okaże się przydatny w wielu sytuacjach. Jeśli nawet nie jesteś zaznajomiony z poleceniami systemu Unix, na których bazuje wiele nazw metod modułu `FileUtils`, opanujesz je szybko. Ponadto wyeliminują one w celu zrealizowania zadań konieczność zagłębiania się bardziej w niższego poziomu biblioteki operacji wejścia-wyjścia i biblioteki plików.

W dalszej kolejności przyjrzymy się następnemu składnikowi biblioteki standardowej powiązanemu z plikami, czyli rozszerzeniu nazwy ścieżki.

### *12.5.2. Klasa Pathname*

Klasa `Pathname` umożliwia tworzenie obiektów `Pathname` oraz ich modyfikowanie i odpytywanie. Dzięki temu możesz na przykład określić nazwę bazową i rozszerzenie nazwy ścieżki lub dokonywać iteracji ścieżki podczas przechodzenia w górę struktury katalogów.

Obiekty `Pathname` zawierają też dużą liczbę metod reprezentujących klasy `File`, `Dir`, `IO` oraz inne klasy. Nie będziemy się tutaj zajmować tymi metodami. Pozostaniemy przy metodach unikatowych dla klasy `Pathname`.

Zaczniemy od obiektu `Pathname`:

```
>> require 'pathname'
=> true
>> path = Pathname.new("/Users/dblack/hacking/test1.rb")
=> #<Pathname:/Users/dblack/hacking/test1.rb>
```

W przypadku wywołania metod obiektu `Pathname` często uzyskuje się kolejny obiekt `Pathname`. Jednak nowy obiekt zawsze ma swoją reprezentację łańcuchową widoczną w jego własnym łańcuchu `inspect`. Aby wyświetlić sam łańcuch, możesz użyć metody `to_s` lub wywołać metodę `puts` dla nazwy ścieżki.

Oto dwa sposoby sprawdzania nazwy bazowej ścieżki:

```
>> path.basename
=> #<Pathname:test1.rb>
>> puts path.basename
test1.rb
```

Możliwe jest też sprawdzenie katalogu zawierającego plik lub katalog reprezentowany przez nazwę ścieżki:

```
>> path.dirname
=> #<Pathname:/Users/dblack/hacking>
```

Jeśli ostatni segment ścieżki zawiera rozszerzenie, możesz je uzyskać z obiektu `Pathname`:

```
>> path.extname
=> ".rb"
```

Używając metody `ascend` i bloku kodu, obiekt `Pathname` może też przechodzić w górę swojej struktury plików i katalogów, obcinając się od prawej strony dla każdej iteracji:

```
>> path.ascend do |dir|
?>   puts "Następny wyższy poziom: #{dir}"
>> end
```

Oto dane wyjściowe:

```
Następny wyższy poziom: /Users/dblack/hacking/test1.rb
Następny wyższy poziom: /Users/dblack/hacking
Następny wyższy poziom: /Users/dblack
Następny wyższy poziom: /Users
Następny wyższy poziom: /
```

Kluczową cechą obiektów `Pathname` związaną z ich zachowaniem jest zwracanie przez nie innych obiektów `Pathname`. Oznacza to, że możliwe jest rozszerzenie logiki operacji dotyczących nazwy ścieżki bez konieczności dokonywania ciągłych konwersji z czystych łańcuchów i do ich postaci. Aby to zobrazować, poniżej ponownie zamieszczono poprzedni przykład, ale tak zmieniony, żeby wykorzystać fakt, że to, co w każdej iteracji przechodzi przez parametr bloku `dir`, nie jest łańcuchem (nawet pomimo tego, że po wyświetleniu tak wygląda), lecz obiektem `Pathname`:

```
>> path = Pathname.new("/Users/dblack/hacking/test1.rb")
=> #<Pathname:/Users/dblack/hacking/test1.rb>
>> path.ascend do |dir|
?>   puts "Zmniejszono do: #{dir.basename}"
>> end
```

Oto dane wyjściowe:

```
Zmniejszono do: test1.rb
Zmniejszono do: hacking
Zmniejszono do: dblack
Zmniejszono do: Users
Zmniejszono do: /
```

To, że parametr `dir` zawsze jest obiektem `Pathname`, oznacza, że możliwe jest wywołanie dla niego metody `basename`. Prawdą jest, że zawsze możesz wywołać metodę

`File.basename(string)` dla dowolnego łańcucha. Klasa `Pathname` określa jednak konkretne informacje, jakie ścieżka przypuszczalnie może o sobie uwzględnić i udostępnić programiście za pośrednictwem prostych wywołań metod.

W dalszej kolejności przyjrzymy się innej klasie biblioteki standardowej, która oferuje duże możliwości. Mowa o klasie `StringIO`.

### *12.5.3. Klasa StringIO*

Klasa `StringIO` daje możliwość traktowania łańcuchów jak obiektów `IO`. Dla takich łańcuchów mogą być stosowane metody `seek`, `rewind` itp.

Korzyścią zapewnianą przez klasę `StringIO` jest możliwość tworzenia metod, które używają interfejsu API obiektu `IO`. Metody te będą w stanie obsługiwać łańcuchy. Może to być przydatne podczas testowania, a także w kilku sytuacjach związanych ze środowiskiem wykonawczym.

Załóżmy na przykład, że istnieje moduł usuwający z pliku komentarz: po wczytaniu danych z jednego pliku moduł zapisuje w drugim pliku wszystko, co nie jest komentarzem. Taki moduł może wyglądać następująco:

```
module DeCommenter
  def self.decomment(infile, outfile, comment_re = /\A\s*#/)
    infile.each do |inline|
      outfile.print inline unless inline =~ comment_re
    end
  end
end
```

Metoda `DeCommenter.decomment` oczekuje dwóch otwartych uchwytów pliku: jednego, który umożliwia odczyt, oraz jednego pozwalającego na zapis. Metoda pobiera też wyrażenie regularne z wartością domyślną. Wyrażenie regularne określa, czy każdy wiersz w danych wejściowych to komentarz. Każdy wiersz *niezgodny* z wyrażeniem regularnym jest umieszczany w pliku wyjściowym.

Oto typowy przypadek użycia modułu `DeCommenter`:

```
File.open("myprogram.rb") do |inf|
  File.open("myprogram.rb.out", "w") do |outf|
    DeCommenter.decomment(inf, outf)
  end
end
```

W tym przykładzie komentarze są usuwane z hipotetycznego pliku programu *myprogram.rb*.

Co będzie, jeśli postanowisz utworzyć test dla modułu `DeCommenter`? Testowanie transformacji plików może być trudne, ponieważ musisz utrzymać plik wejściowy jako część testu, a ponadto upewnić się, że masz możliwość zapisu w pliku wyjściowym, który trzeba następnie ponownie wczytać. Klasa `StringIO` ułatwia to, umożliwiając pozostawienie całego kodu w jednym miejscu bez konieczności wczytywania lub zapisywania faktycznych plików.

**Testowanie za pomocą rzeczywistych plików**

Jeśli zamierzasz uruchomić testy dotyczące plikowych danych wejściowych i wyjściowych przy użyciu rzeczywistych plików, pomocna może okazać się klasa `tempfile` języka Ruby. Jest to funkcja biblioteki standardowej, dlatego musisz zastosować instrukcję `require 'plik_tymczasowy'`. W dalszej kolejności tworzysz pliki tymczasowe za pomocą konstruktora, przekazując nazwę, którą interpreter języka Ruby przekształca do postaci unikatowej nazwy pliku. Oto przykład:

```
tf = Tempfile.new("my_temp_file")
```

Za pomocą metody `tf` obiektu `File` możesz następnie dokonywać zapisu i odczytu pliku.

Aby w przypadku klasy `StringIO` użyć kodu usuwającego komentarze, moduł zapisz w pliku *decommenter.rb*. Utwórz później w tym samym katalogu drugi plik *decomment-demo.rb* z następującą zawartością:

```
require 'stringio'
require_relative 'decommenter'                    ❶
string = <<EOM
# To jest komentarz.
To nie jest komentarz.
# A to jest komentarz.                            ❷
   # To też jest.
To także nie jest komentarz.
EOM
infile = StringIO.new(string)                     ❸
outfile = StringIO.new("")
DeCommenter.decomment(infile,outfile)             ❹
puts "Test powiódł się" if outfile.string == <<EOM   ❺
To nie jest komentarz.
To także nie jest komentarz.
EOM
```

Po załadowaniu biblioteki `stringio` i kodu usuwającego komentarz ❶ program ustawia dla zmiennej `string` pięciowierszowy łańcuch (utworzony za pomocą dokumentu miejscowego), który zawiera kombinację wierszy komentarza i wierszy innych niż komentarz ❷. Następnie tworzone są dwa obiekty `StringIO`: jeden używający zawartości zmiennej `string` jako swojej zawartości oraz jeden, który jest pusty ❸. Pusty obiekt reprezentuje plik wyjściowy.

Dalej następuje wywołanie metody `DeCommenter.decomment` ❹. Moduł traktuje swoje dwa argumenty jako obiekty `File` lub `IO`, dokonując odczytu z jednego i zapisując w drugim. Obiekty `StringIO` bez problemu zachowują się podobnie do obiektów `IO`. Ponadto między nimi odbywa się filtrowanie. Po zakończeniu filtrowania możesz przeprowadzić jawne sprawdzenie, aby upewnić się, że w „pliku" wyjściowym zostało zapisane to, czego oczekiwano ❺. Oryginalna i zmieniona zawartość jest fizycznie obecna w tym samym pliku, co ułatwia stwierdzenie, co wykonuje test, a także zmodyfikowanie go.

Kolejną przydatną funkcją biblioteki standardowej jest moduł `open-uri`.

### 12.5.4. Moduł open-uri

Pakiet `open-uri` biblioteki standardowej umożliwia pobieranie informacji z sieci za pomocą protokołów HTTP i HTTPS tak prosto, jakby były wczytywane pliki lokalne. Konieczne jest jedynie zażądanie biblioteki (`require 'open-uri'`) i użycie metody `Kernel#open` z identyfikatorem URI jako argumentem. W odpowiedzi uzyskasz obiekt `StringIO` zawierający wyniki żądania:

```
require 'open-uri'
rubypage = open("http://rubycentral.org")
puts rubypage.gets
```

Deklaracja `doctype` jest uzyskiwana z głównej strony internetowej Ruby Central. Choć nie jest to najbardziej błyskotliwy sposób wczytywania, demonstruje łatwość, z jaką moduł `open-uri` pozwala importować materiały sieciowe.

## 12.6. Podsumowanie

W rozdziale zaprezentowano następujące zagadnienia:

- Operacje wejścia-wyjścia i operacje na plikach w języku Ruby.
- Obiekty plików jako obiekty wyliczeniowe.
- Obiekty `STDIN`, `STDOUT` i `STDERR`.
- Moduł `FileUtils`.
- Moduł `Pathname`.
- Klasa `StringIO`.
- Moduł `open-uri`.

Operacje wejścia-wyjścia są oparte na klasie `IO`, której podklasą jest klasa `File`. Spora część działań realizowanych przez klasy `IO` i `File` składa się z opakowanych wywołań biblioteki. Są to zasadniczo biblioteki interfejsów API znajdujące się ponad systemowymi narzędziami operacji wejścia-wyjścia.

Dla uchwytów plików w kodzie Ruby iteracja może być wykonywana tak, jakby były one tablicami, z wykorzystaniem metod `each`, `map` i `reject` oraz innych metod z modułu `Enumerable`. Interpreter języka Ruby zajmie się szczegółami związanymi z obsługą plików. W razie potrzeby obiekty `IO` i `File` mogą być też adresowane za pomocą poleceń niskopoziomowych.

Część narzędzi biblioteki standardowej służących do modyfikowania plików jest niezbędna. Przyjrzeliśmy się kilku z nich: modułowi `FileUtils` (zapewnia rozbudowany zestaw narzędziowy do wykonywania operacji dyskowych i plikowych), klasie `StringIO` (umożliwia adresowanie łańcucha tak, jakby był strumieniem wejścia-wyjścia), rozszerzeniu `Pathname` (pozwala realizować rozbudowane i proste operacje dla łańcuchów reprezentujących ścieżki systemu plików) oraz modułowi `open-uri` (ułatwia „otwieranie" dokumentów znajdujących się w sieci).

Przyjrzeliśmy się również danym wprowadzanym za pomocą klawiatury i wyświetlanym na ekranie, które są obsługiwane za pośrednictwem obiektów `IO`, a w szczególności uchwytom operacji wejścia-wyjścia dla standardowego wejścia, wyjścia i błę-

dów. W języku Ruby możliwe jest ponowne przypisywanie, które w razie potrzeby pozwala na przekierowanie danych wejściowych i wyjściowych.

Niniejszym rozdziałem zakończyliśmy część drugą książki, a tym samym przegląd wbudowanych funkcji i klas języka Ruby. W części trzeciej przejdziemy do obszernej tematyki związanej z dynamiką języka Ruby. Najpierw omówimy jedno z najprostszych, ale jednocześnie najbardziej fundamentalnych założeń języka: zgodnie z tym założeniem obiekty, a nawet obiekty tej samej klasy, mogą działać i reagować indywidualnie.

# Część III

# Dynamika języka Ruby

*Język Ruby jest dynamiczny jak ludzka natura.*
— Matz, konferencja RubyConf 2001

Termin *dynamika języka Ruby* jest niemal zbędny: wszystko związane z tym językiem jest dynamiczne. Dla zmiennych nie ma znaczenia to, z jaką klasą obiektu są wiązane. Oznacza to, że nie musisz (tak naprawdę nie możesz) wcześniej deklarować ich typu. Obiekty uzyskują swoje możliwości od tworzących je klas, ale mogą też oddzielać się od nich, jeśli doda się poszczególne metody do tych obiektów. Klasy i moduły mogą być ponownie otwierane i modyfikowane po utworzeniu ich początkowych definicji. Nic nie musi pozostać takie samo w trakcie cyklu „życia" działającego programu.

Przykłady te są zaledwie początkiem. W tej ostatniej części książki przyjrzymy się dokładniej niż dotychczas temu, jak język Ruby umożliwia zmianę w programie warunków jego wykonywania.

Najpierw w rozdziale 13. zajmiemy się *indywidualizacją obiektów*, omawiając szczegóły tego, jak w języku Ruby możliwe jest zapewnienie poszczególnym obiektom samodzielnego funkcjonowania, a także opracowanie własnych parametrów i zachowań niezależnie od parametrów klasy, która obiekty „powołała do życia". Tym sposobem powrócimy do jednego z zagadnień najwcześniej przedstawionych w książce, czyli do dodawania metod do poszczególnych obiektów. Jednak w tym rozdziale, dysponując wiedzą zdobytą po przeczytaniu dotychczasowej zawartości książki, w znacznie większym stopniu skoncentrujemy się na bazowych mechanizmach indywidualizacji obiektów.

W rozdziale 14. zajmiemy się *obiektami wywoływanymi*, które możesz wykonywać. Oczywiście zaprezentowano już metody, ale nie było jeszcze mowy o *obiektach* metod, które zostaną omówione w tym rozdziale wraz z funkcjami anonimowymi w postaci obiektów `Proc`. Same łańcuchy nie mogą być wywoływane, ale możliwe jest określenie wartości dla łańcucha w czasie wykonywania jako części kodu Ruby. W rozdziale 14. uwzględniono tę technikę (budzącą czasami wątpliwości). W rozdziale zaprezentowano również wątki języka Ruby, które umożliwiają równoległe uruchamianie segmentów kodu.

W rozdziale 15. omówiono narzędzia zapewniane w języku Ruby na potrzeby *introspekcji* czasu wykonywania, czyli sprawdzania i modyfikowania stanu programu oraz obiektów podczas działania programu i istnienia obiektów. Język Ruby umożliwia uzyskiwanie od obiektów informacji o nich, takich jak metody, jakie mogą wykonywać w czasie działania. Kolejna informacja to liczba dostępnych „haków" w postaci metod, jakie możesz tworzyć przy użyciu specjalnych nazw zastrzeżonych w celu przechwytywania zdarzeń środowiska wykonawczego (np. dziedziczenie klas i dołączenie modułu). W tym przypadku „wkraczamy na terytorium" introspekcji dynamicznej i podejmowania decyzji, co zapewnia językowi Ruby jego charakterystyczną i znamienną jakość w postaci elastyczności i możliwości.

W rozdziale 13. zawarto też informacje i radę dotyczące procesu dokonywania zmian we własnych programach, które dotyczą podstawowych składników języka Ruby. Tak naprawdę cała niniejsza część książki uwzględnia wiele wskazówek ze sprawdzonymi praktykami (a także wskazówki zalecające unikanie części praktyk, które nie są już tak sprawdzone). Nie jest to zaskoczeniem, biorąc pod uwagę rodzaj podstawowych informacji zawartych w rozdziałach tej części. W tym przypadku tworzone programy mogą się wyróżniać w lepszym lub gorszym świetle, jeśli chodzi o charakter i jakość wykorzystania w nich zestawu dynamicznych narzędzi programistycznych języka Ruby. Zdecydowanie warto zastanowić się nie tylko nad tym, jak i dlaczego użyć w określonych sytuacjach niektórych z tych bogatych w możliwości technik, ale też *czy* należy to robić. Świadome i rozsądne stosowanie możliwości dynamicznych języka Ruby może umożliwić Ci osiągniecie nowych, fascynujących poziomów.

# 13 Indywidualizacja obiektów

**W tym rozdziale:**

- Metody i klasy pojedynczego obiektu.
- Metody klasy.
- Metoda `extend`.
- Przesłanianie działania podstawowych elementów języka Ruby.
- Klasa `BasicObject`.

Jednym z kamieni milowych projektu języka Ruby jest indywidualizacja obiektów, czyli możliwość sprawienia, że poszczególne obiekty będą zachowywać się odmiennie od innych obiektów tej samej klasy. Każdy obiekt to „pełnoprawny obywatel świata" środowiska wykonawczego programu. Z tego też powodu obiekt może funkcjonować zgodnie ze swoimi wymaganiami.

Ze swobodą obiektów dotyczącą odchodzenia od warunków ustalonych w momencie ich utworzenia związany jest swego rodzaju pierścień filozoficzny. Powoduje to istotne konsekwencje natury technicznej. Znaczna liczba elementów i właściwości języka Ruby wywodzi się z indywidualności obiektów lub jest z nią zbieżna. Spora część elementów języka Ruby została tak skonstruowana, aby umożliwić indywidualizację obiektów. Ostatecznie indywidualizacja jest ważniejsza niż inżynieria. Matz wielokrotnie powtarzał, że w przypadku zasady indywidualizacji obiektów istotne jest to, co kod Ruby implementuje, a to, jak to jest realizowane, ma drugorzędne znaczenie. Niemniej jednak z implementacją indywidualizacji obiektów związanych jest kilka przydatnych komponentów o dużych możliwościach.

W rozdziale przyjrzymy się temu, jak w języku Ruby umożliwiane jest poszczególnym obiektom uzyskiwanie metod i zachowań, a także w jaki sposób najlepiej wykorzystać elementy języka, które pozwalają określać zachowanie pojedynczych obiektów. Zaczniemy od szczegółowego omówienia *metod pojedynczego obiektu*, które należą do poszczególnych obiektów, w kontekście *klas pojedynczych obiektów* przechowujących definicje tego typu metod. Dalej zostaną zaprezentowane metody klasy, które w gruncie rzeczy są metodami pojedynczego obiektu dołączonymi do obiektów klasy. Inna kluczowa technika związana z określaniem zachowania poszczególnych obiektów oparta jest na metodzie `extend`, która pod względem działania przypomina dołączanie modułu, z tym że operacja ta jest wykonywana każdorazowo dla jednego obiektu. Dowiesz się, jak możesz wykorzystać metodę `extend` do indywidualizacji obiektów.

Być może najważniejszym zagadnieniem powiązanym w jakiś sposób z indywidualizacją obiektów jest zmiana podstawowego zachowania klas języka Ruby. Dodanie metody do już istniejącej klasy, takiej jak `Array` lub `String`, jest formą indywidualizacji obiektów, ponieważ klasy są obiektami. Jest to ryzykowna technika o dużych możliwościach. Istnieją jednak sposoby osiągnięcia tego przy stosunkowo niewielkim ryzyku, a ponadto dla poszczególnych obiektów (dodawanie zachowania do jednego łańcucha, a nie do klasy `String`). Dokonamy przeglądu podstawowych zmian dotyczących środowiska wykonawczego ze szczególnym uwzględnieniem tego, jak techniki stosowane dla pojedynczych obiektów mogą ułatwić jak najlepsze wykorzystanie czasami zaskakującego otwartego modelu klas języka Ruby.

Na końcu odnowimy wcześniejszą „znajomość", czyli wrócimy do klasy `BasicObject`. Instancje tej klasy zapewniają niemal najbardziej idealne laboratorium, jakie można sobie wyobrazić, umożliwiające tworzenie pojedynczych obiektów. Przeanalizujemy to, jak wzajemnie uzupełniają się te indywidualizacje klas i obiektów.

## 13.1. Tam, gdzie znajdują się metody pojedynczego obiektu: klasa pojedynczych obiektów

Większość działań mających miejsce w kodzie Ruby obejmuje klasy i moduły, które zawierają definicje metod instancji:

```
class C
  def talk
    puts "Witaj!"
  end
end
```

W konsekwencji działania te uwzględniają tworzenie instancji klas i wywoływanie tych metod instancji:

```
c = C.new
c.talk      ←——— Dane wyjściowe: Witaj!
```

Jak jednak wcześniej zaprezentowano (nawet jeszcze przed pokazaniem metod instancji wewnątrz klas), metody pojedynczego obiektu mogą być też definiowane bezpośrednio w poszczególnych obiektach:

```
obj = Object.new
def obj.talk
  puts "Witaj!"
end
obj.talk      ←——— Dane wyjściowe: Witaj!
```

Wspomniano również, że najpowszechniejszy typ metody pojedynczego obiektu to metoda klasy, która jest dodawana do poszczególnych obiektów `Class`:

```
class Car
  def self.makes
    %w{ Honda Ford Toyota Chevrolet Volvo }
  end
end
```

Jednak do dowolnego obiektu mogą zostać dodane metody pojedynczego obiektu (dotyczy to prawie każdego obiektu; więcej informacji znajdziesz w poniższej ramce). Możliwość definiowania zachowania dla poszczególnych obiektów to jedna z cech charakterystycznych projektu języka Ruby.

**Niektóre obiekty mogą być bardziej indywidualizowane niż inne**

Niemal każdy obiekt języka Ruby umożliwia dodanie do niego metod. Wyjątkiem są instancje określonych podklas klasy `Numeric`, w tym klasy liczb całkowitych, liczby zmiennoprzecinkowe oraz symbole. Jeśli spróbujesz wykonać następujący kod:

```
def 10.jakas_metoda; end
```

uzyskasz błąd składni. W przypadku użycia następującego kodu:

```
class << 10; end
```

zostanie wygenerowany błąd typu z komunikatem informującym o braku możliwości zdefiniowania pojedynczego obiektu. W obu przypadkach to samo dotyczy liczb zmiennoprzecinkowych i symboli.

Metody instancji (dostępne dla wszystkich bez wyjątku instancji danej klasy) istnieją w obrębie klasy lub modułu. Tam też mogą zostać znalezione przez obiekty, które mają możliwość ich wywołania. Co jednak z metodami pojedynczego obiektu? Gdzie znajduje się metoda, jeśli istnieje ona tylko w celu wywoływania jej przez pojedynczy obiekt?

### 13.1.1. Podwójne określanie za pomocą klas pojedynczych obiektów

Zgodnie z oczekiwaniami język Ruby zapewnia prostą odpowiedź na to trudne pytanie: metody pojedynczego obiektu występują w *klasie pojedynczych obiektów* danego obiektu. Ostatecznie każdy obiekt ma następujące dwie klasy:

- klasę, której jest instancją;
- własną klasę pojedynczego obiektu.

Obiekt może wywoływać metody instancji ze swojej pierwotnej klasy, a także z własnej klasy pojedynczego obiektu. Obiekt zawiera obie klasy. Wszystkie możliwości wywoływania klasy obiektu dają razem sumę wszystkich metod instancji zdefiniowanych w tych dwóch klasach wraz z metodami dostępnymi za pośrednictwem klas

nadrzędnych (superklasa klasy obiektu, superklasa tej klasy itd.) lub dowolnych modułów dołączonych do dowolnej z tych klas. Klasę pojedynczych obiektów danego obiektu możesz traktować jak specjalny magazyn metod dopasowany dla tego obiektu, który ponadto nie jest współużytkowany z innymi obiektami (nawet z innymi instancjami klasy wybranego obiektu).

### 13.1.2. Bezpośrednie sprawdzanie i modyfikowanie klasy pojedynczych obiektów

Klasy pojedynczych obiektów są anonimowe: choć są obiektami klasy (instancjami klasy `Class`), pojawiają się automatycznie bez nadawania nazwy. Niemniej jednak możesz otworzyć zawartość definicji klasy pojedynczych obiektów i dodawać do niej metody instancji, metody klasy oraz stałe tak jak w przypadku zwykłej klasy.

Umożliwia to specjalna postać słowa kluczowego `class`. Zwykle po tym słowie następuje stała:

```
class C
  # w tym miejscu znajdują się definicje stałych i metod
end
```

Aby jednak uzyskać dostęp do wnętrza definicji klasy pojedynczych obiektów, użyj specjalnej notacji:

```
class << object
  # w tym miejscu znajdują się definicje stałych i metod
end
```

Notacja `<< object` oznacza anonimową klasę pojedynczych obiektów `object`. Po uzyskaniu dostępu do zawartości definicji klasy pojedynczych obiektów możesz zdefiniować metody. Będą to metody pojedynczego obiektu, do którego wnętrza klasy pojedynczych obiektów uzyskano dostęp.

Dla przykładu przeanalizuj kod następującego programu:

```
str = "Jestem łańcuchem"
class << str
  def twice
    self + " " + self
  end
end
puts str.twice
```

Oto dane wyjściowe:

```
Jestem łańcuchem Jestem łańcuchem
```

Metoda `twice` to metoda pojedynczego obiektu łańcucha `str`. Odpowiada to dokładnie użyciu następującego kodu:

```
def str.twice
  self + " " + self
end
```

Różnica polega na tym, że uzyskano dostęp do klasy pojedynczych obiektów obiektu `str` i zdefiniowano w niej metodę.

**Różnica między def obj.meth i class << obj; def meth**

Często pojawia się następujące pytanie: czy istnieje jakaś różnica między definiowaniem metody bezpośrednio w obiekcie (za pomocą notacji `def obj.some_method`) i jawnym dodawaniem metody do klasy pojedynczych obiektów danego obiektu (przy użyciu kodu `class << obj; def some_method`)? Odpowiedź jest następująca: występuje jedna różnica w postaci odmiennego określania stałych.

Jeśli istnieje stała `N` najwyższego poziomu, możesz też zdefiniować ją wewnątrz klasy pojedynczych obiektów danego obiektu:

```
N = 1
obj = Object.new
class << obj
  N = 2
end
```

Dla następującej sekwencji instrukcji dwa sposoby dodawania do obiektu `obj` metody pojedynczego obiektu różnią się tym, jaka stała `N` jest widoczna z wnętrza definicji metody:

```
def obj.a method
  puts N
end
class << obj
  def another method
    puts N
  end
end
obj.a_metoda
obj.another_method
```

**Dane wyjściowe: 1 (stała `N` na zewnętrznym poziomie)**

**Dane wyjściowe: 2 (stała `N` należąca do klasy pojedynczych obiektów obiektu `obj`)**

Stosunkowo mało prawdopodobne będzie, że na utworzony kod będzie mieć wpływ taka różnica w widoczności stałych. W większości sytuacji dwie notacje definicji metody pojedynczego obiektu możesz traktować jako wymienne. Warto jednak wiedzieć o różnicy, ponieważ może mieć znaczenie w niektórych przypadkach, a ponadto może być pomocna w wyjaśnieniu nieoczekiwanych wyników.

Notacja `class << object` ma trochę reputację enigmatycznej lub zawiłej. Nie musi być ani taka, ani taka. Pomyśl o tym w następujący sposób: istnieje słowo kluczowe `class`, które zaakceptuje stałą lub wyrażenie `<< object`. Nowością jest tutaj pojęcie klasy pojedynczych obiektów. Gdy oswoisz się z tym, że obiekty mają klasy pojedynczych obiektów, sensowne będzie uzyskanie możliwości otwarcia tych klas za pomocą słowa kluczowego `class`. Notacja `<< object` to sposób wyrażania pojęcia „klasy pojedynczych obiektów danego obiektu”, gdy jest to wymagane przez słowo kluczowe `class`.

Jak dotąd do wprowadzania klasy metod pojedynczego obiektu notacja `class << object` jest najczęściej używana w powiązaniu z definicjami metod klasy.

### DEFINIOWANIE METOD KLASY ZA POMOCĄ NOTACJI CLASS <<

Oto idiom, z którym się często spotkasz:

```
class Ticket
  class << self
    def most_expensive(*tickets)
      tickets.max_by(&:price)
```

```
    end
  end
end
```

Kod powoduje utworzenie metody klasy `Ticket.most_expensive`, czyli dokładnie tej samej metody, którą zdefiniowano w punkcie 3.6.3, lecz tym razem użyto następującego wiersza:

```
def Ticket.most_expensive(*tickets)  # itp.
```

W obecnej wersji kodu używamy idiomu `class << object`, otwierając klasę pojedynczych obiektów danego obiektu. W tym szczególnym przypadku stosowanym obiektem jest obiekt klasy `Ticket`, który w momencie wywołania w kodzie notacji `class << self` jest wartością obiektu `self`. Wynikiem definiowania metody `most_expensive` wewnątrz bloku definicji klasy jest to, że metoda ta jest definiowana jako metoda pojedynczego obiektu `Ticket`, czyli jako metoda klasy.

Ta sama metoda klasy mogłaby zostać zdefiniowana w następujący sposób (zakładając, że kod ten występuje w programie w momencie, gdy istnieje już klasa `Ticket`):

```
class << Ticket
  def most_expensive(tickets)
  # itp.
```

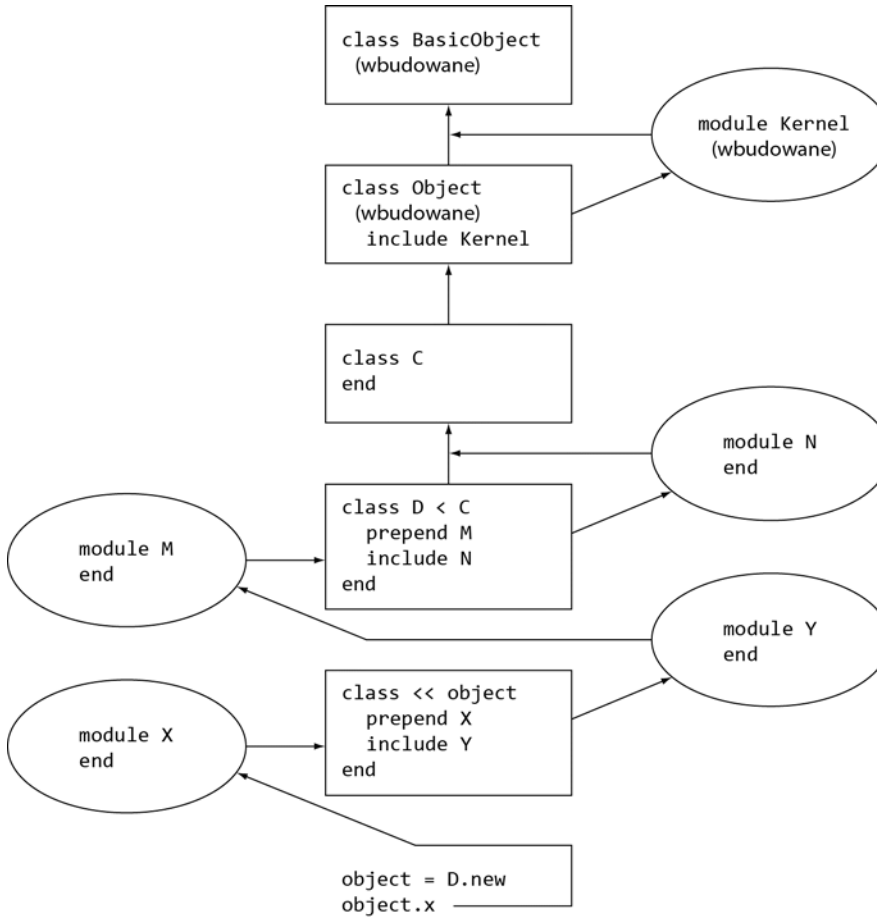
Ponieważ obiekt `self` jest obiektem `Ticket` wewnątrz definicji `class Ticket`, kod `class << self` *w* treści definicji jest taki sam jak kod `class << Ticket` *poza* obrębem zawartości definicji (z technicznego punktu widzenia możesz nawet użyć kodu `class << Ticket` wewnątrz zawartości klasy `Ticket`, ale w praktyce przeważnie spotkasz się z kodem `class << self` każdorazowo, gdy obiekt, którego klasa pojedynczych obiektów wymaga otwarcia, jest obiektem `self`).

To, że kod `class << self` pojawia się często przy tworzeniu metod klasy, powoduje czasami fałszywe wrażenie, że notacja `class << object` może być stosowana tylko do tworzenia metod klasy albo że obiekt `self` to jedyne wyrażenie, jakie może zostać umieszczone po prawej stronie. Okazuje się, że kod `class << self` wewnątrz bloku definicji klasy to tylko jeden szczególny przypadek użycia notacji `class << object`. Technika ma ogólny charakter: umożliwia przejście do bloku definicji klasy pojedynczych obiektów obiektu `object`, który może być dowolnym obiektem.

W rozdziale 4. omówiono kroki podejmowane przez obiekt podczas szukania metody wśród zdefiniowanych w swojej klasie, w klasie swojej klasy itd. Pora zająć się teraz nowym elementem, czyli klasą pojedynczych obiektów. Jaki wpływ na proces wyszukiwania metod ma ta specjalna klasa?

### *13.1.3. Klasy pojedynczych obiektów w ścieżce wyszukiwania metod*

Jak wcześniej wspomniano, wyszukiwanie metod odbywa się w górę łańcucha dziedziczenia klas z dodatkowym uwzględnieniem wszelkich modułów, które zostały dołączone. Gdy proces ten omawiano po raz pierwszy, nie wspomniano o klasach pojedynczych obiektów i metodach pojedynczego obiektu. Ponadto nie były one obecne na diagramie. Pora skorygować diagram w celu ich uwzględnienia (rysunek 13.1).




**Rysunek 13.1.** Kolejność wyszukiwania metod skorygowana w celu uwzględnienia klas pojedynczych obiektów

Prostokąt zawierający kod `class << object` reprezentuje klasę pojedynczych obiektów obiektu `object`. Szukając metody `x`, obiekt `object` szuka najpierw wszystkich modułów dołączonych do jego klasy pojedynczych obiektów. Później obiekt szuka w samej klasie pojedynczych obiektów. W dalszej kolejności obiekt przeszukuje wszystkie moduły dołączone przez klasę pojedynczych obiektów (na diagramie znajduje się jeden moduł o nazwie `N`). Proces wyszukiwania jest następnie kontynuowany w górę do oryginalnej klasy obiektu (klasa `D`) itd.

Zwróć w szczególności uwagę na to, że klasa pojedynczych obiektów ma możliwość dołączenia modułu. W końcu jest to klasa.

### DOŁĄCZANIE MODUŁU DO KLASY POJEDYNCZYCH OBIEKTÓW

Utwórzmy niewielki program ilustrujący efekt dołączenia modułu do klasy pojedynczych obiektów. Zaczniemy od prostej klasy `Person` oraz kilku jej instancji:

```
class Person
  attr_accessor :name
end
david = Person.new
david.name = "Dawid"
matz = Person.new
matz.name = "Matz"
ruby = Person.new
ruby.name = "Ruby"
```

Przyjmijmy teraz, że niektóre osoby (czyli obiekty klasy `Person`) nie lubią podawać swoich imion. Logicznym sposobem zapewnienia tego rodzaju poufności wybranym obiektom jest dodanie do każdego z nich wersji metody `name` w postaci metody pojedynczego obiektu:

```
def david.name
  "[niedostępne]"
end
```

W efekcie Matz i Ruby ujawniają swoje imiona, ale Dawid trzyma je w tajemnicy. W przypadku wywołania osób na apelu:

```
puts "Jest jedna osoba o imieniu #{matz.name}, "
puts "jedna o imieniu #{david.name},"
puts "a także jedna o imieniu #{ruby.name}."
```

zostaną uzyskane tylko dwa imiona:

```
Jest jedna osoba o imieniu Matz,
jedna o imieniu [niedostępne],
a także jedna o imieniu Ruby.
```

Na razie wszystko wygląda dobrze. Co będzie jednak, gdy więcej niż jedna osoba postanowi nie ujawniać swojego imienia? Uciążliwa byłaby konieczność tworzenia kodu definicji `def person.name` dla każdej takiej osoby.

Rozwiązaniem w tej sytuacji jest użycie modułu. Oto jego zawartość:

```
module Secretive
  def name
    "[niedostępne]"
  end
end
```

Ukryjmy teraz imię Ruby. Zamiast stosować instrukcję `def` do definiowania nowej wersji metody `name`, dołączymy moduł do klasy pojedynczych obiektów języka Ruby:

```
class << ruby
  include Secretive
end
```

Wywołanie na apelu potwierdza teraz, że imię Ruby znalazło się w obozie utajnionych imion. Ponowne wykonanie poprzednich instrukcji `puts` zapewnia następujące dane wyjściowe:

```
Jest jedna osoba o imieniu Matz,
jedna o imieniu [niedostępne],
a także jedna o imieniu [niedostępne].
```

Co się stało w przypadku imienia Ruby? Do obiektu `ruby` wysłano komunikat „imię". W efekcie obiekt rozpoczął szukanie metody. Najpierw poszukał jej w swojej własnej klasie pojedynczych obiektów, gdzie nie znalazł metody `name`. Później wyszukiwanie objęło moduły dołączone do klasy pojedynczych obiektów tego obiektu. Klasa pojedynczych obiektów obiektu `ruby` dołączyła moduł `Secretive`. Z całą pewnością moduł ten zawiera metodę instancji o nazwie `name`. W tym momencie następuje wykonanie metody.

Zrozumienie kolejności, w jakiej obiekty przeszukują swoje ścieżki wyszukiwania w celu znalezienia metod, pozwala stwierdzić, jaka wersja metody (czyli dla jakiej klasy lub modułu) zostanie znaleziona przez obiekt jako pierwsza. Przykłady również są pomocne, zwłaszcza w ilustrowaniu różnicy między dołączaniem modułu do klasy pojedynczych obiektów oraz do zwykłej klasy.

### PORÓWNANIE DOŁĄCZANIA MODUŁU DO KLASY POJEDYNCZYCH OBIEKTÓW ORAZ DO ORYGINALNEJ KLASY

W przypadku dołączania modułu do klasy pojedynczych obiektów danego obiektu masz do czynienia z konkretnym obiektem. Metody uzyskiwane przez obiekt z modułu mają pierwszeństwo nad wszelkimi metodami o takiej samej nazwie należącymi do oryginalnej klasy obiektu. Listing 13.1 prezentuje mechanizm i wynik przeprowadzenia tego rodzaju operacji `include`.

**Listing 13.1. Dołączanie modułu do klasy pojedynczych obiektów**

```
class C
  def talk
    puts "Powitanie oryginalnej klasy!"
  end
end
module M
  def talk
    puts "Powitanie modułu!"
  end
end
c = C.new
c.talk          ←❶
class << c
  include M     ←❷
end
c.talk          ←❸
```

Dane wyjściowe kodu z tego listingu są następujące:

```
Powitanie oryginalnej klasy!
Powitanie modułu!
```

Pierwsze wywołanie metody `talk` ❶ powoduje wykonanie metody instancji `talk` zdefiniowanej w klasie `C` obiektu `c`. Następnie do klasy pojedynczych obiektów obiektu `c` ❷ dołączany jest moduł `M`, który również definiuje metodę o nazwie `talk`. W efekcie przy kolejnym wywołaniu metody `talk` w obiekcie `c` ❸ wykonywana jest jej wersja (napotykana przez obiekt `c` jako pierwsza) zdefiniowana w module `M`.

Wszystko sprowadza się do tego, jak w ścieżce wyszukiwania metod obiektu są rozmieszczone klasy i moduły. Moduły dołączone do klasy pojedynczych obiektów występują przed oryginalną klasą, a także przed wszelkimi dołączonymi do niej modułami.

Graficzną ilustrację tego zapewnia metoda `ancestors`, która zwraca listę klas i modułów w ramach hierarchii dziedziczenia i dołączania dowolnej klasy lub modułu. Począwszy od miejsca występującego po definicjach klas i modułów z poprzedniego przykładu, za pomocą metody `ancestors` spróbuj sprawdzić, jak wygląda taka hierarchia:

```
c = C.new
class << c
  include M
  p ancestors
end
```

Jako wynik uzyskasz tablicę elementów nadrzędnych. Zasadniczo jest to ścieżka wyszukiwania metod dla instancji przykładowego obiektu `c`. Ponieważ jest to klasa pojedynczych obiektów, szukanie jej klas nadrzędnych oznacza wyszukiwanie w ścieżce wyszukiwania metod dla obiektu `c`. Zauważ, że klasa pojedynczych obiektów obiektu `c` występuje na liście klas nadrzędnych jako pierwsza:

```
[#<Class:#<C:0x007fbc8b9129f0>>, M, C, Object, Kernel, BasicObject]
```

Przyjrzyj się teraz temu, co się stanie, gdy moduł `M` nie tylko zostanie dołączony do klasy pojedynczych obiektów obiektu `c`, ale też do klasy `C` tego obiektu. Wykorzystujemy poprzedni przykład:

```
class C
  include M
end
class << c
  p ancestors
end
```

Tym razem uzyskujemy następujący wynik:

```
[#<Class:#<C:0x007fbc8b9129f0>>, M, C, M, Object, Kernel, BasicObject]
```

Moduł `M` pojawia się dwa razy! Dwie różne klasy, klasa pojedynczych obiektów obiektu `c` i klasa `C`, dołączyły ten moduł. Każde dołączenie to osobna transakcja. Każda klasa realizuje to niezależnie. Klasy nie konsultują tego ze sobą (możesz nawet dołączyć moduł `M` do klasy `Object`; w tym przypadku moduł ten pojawiłby się trzykrotnie na liście elementów nadrzędnych).

Zachęcamy do skorzystania z tych przykładów, modyfikowania ich na różne sposoby i sprawdzania wyników. Klasy również są obiektami, dlatego sprawdź, co się stanie po użyciu klasy pojedynczych obiektów należącej do klasy pojedynczych obiektów danego obiektu. A jak przedstawia się kwestia dołączania modułów do innych modułów? Wypróbuj kilka przykładów z instrukcją `prepend`. Możliwych jest wiele kombinacji. Drogą eksperymentowania możesz wiele się dowiedzieć, bazując na przedstawionym tutaj materiale.

Podstawową informacją jest to, że zachowanie poszczególnych obiektów języka Ruby oparte jest na tej samej zasadzie co zachowanie zwykłego obiektu wywodzącego się z klasy. Dotyczy to definiowania metod instancji w klasach i modułach, dołączania modułów do klas oraz przechodzenie ścieżki wyszukiwania metod składającej się z klas i modułów. Jeśli opanujesz te zagadnienia i powrócisz do nich, gdy tylko coś będzie się wydawać niejasne, z pewnością zdobyta wiedza pozwoli rozwiać wątpliwości.

### 13.1.4. Metoda singleton_class

Aby odwołać się bezpośrednio do klasy pojedynczych obiektów obiektu, użyj metody `singleton_class`. Pozwala ona ograniczyć liczbę cykli związanych z użyciem notacji `class << object`.

Oto przykład zastosowania tej metody do uzyskania elementów nadrzędnych klasy pojedynczych obiektów obiektu:

```
string = "łańcuch"
p string.singleton_class.ancestors
```

Cofnijmy się teraz i przyjrzyjmy szczególnemu przypadkowi ze świata metod pojedynczego obiektu (szczególnemu ze względu na jego powszechność i przydatność). Mowa o metodach klasy.

### 13.1.5. Obszerne omówienie metod klasy

Metody klasy są metodami pojedynczego obiektu definiowanymi w obiektach klasy `Class`. Pod wieloma względami zachowują się one podobnie do dowolnej innej metody pojedynczego obiektu:

```
class C
end
def C.a_class_method
  puts "Metoda pojedynczego obiektu zdefiniowana w klasie C"
end
C.a_class_method   ←——— Dane wyjściowe: metoda pojedynczego obiektu zdefiniowana w klasie C
```

Jednak metody klasy przejawiają też specjalne zachowanie. Standardowo, gdy w obiekcie definiowana jest metoda pojedynczego obiektu, żaden inny obiekt nie może być odbiorcą w wywołaniu tej metody (właśnie to sprawia, że metody pojedynczego obiektu są takie, a nie inne). Metody klasy są trochę inne: metoda zdefiniowana jako metoda pojedynczego obiektu dla obiektu klasy może też być wywoływana w podklasach tej klasy. W przypadku poprzedniego przykładu z klasą `C` możliwe jest użycie następującego kodu:

```
class D < C
end
D.a_class_method
```

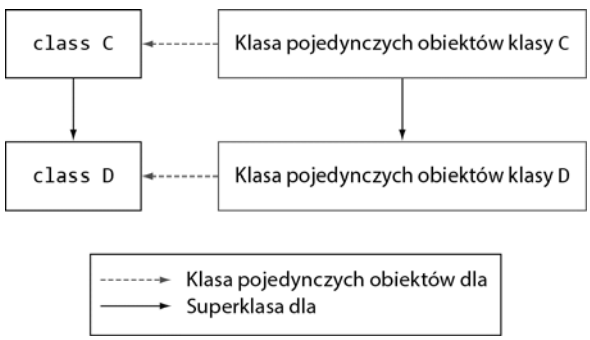
Oto raczej dezorientujące dane wyjściowe (dlatego że obiekt klasy, do którego wysłano komunikat, to klasa `D`, a nie klasa `C`):

```
Metoda pojedynczego obiektu zdefiniowana w klasie C
```

Dzięki specjalnej konfiguracji obejmującej klasy pojedynczych obiektów dla obiektów klasy możliwe jest wywołanie metod pojedynczego obiektu klasy `C` w jej podklasie, a nie tylko w klasie `C`. W przykładzie klasa pojedynczych obiektów klasy `C` (znajduje się w niej metoda `a_class_method`) jest uważana za superklasę klasy pojedynczych obiektów klasy `D`.

Po wysłaniu komunikatu do obiektu klasy `D` ma miejsce przechodzenie standardowej ścieżki wyszukiwania, z tym wyjątkiem, że po klasie pojedynczych obiektów klasy `D` wyszukiwanie dotyczy jej superklasy. Jest to zawierająca metodę klasa pojedynczych obiektów superklasy klasy `D`.

Rysunek 13.2 przedstawia relacje między klasami w hierarchii dziedziczenia oraz ich klasami pojedynczych obiektów.




**Rysunek 13.2.** Relacje między klasami w hierarchii dziedziczenia oraz ich klasami pojedynczych obiektów

Jak widać na rysunku 13.2, klasa pojedynczych obiektów klasy podrzędnej `D` klasy `C` jest traktowana jak klasa podrzędna (podklasa) klasy pojedynczych obiektów klasy `C`.

Klasy pojedynczych obiektów dla obiektów klasy są czasem nazywane *metaklasami*. Może się zdarzyć, że usłyszysz o stosowaniu tego terminu do klas pojedynczych obiektów w ogólności, choć słowo *meta* nieszczególnie pasuje do tych klas. Klasa pojedynczych obiektów to bardziej ogólny termin opisowy.

Objaśnienie to możesz traktować jako dodatkowy temat. Mało prawdopodobne jest to, że często będzie się pojawiać pilna potrzeba zrozumienia tego. Niemniej jednak jest to znakomity przykład tego, jak projekt języka Ruby oparty jest na stosunkowo niewielkiej liczbie reguł (takich jak posiadanie przez każdy obiekt klasy pojedynczych obiektów oraz sposób wyszukiwania metod). Klasy to szczególny przypadek obiektów: są przecież fabrykami obiektów, a także pełnoprawnymi obiektami. Niewiele jednak jest w języku Ruby elementów, które nie bazują w naturalny sposób na podstawowych zasadach projektu języka (dotyczy to nawet szczególnych przypadków).

Ponieważ klasy i moduły języka Ruby są obiektami, dokonywane w nich zmiany są wprowadzane dla poszczególnych obiektów. Oznacza to, że dyskusja dotycząca tego, jak, kiedy i czy dokonywać modyfikacji w podstawowych klasach i modułach języka Ruby, stanowi część omówienia indywidualizacji obiektów. W dalszej kolejności zostaną objaśnione zmiany dotyczące podstawowych składników.

**KLASY POJEDYNCZYCH OBIEKTÓW I WZORZEC POJEDYNCZEGO OBIEKTU**
Termin „pojedynczy obiekt" ma drugie, odmienne znaczenie w języku Ruby (i gdziekolwiek indziej): odnosi się do wzorca pojedynczego obiektu, który opisuje klasę mającą tylko jedną instancję. Biblioteka standardowa języka Ruby obejmuje implementację wzorca pojedynczego obiektu (dostępna za pośrednictwem polecenia `require 'singleton'`). Pamiętaj o tym, że klasy pojedynczych obiektów nie są bezpośrednio powiązane z wzorcem pojedynczego obiektu. Z terminem „pojedynczy obiekt" po prostu związanych jest trochę za dużo informacji. Zamierzone znaczenie tego terminu można przeważnie odczytać z kontekstu.

## 13.2. Modyfikowanie podstawowych klas i modułów języka Ruby

Otwartość klas i modułów języka Ruby (oznacza to, że programista może „zajrzeć do wnętrza" języka i zmienić sposób jego działania) to jedna z jego najważniejszych cech, a także jedna z najtrudniejszych, z jakimi trzeba się pogodzić. Można to porównać z możliwością zjedzenia w restauracji talerzy razem z daniem. Jak stwierdzisz, gdzie jedno się kończy, a drugie zaczyna? Jak określisz moment, w którym trzeba przerwać? Czy możesz też zjeść obrus?

Nauka sposobu radzenia sobie z otwartością języka Ruby ma trochę wspólnego z techniką programowania, a wiele ze sprawdzonymi praktykami. Nie jest skomplikowane wprowadzanie modyfikacji w podstawowych składnikach języka. Trudnym elementem jest stwierdzenie, kiedy można to robić, kiedy nie powinno się, a także w jaki sposób przeprowadzić to bezpiecznie.

W tym podrozdziale przyjrzymy się „krajobrazowi" podstawowych zmian. Odpowiemy na pytania: jak, co i dlaczego (oraz dlaczego nie). Omówimy znaczące pułapki, możliwe korzyści oraz sposoby postrzegania obiektów i ich zachowań, które umożliwiają zapewnienie sobie najlepszego z obu światów, czyli elastyczności i bezpieczeństwa.

Zaczniemy od porcji powiastek umoralniających.

### 13.2.1. Zagrożenia związane z modyfikowaniem podstawowych funkcjonalności

Problem z wprowadzaniem zmian w podstawowych klasach języka Ruby polega na tym, że są one globalne: dopóki program działa, będą obowiązywać dokonane zmiany. Jeśli zmienisz sposób działania metody, która jest też używana gdzie indziej (w obrębie samego języka Ruby lub w ładowanej bibliotece), dojdzie do destabilizacji całego interpretera w wyniku zmiany reguł gry w jej trakcie.

Niemniej jednak kuszące jest dostosowanie języka Ruby do własnych potrzeb przez globalne zmodyfikowanie podstawowych metod. Ostatecznie jest to możliwe. Jednak jest to najmniej bezpieczny i najmniej wskazany sposób dostosowywania zachowań podstawowych obiektów. Zajmujemy się tym tylko w celu umożliwienia zorientowania się w naturze problemu.

Klasa `Regexp` to jeden z często cytowanych kandydatów do wprowadzenia zmiany doraźnej.

### MODYFIKOWANIE METODY REGEXP#MATCH (I DLACZEGO NIE NALEŻY TEGO ROBIĆ)

Jak wspomniano w rozdziale 11., gdy nie powiedzie się operacja dopasowania bazująca na metodzie `match`, zostanie zwrócony obiekt `nil`. W przypadku powodzenia operacji wynikiem będzie obiekt `MatchData`. Taki wynik jest irytujący, ponieważ obiekt `nil` nie pozwala na zrealizowanie tego samego, co umożliwia obiekt `MatchData`.

Na przykład następujący kod zakończy się powodzeniem, jeśli pierwsze przechwycenie zostanie utworzone przez dopasowanie:

```
some_regexp.match(some_string)[1]
```

Jeżeli jednak nie wystąpi dopasowanie, zostanie zwrócony obiekt `nil`. Ponieważ nie zawiera on metody `[]`, przy próbie wykonania operacji `[1]` zostanie zgłoszony wyjątek krytyczny `NoMethodError`:

```
string = "Łańcuch testowy"
re = /Przykładowy łańcuch/
substring = re.match(string)[1]   <------ NoMethodError: undefined method [ ] for nil:NilClass
```

W celu uniknięcia błędu kuszące może być zastosowanie następującego kodu:

```
class Regexp
  alias __old_match__ match        <------ ❶
  def match(string)
    __old_match__(string) || []
  end
end
```

Kod ten ustawia najpierw alias dla metody `match` dzięki słowu kluczowemu `alias` ❶. Później kod zmienia definicję tej metody. Nowa wersja metody `match` łączy się z jej oryginalną wersją (za pomocą aliasu), a następnie zwraca wynik wywoływania oryginalnej wersji (jeśli zwraca ono obiekt `nil`) lub pustą tablicę.

> **UWAGA** *Alias* to synonim nazwy metody. Wywoływanie metody za pomocą aliasu nie obejmuje żadnej zmiany zachowania ani żadnej modyfikacji procesu wyszukiwania metody. Wybór nazwy aliasu w poprzednim przykładzie oparty jest na dość konwencjonalnej formule, czyli dodaniu słowa *old* oraz początkowych i końcowych znaków podkreślenia (warto zaznaczyć, że formuła ta jest zbyt popularna, dlatego należy tworzyć nazwy, które z mniejszym prawdopodobieństwem zostaną wybrane przez innych programistów stosujących zmiany i również znających tę formułę!).

Możesz teraz użyć następującego wiersza:

```
/abc/.match("X")[1]
```

Nawet pomimo tego, że nie wystąpi dopasowanie, program nie przestanie działać, ponieważ brak dopasowania powoduje teraz zwrócenie pustej tablicy, a nie obiektu `nil`.

Najgorszą rzeczą, jaką możesz zrobić w przypadku nowej metody `match`, jest podjęcie próby indeksowania pustej tablicy, co jest dozwolone (wynikiem takiej operacji będzie obiekt `nil`, ale przynajmniej nie jest podejmowana próba indeksowania tego obiektu).

Problem polega na tym, że osoba korzystająca z Twojego kodu może oczekiwać, że w przypadku niepowodzenia operacja dopasowania zwróci obiekt `nil`:

```
if regexp.match(string)
  do cos
else
  do cos innego
end
```

Ze względu na to, że tablica (nawet pusta) zapewnia wartość prawda, natomiast obiekt `nil` oznacza fałsz, zwracanie tablicy dla nieudanej operacji dopasowania oznacza, że test prawda/fałsz (co jest odzwierciedlone w instrukcji `if/else`) zawsze zwraca wartość prawda.

Być może modyfikowanie metody `Regexp#match` w taki sposób, aby nie zwracała obiektu `nil` w przypadku niepowodzenia, jest czymś, co i tak sam uznasz za niewskazane. Nikt nie będzie tego popierać. Co więcej, niektórzy początkujący użytkownicy języka Ruby nie rozumieją istniejących relacji, dlatego nie zauważą, że zmiana podstawowej metody w jednym miejscu spowoduje jej zmianę w każdym innym miejscu.

Inny typowy przykład, który jest trochę bardziej subtelny (zarówno pod względem tego, co w nim jest realizowane, jak i z powodów, dla których kod z przykładu nie jest dobrym pomysłem), uwzględnia metodę `String#gsub!`.

### WARTOŚĆ ZWRACANA METODY STRING#GSUB! I POWODY, DLA KTÓRYCH WARTOŚĆ NIE POWINNA SIĘ ZMIENIĆ

Jak już wspomniano, metoda `String#gsub!` wykonuje dla swojego odbiorcy globalną operację zastąpienia, zapisując zmiany w oryginalnym obiekcie:

```
>> string = "Witaj!"
=> "Witaj!"
>> string.gsub!(/i/, "I")
=> "WItaj!"          ←❶
>> string
=> "WItaj!"          ←❷
```

Jak widać, wartość zwracana wywołania metody `gsub!` jest obiektem łańcucha z wprowadzonymi zmianami ❶. Ponowne sprawdzenie obiektu za pomocą zmiennej `string` potwierdza, że zmiany faktycznie są trwałe ❷.

Interesujące jest jednak to, że coś innego ma miejsce, gdy operacja metody `gsub!` nie powoduje dokonania żadnych zmian w łańcuchu:

```
>> string = "Witaj!"
=> "Witaj!"
>> string.gsub!(/zzz/, "xxx")
=> nil
>> string
=> "Witaj!"
```

Ponieważ nie ma dopasowania dla wyrażenia `/zzz/`, łańcuch nie jest modyfikowany, a wartością zwracaną wywołania metody `gsub!` jest obiekt `nil`.

Podobnie jak obiekt `nil` zwracany przez operację dopasowania obiekt ten zwracany przez metodę `gsub!` ma potencjał do działań destrukcyjnych, gdy raczej nie tego oczekujesz. Dokładniej rzecz biorąc, oznacza to, że nie możesz wiarygodnie używać metody `gsub!` w łańcuchu metod:

```
>> string = "Witaj!"
=> "Witaj!"
>> string.gsub!(/i/, "I").reverse!   ←❶
=> "!jatIW"                          ←❷
>> string = "Witaj!"
=> "Witaj!"
>> string.gsub!(/zzz/, "xxx").reverse!
NoMethodError: undefined method `reverse!' for nil:NilClass   ←❸
```

W tym przykładzie dwukrotnie ma miejsce coś podobnego (ale nie identycznego). Za pierwszym razem połączone w łańcuch wywołania metod `gsub!` i `reverse!` zwracają odwrócony łańcuch, który na nowo został przetworzony przez metodę `gsub!` ❷. Jednak za drugim razem łańcuch wywołań powoduje błąd krytyczny ❸: wywołanie metody `gsub!` nie zmieniło łańcucha, dlatego zwróciło obiekt `nil`. Oznacza to, że metoda `reverse!` została wywołana dla obiektu `nil`, a nie dla łańcucha.

**Metoda tap**

Metoda `tap` (możliwa do wywołania dla dowolnego obiektu) wykonuje trochę dziwne, ale potencjalnie przydatne zadanie, które polega na wykonaniu bloku kodu, przekazaniu odbiorcy blokowi i zwróceniu odbiorcy. Łatwiej będzie zaprezentować to na przykładzie, niż opisywać:

```
>> "Witaj".tap {|string| puts string.upcase }.reverse
WITAJ
=> "jatiW"
```

Wywoływana dla odbiorcy `"Witaj"` metoda `tap` przekazuje ten łańcuch swojemu blokowi kodu, co potwierdza wyświetlenie wersji łańcucha w wersji złożonej z dużych liter. Metoda ta zwraca następnie cały łańcuch. Dla łańcucha wykonywana jest operacja odwracania. Jeśli wewnątrz bloku metody `tap` wywołasz metodę `gsub!` dla łańcucha, nie będzie mieć znaczenia to, czy blok zwróci obiekt `nil`, ponieważ metoda `tap` zwraca łańcuch. Zachowaj jednak ostrożność. Użycie tej metody do udaremnienia zwrócenia obiektu `nil` przez metodę `gsub!` (lub inne podobnie zachowujące się metody z wykrzyknikiem w nazwie) samo w sobie może wywołać komplikacje, zwłaszcza w przypadku stosowania wielu łańcuchów metod, gdy część z nich realizuje operacje wewnętrzne, a część zwraca kopie obiektów.

Możliwym sposobem poradzenia sobie z niedogodnością w postaci konieczności wyeliminowania zwracania obiektu `nil` przez metodę `gsub!` jest przyjęcie wariantu, który jednak nie jest zwykle zbyt odpowiedni do tworzenia łańcuchów z wywołań metod. Jeśli nie użyjesz łańcuchów, unikniesz związanych z nimi problemów:

```
>> string = "Witaj!"
=> "Witaj!"
>> string.gsub!(/zzz/, "xxx")
```

```
=> nil
>> string.reverse!
=> "!jatiW"
```

Jednak pewnej liczbie osób używających języka Ruby dała się we znaki wartość zwracana `nil` albo dlatego, że oczekiwały one, że metoda `gsub!` będzie się zachowywać jak metoda `gsub` (wersja metody bez wykrzyknika w nazwie, która zawsze zwraca swojego odbiorcę, niezależnie od tego, czy wystąpiła zmiana, czy nie), albo z tego powodu, że nie przewidziały sytuacji, w której łańcuch nie zostanie zmieniony. A zatem metoda `gsub!` i jej wartość zwracana `nil` stały się popularnym kandydatem do zmiany.

Zmiana może zostać osiągnięta w następujący sposób:

```
class String
  alias __old_gsub_bang__ gsub!
  def gsub!(*args, &block)
    __old_gsub_bang__(*args, &block)
    self
  end
end
```

Pierwsza oryginalna metoda `gsub!` uzyskuje alias, który umożliwi wywołanie oryginalnej wersji w obrębie nowej wersji. Nowa metoda `gsub!` pobiera dowolną liczbę argumentów (same argumenty nie są istotne; będą on przekazywane starej metodzie `gsub!`) oraz blok kodu, który będzie przechwytywany w zmiennej `block`. Jeśli nie podano bloku, a metoda `gsub!` może być wywoływana z blokiem lub bez niego, zmienna `block` ma wartość `nil`.

Wywołujemy starą wersję metody `gsub!`, przekazując jej argumenty i ponownie używając bloku kodu. Na końcu nowa wersja metody `gsub!` realizuje działanie, dla którego została utworzona: zwraca obiekt `self` (łańcuch) niezależnie od tego, czy wywołanie metody `__old_gsub_bang__` spowodowało zwrócenie łańcucha lub obiektu `nil`.

Pora przedstawić powody, dla których nie należy tego robić.

Zmiana metody `gsub!` w ten sposób raczej z mniejszym prawdopodobieństwem — co wynika ze statystyki — spowoduje problemy niż zmiana metody `Regexp#match`. Możliwe jest jednak, że ktoś napisze kod zależny od udokumentowanego zachowania metody `gsub!`, a zwłaszcza od zwracania obiektu `nil`, gdy łańcuch nie zostanie zmieniony. Poniżej zamieszczono przykład. Choć jest on zaaranżowany (jak większość przykładów takiego scenariusza), zawiera poprawny kod Ruby zależny od udokumentowanego zachowania metody `gsub!`.

```
>> states = { "NY" => "Nowy Jork", "NJ" => "New Jersey", "ME" => "Maine" }   ❶
=> {"NY"=>"Nowy Jork", "NJ"=>"New Jersey", "ME"=>"Maine"}
>> string = "Wschodnie stany obejmują NY, NJ i ME."   ❷
=> "Wschodnie stany obejmują NY, NJ i ME."
>> if string.gsub!(/\b([A-Z]{2})\b/) { states[$1] }   ❸
>>   puts "Wystąpiło zastąpienie"
>> else
?>   puts "Łańcuch nie został zmieniony"
>> end
Wystąpiło zastąpienie   ❹
```

Na początku określono tablicę asocjacyjną skrótów stanów i ich pełnych nazw ❶. Dalej w kodzie występuje łańcuch używający skrótów stanów ❷. Celem jest zastąpienie skrótów pełnymi nazwami za pomocą operacji metody `gsub!`, która przechwytuje dowolne dwie kolejne duże litery otoczone ogranicznikami słowa (`\b`), a następnie zastępuje je wartością z tablicy asocjacyjnej odpowiadającą podłańcuchowi dwuliterowemu ❸. W trakcie wykonywania tych działań potwierdzane jest, czy takie zastąpienia miały miejsce. Jeśli tak, metoda `gsub!` zwraca nową wersję zmiennej `string`. Jeżeli nie wystąpiły żadne zastąpienia, metoda ta zwraca obiekt `nil`. Na końcu wykonywania kodu wyświetlany jest wynik całego procesu ❹.

Pojawiająca się tutaj szkoda jest stosunkowo niewielka, ale wynikająca z tego nauka jest oczywista: nie zmieniaj udokumentowanego zachowania podstawowych metod języka Ruby. Poniżej zaprezentowano kolejną wersję przykładu z tablicą asocjacyjną stanów, w którym użyto metody `sub!` zamiast metody `gsub!`. W tej wersji niepowodzenie operacji zwracania obiektu `nil`, gdy łańcuch nie zostanie zmieniony, powoduje wygenerowanie pętli nieskończonej. Zakładając, że istnieje tablica asocjacyjna stanów oraz oryginalna wersja obiektu `string`, możemy wykonać operacje zastąpienia (po jednej na raz), w przypadku których raportowane jest każde zastąpienie:

```
>> string = " Wschodnie stany obejmują NY, NJ i ME."
=> "Wschodnie stany obejmują NY, NJ i ME."
>> while string.sub!(/\b([A-Z]{2})\b/) { states[$1] }
>>   puts "Zastąpienie stanu #{$1} stanem #{states[$1]}..."
>> end
Zastąpienie stanu NY stanem Nowy Jork...
Zastąpienie stanu NJ stanem New Jersey...
Zastąpienie stanu ME stanem Maine...
```

Jeśli metoda `string.sub!` zawsze zwraca wartość inną niż `nil` (łańcuch), warunek instrukcji `while` zawsze zostanie spełniony, a pętla będzie wykonywana bez końca.

A zatem *nie* powinno się modyfikować podstawowych metod, aby nie zapewniały działań, jakie są oczekiwane przez innych. Nie ma tutaj żadnego wyjątku. Jest to coś, czego nigdy nie należy robić, nawet pomimo tego, że jest to możliwe.

W ten sposób zostajemy z kwestią dotyczącą tego, jak bezpiecznie zmieniać funkcjonalność podstawowych elementów języka Ruby. Przyjrzymy się czterem technikom możliwym do rozważenia. Pierwsze trzy to zmiana addytywna, zmiana z przekazaniem lub dołączeniem oraz zmiana poszczególnych obiektów. Tylko jedna z tych technik jest naprawdę bezpieczna, choć wszystkie trzy są na tyle pewne, że mogą być stosowane w wielu sytuacjach. Czwarta technika to *doprecyzowania*, które są zmianami o zasięgu modułu dotyczącymi klas. Mogą one być pomocne w określeniu własnych, podstawowych zmian dotyczących kodu Ruby tak, aby nie objęły one sąsiadującego kodu i elementów samego języka Ruby.

Przyjrzymy się także specjalnie dostosowanym przykładom oraz wybranym przykładom z biblioteki Active Support, która zwykle używana jest jako część środowiska projektowania aplikacji internetowych Rails. Biblioteka zapewnia odpowiednie przykłady pierwszych dwóch rodzajów zmiany podstawowych elementów języka, czyli zmiany addytywnej i zmiany z przekazaniem. Zaczniemy od techniki addytywnej.

### 13.2.2. Zmiany addytywne

Najpowszechniejsza kategoria zmian klas wbudowanych języka Ruby to *zmiana addytywna* polegająca na dodawaniu metody, która nie istnieje. Zaletą zmiany addytywnej jest to, że nie ma negatywnego wpływu na istniejące metody języka Ruby. Związane z tym zagrożenie jest takie, że jeśli dwóch programistów utworzyło dodawane metody o takiej samej nazwie, a ponadto obie dołączono do interpretera podczas wykonywania określonej biblioteki lub programu, jedna z tych metod wpłynie niekorzystnie na drugą. Nie ma możliwości całkowitego wyeliminowania tego zagrożenia.

Dodawane metody często służą do zapewniania funkcjonalności pożądanych przez dużą liczbę osób. Inaczej mówiąc, nie są one tworzone na potrzeby specjalnego użytku w jednym programie. Bezpieczeństwo tkwi w liczbach: jeśli dana metoda od lat jest tematem dyskusji wielu osób, a ponadto faktyczna implementacja metody funkcjonuje w świecie języka Ruby, są spore szanse na to, że jeśli utworzysz metodę lub użyjesz istniejącej implementacji, nie spowodujesz tym kolizji z czymkolwiek, co zostało stworzone przez kogoś innego.

Biblioteka Active Support, a szczególnie jej podstawowa podbiblioteka rozszerzeń, dodaje wiele metod do podstawowych klas języka Ruby. Dodatki do klasy `String` zapewniają kilka dobrych przykładów. Biblioteka ta oferuje zestaw „odmian" dotyczących klasy `String`, z takimi metodami jak `pluralize` i `titleize`. Oto kilka przykładów (w celu zastosowania ich musisz wykonać polecenie `gem install activesupport`, jeśli nie zainstalowano jeszcze pakietu *gem*):

```
>> require 'active_support/core_ext'
=> true
>> "person".pluralize
=> "people"
>> "little_dorritt".titleize
=> "Little Dorritt"
```

Każdorazowo przy dodawaniu nowych metod do podstawowych klas języka Ruby pojawia się ryzyko, że ktoś inny doda metodę o takiej samej nazwie, która zachowuje się trochę odmiennie. Biblioteka, taka jak Active Support, jest zależna od dobrej wiary jej użytkowników, a także od własnej reputacji: jeśli używasz tej biblioteki, prawdopodobnie wiesz, że zawierasz coś w rodzaju niepisanej umowy, w ramach której nie będą przesłaniane jej metody ani ładowane inne biblioteki, które do tego doprowadzą. W tym sensie biblioteka Active Support jest chroniona przez jej własną reputację oraz szeroki zakres zastosowań. Z pewnością możesz z niej skorzystać, jeśli zapewni coś, czego wymagasz, ale nie traktuj tego jak sygnału, że generalnie nie ma przeciwwskazań do dodawania metod do klas podstawowych. W przypadku decydowania się na coś takiego trzeba być dość powściągliwym.

Innym sposobem dodawania funkcjonalności do istniejących klas i modułów języka Ruby jest technika zmiany z przekazaniem lub dołączeniem pasywnym.

### 13.2.3. Zmiany z przekazaniem

Zmiana metody *z przekazaniem* polega na nadpisaniu istniejącej metody w taki sposób, że oryginalna wersja metody jest ostatecznie wywoływana wraz z nową wersją. Nowa wersja realizuje niezbędne działania, a następnie przekazuje swoje argumenty

oryginalnej wersji metody. Aby zapewnić wartość zwracaną, nowa wersja bazuje na oryginalnej metodzie (jak wiesz z przykładów przesłaniania metod `match` i `gsub!`, wywołanie oryginalnej wersji metody nie wystarczy, jeśli zamierzasz zmienić podstawowy interfejs metody, modyfikując jej wartość zwracaną).

Zmiany z przekazaniem mogą zostać wykorzystane do wielu celów, w tym rejestrowania i debugowania:

```
class String
  alias __old_reverse__ reverse
  def reverse
    $stderr.puts "Odwracanie łańcucha!"
    __old_reverse__
  end
end
puts "Dawid".reverse
```

Dane wyjściowe tego fragmentu kodu są następujące:

```
Odwracanie łańcucha!
diwaD
```

Pierwszy wiersz jest kierowany do stałej `STDERR`, a drugi wiersz do stałej `STDOUT`. Przykład jest zależny od utworzenia aliasu dla oryginalnej metody `reverse`, a następnie wywołania tego aliasu na końcu nowej metody `reverse`.

**Tworzenie aliasu i związane z tym aliasy**

Oprócz słowa kluczowego `alias` język Ruby zapewnia metodę o nazwie `alias method`, która jest prywatną metodą instancji kolekcji `Module`. Oznacza to, że możesz utworzyć alias dla metody na jeden z następujących sposobów:

```
class String
  alias   old reverse   reverse
end
```

lub

```
class String
  alias method :  old reverse  , :reverse
end
```

Ponieważ jest to metoda, a nie słowo kluczowe, jako swoich argumentów metoda `alias_method` wymaga obiektów zamiast samych nazw metod. Metoda ta może pobierać symbole lub łańcuchy. Zauważ też, że argumenty słowa kluczowego `alias` nie są rozdzielone przecinkiem. Słowa kluczowe zezwalają na to, natomiast metody nie.

Możliwe jest utworzenie metod, które łączą w sobie filozofie zmiany addytywnej i zmiany z przekazaniem. Kilka przykładów z biblioteki Active Support demonstruje, jak to zrealizować.

### HYBRYDY ŁĄCZĄCE ZMIANĘ ADDYTYWNĄ I ZMIANĘ Z PRZEKAZANIEM

*Hybryda w postaci zmiany addytywnej i zmiany z przekazaniem* jest metodą o takiej samej nazwie co istniejąca metoda podstawowa. Hybryda wywołuje starą wersję metody (nie jest ona zatem całkowitym zastąpieniem) i dodaje coś do interfejsu metody. Inaczej mówiąc, jest to nadpisanie oferujące nadzbiór funkcjonalności oryginalnej metody.

Biblioteka Active Support oferuje kilka metod będących hybrydami łączącymi zmianę addytywną i zmianę z przekazaniem. Dobrym przykładem jest metoda `to_s` klasy `Time`. W niezmienionej postaci metoda `Time#to_s` zapewnia czytelny dla użytkownika łańcuch, który reprezentuje czas:

```
>> Time.now.to_s
=> "2013-12-31 08:37:32 -0500"
```

Biblioteka Active Support rozszerza metodę w taki sposób, że może ona pobrać argument wskazujący specyficzny rodzaj formatowania. Możesz na przykład (przy założeniu, że dostępny jest wymagany moduł `active_support`) sformatować obiekt `Time` w sposób odpowiedni pod kątem operacji wstawiania danych do bazy danych:

```
>> Time.now.to_s(:db)
=> "2013-12-31 08:37:40"
```

Aby data była reprezentowana jako liczba, zażądaj formatu `:number`:

```
>> Time.now.to_s(:number)
=> " 20131231083748 "
```

Argument `:rfc822` pobiera czas sformatowany zgodnie ze standardem RFC822 obowiązującym dla formatu dat w nagłówkach wiadomości e-mail. Pod względem działania przypomina to metodę `Time#rfc822`:

```
>> Time.now.to_s(:rfc822)
=> "Tue, 31 Dec 2013 08:38:00 -0500"
```

Różne formaty dodawane do metody `Time#to_s` korzystają z metody `strftime`, która opakowuje wywołanie systemowe o takiej samej nazwie, a ponadto umożliwia formatowanie czasu na wiele sposobów. A zatem nie ma niczego w zmodyfikowanej metodzie `Time#to_s`, czego nie mógłbyś sam zrobić. Opcjonalny argument jest dodawany dla Twojej wygody (i oczywiście przyjazny dla bazy danych format `:db` jest pożądany głównie wtedy, gdy biblioteki Active Support używasz w połączeniu z biblioteką obiektowo-relacyjną, taką jak `ActiveRecord`). Wynikiem jest nadzbiór metody `Time#to_s`. Możesz zignorować dodatki, a metoda zadziała tak jak zawsze.

Tak jak jest w przypadku poszerzania czystej metody (np. `String#pluralize`), tak i z formą nadpisywania z wykorzystaniem nadzbioru metod podstawowych reprezentowanych przez przedstawione przykłady wiąże się pewne ryzyko, a dokładniej rzecz biorąc, ryzyko kolizji. Czy prawdopodobne jest to, że ostatecznie zostaną załadowane dwie biblioteki, które do metody `Time#to_s` dodadzą opcjonalny argument `:db`? Nie, jest to mało prawdopodobne, ale możliwe. I tym razem biblioteka, taka jak Active Support, jest chroniona przez jej powszechność: jeśli ją ładujesz, prawdopodobnie jesteś zaznajomiony z jej możliwościami, a ponadto będziesz wiedzieć, że nie należy przesłaniać nadpisań. Niemniej jednak w pewnym stopniu możliwe jest, że inna ładowana biblioteka wywoła kolizję z biblioteką Active Support. Jak zawsze trudne lub niemożliwe jest całkowite wyeliminowanie ryzyka kolizji. Musisz sam się zabezpieczyć, zaznajamiając się z możliwościami każdej biblioteki, a także odpowiednio testując własny kod.

Ostatnim zasadniczym sposobem przesłaniania zachowania podstawowych elementów języka Ruby, jakiemu się przyjrzymy, jest dodawanie za pomocą metody `Object#extend` funkcjonalności dla konkretnych obiektów.

### *13.2.4. Zmiany poszczególnych obiektów za pomocą metody extend*

W odniesieniu do przepływu tematów metodę `Object#extend` można porównać do powrotu do jego początku. „Wybraliśmy się do zewnętrznych granic" modyfikowania klas podstawowych. Metoda `extend` prowadzi nas z powrotem do centralnego procesu znajdującego się w „sercu" wszystkich takich zmian, czyli zmiany zachowania konkretnego obiektu. Metoda prowadzi też nas do wcześniejszego tematu poruszonego w tym rozdziale. Mowa o dołączaniu modułu do klasy pojedynczych obiektów danego obiektu. Zasadniczo tym zajmuje się metoda `extend`.

#### ROZSZERZANIE FUNKCJONALNOŚCI OBIEKTU ZA POMOCĄ METODY EXTEND

Przyjrzyjmy się ponownie treści punktu 13.1.3, a zwłaszcza przykładowi z klasą `Person`, w którym do klas pojedynczych obiektów wybranych obiektów `Person` dołączono moduł `Secretive`. W ramach przypomnienia użyto następującej techniki (`ruby` to instancja klasy `Person`):

```
class << ruby
  include Secretive
end
```

Poniżej pokazano, jak przykład z klasą `Person` będzie wyglądać, gdy zamiast jawnego otwierania klasy pojedynczych obiektów obiektu `ruby` zostanie użyta metoda `extend`. Metodę tę zastosujmy także dla obiektu `david` (zamiast definicji metody pojedynczego obiektu z instrukcją `def`):

```
module Secretive
  def name
    "[niedostępne]"
  end
end
class Person
  attr_accessor :name
end
david = Person.new
david.name = "Dawid"
matz = Person.new
matz.name = "Matz"
ruby = Person.new
ruby.name = "Ruby"
david.extend(Secretive)   ←❶
ruby.extend(Secretive)
puts "Jest jedna osoba o imieniu #{matz.name}, " +
     "jedna o imieniu #{david.name}, "           +
     "a także jedna o imieniu #{ruby.name}."
```

Większość tego programu jest taka sama jak jego pierwsza wersja. To samo dotyczy danych wyjściowych. Kluczową różnicą jest użycie metody `extend` ❶, która powoduje

dodanie modułu `Secretive` do ścieżek wyszukiwania poszczególnych obiektów `david` i `ruby` przez dołączenie go do ich odpowiednich klas pojedynczych obiektów. Taki proces dołączania ma również miejsce przy rozszerzaniu obiektu klasy.

### DODAWANIE METOD KLASY PRZY UŻYCIU METODY EXTEND

Jeśli w obiekcie klasy utworzysz metodę pojedynczego obiektu o następującej postaci:

```
class Car
  def self.makes
    %w{ Honda Ford Toyota Chevrolet Volvo }
  end
end
```

lub:

```
class Car
  class << self
    def makes
      %w{ Honda Ford Toyota Chevrolet Volvo }
    end
  end
end
```

bądź też z zastosowaniem dowolnego z innych dostępnych wariantów notacji, spowoduje to dodanie metody instancji do klasy pojedynczych obiektów obiektu klasy. Oznacza to, że pomijając inne sposoby, możesz to osiągnąć za pomocą metody `extend`:

```
module Makers
  def makes
    %w{ Honda Ford Toyota Chevrolet Volvo }
  end
end
class Car
  extend Makers
end
```

Jeśli w danej sytuacji jest to właściwsze, możesz rozszerzyć już istniejący obiekt klasy:

```
Car.extend(Makers)
```

W każdym razie efektem jest uzyskanie przez obiekt klasy `Car` dostępu do metody `makes`.

Tak jak w przypadku obiektów innych niż obiekty klasy rozszerzanie obiektu klasy z modułem oznacza dołączanie go do klasy pojedynczych obiektów klasy. Możesz to sprawdzić za pomocą metody `ancestors`:

```
p Car.singleton_class.ancestors
```

Oto dane wyjściowe dla tego wiersza kodu:

```
[#<Class:Car>, Makers, #<Class:Object>, #<Class:BasicObject>, Class, Module,
Object, Kernel, BasicObject]
```

Dziwnie wyglądające pozycje listy to klasy pojedynczych obiektów. Uwzględniana jest klasa pojedynczych obiektów samej klasy `Car`. To samo dotyczy klasy pojedynczych obiektów klasy `Object` (superklasa klasy pojedynczych obiektów klasy `Car`) oraz klasy

pojedynczych obiektów klasy `BasicObject` (superklasa klasy pojedynczych obiektów klasy `Object`). W odniesieniu do naszych potrzeb najważniejsze jest to, że do listy dołączono moduł `Makers`.

Pamiętaj też o tym, że podklasy mają dostęp do metod klasy swojej superklasy. Jeśli dla klasy `Car` utworzysz podklasę i przyjrzysz się klasom nadrzędnym klasy pojedynczych obiektów nowej klasy, na liście zobaczysz moduł `Makers`.

Pierwotnym celem omawiania metody `extend` było przybliżenie sposobu poszerzania podstawowej funkcjonalności języka Ruby. Zajmijmy się teraz tym celem.

### MODYFIKOWANIE ZACHOWANIA PODSTAWOWYCH FUNKCJONALNOŚCI ZA POMOCĄ METODY EXTEND

Prawdopodobnie na tym etapie zebrałeś wszystko w całość. Moduły umożliwiają definiowanie autonomicznych kolekcji metod wielokrotnego użytku. Metoda `Kernel#extend` pozwala zapewnić poszczególnym obiektom dostęp do modułów, co zawdzięcza klasie pojedynczych obiektów i mechanizmowi dołączania. Po zebraniu tego wszystkiego dysponujesz zwięzłym i bezpiecznym sposobem dodawania funkcji do podstawowych obiektów.

Przyjrzyjmy się ponownie zagadce związanej z metodą `String#gsub!`, która polega na tym, że zwraca ona obiekt `nil`, gdy łańcuch nie zostanie zmieniony. Definiując moduł i używając metody `extend`, można zmienić w ograniczonym stopniu zachowania metody `gsub!`, dzięki czemu możesz dokonać tylko niezbędnych zmian i żadnych innych ponadto. Oto odpowiedni kod:

```
module GsubBangModifier
  def gsub!(*args, &block)
    super || self                  ←— ❶
  end
end
str = "Witaj!"
str.extend(GsubBangModifier)       ←— ❷
str.gsub!(/zzz/,"abc").reverse!    ←— ❸
puts str                           ←— Dane wyjściowe: !jatiW
```

W module `GsubBangModifier` definiujemy metodę `gsub!`. Zamiast stosowania techniki opartej na aliasie i wywołaniu wywołujemy metodę `super`, zwracając wartość przekazywaną przez to wywołanie lub obiekt `self` (w przypadku zwrócenia obiektu `nil` przez wywołanie metody `super`) ❶. Jak wcześniej wspomniano, metoda `super` powoduje wykonanie następnej wersji bieżącej metody kolejnej w ścieżce wyszukiwania metod. Trzeba o tym pamiętać.

Tworzymy następnie łańcuch `str` i rozszerzamy go za pomocą modułu `GsubBangModi↪fier` ❷. Wywołanie `str.gsub!` ❸ powoduje wykonanie metody `gsub!` w module `GsubBangModifier`, ponieważ łańcuch `str` napotyka ten moduł w swojej ścieżce wyszukiwania metod przed natrafieniem na klasę `String`, która oczywiście zawiera definicję metody `gsub!`. Wywołanie metody `super` wewnątrz metody `GsubBangModifier#gsub!` powoduje przeskoczenie w górę ścieżki i wykonanie oryginalnej metody `String#gsub!` z przekazaniem jej oryginalnych argumentów i bloku kodu (jeśli go podano). Jest to

efekt wywołania metody `super` bez argumentów i pustej listy argumentów. Wynikiem wywołania tej metody jest sam łańcuch lub obiekt `nil`, zależnie od tego, czy w łańcuchu wprowadzono jakiekolwiek zmiany.

A zatem zachowanie podstawowych obiektów (łańcuchy, tablice, tablice asocjacyjne itd.) możesz zmienić bez ponownego otwierania ich klas, a także bez wprowadzania zmian na poziomie globalnym. Wywołania metody `extend` w kodzie ułatwiają pokazanie tego, co ma miejsce. Ze zmianą metody takiej jak `gsub!` w obrębie samej klasy `String` wiąże się niedogodność tego rodzaju, że nie tylko operacja taka jest globalna, ale też prawdopodobnie jej wynik zostanie ukryty gdzieś w pliku biblioteki, na skutek czego utrudnione będzie wyśledzenie błędów dla osób, którym dała się we znaki zmiana globalna.

Istnieje jeszcze jeden ważny element układanki, który dotyczy sposobu zmiany zachowań podstawowych obiektów: nowa funkcja określana mianem *doprecyzowań*.

### 13.2.5. Użycie doprecyzowań do zmiany zachowania podstawowych obiektów

Doprecyzowania zostały dodane do języka Ruby 2.0, ale do momentu pojawienia się wersji 2.1 były uważane za „eksperymentalne". Ideą doprecyzowania jest wprowadzenie w klasie (nie musi jednak być klasą podstawową) tymczasowej zmiany o ograniczonym zasięgu.

Oto przykład, w którym do klasy `String` wprowadzono metodę `shout`, ale tylko w ograniczonym stopniu:

```
module Shout
  refine String do        ←——❶
    def shout
      self.upcase + "!!!"
    end
  end
end
class Person
  attr_accessor :name
  using Shout             ←——❷
def announce
    puts "Ogłaszanie imienia: #{name.shout}"
  end
end
david = Person.new
david.name = "Dawid"
david.announce            ←—— Dane wyjściowe: Ogłaszanie imienia: DAWID!!!
```

W przykładzie pojawiają się dwie różne metody, które współpracują ze sobą: `refine` ❶ i `using` ❷. Metoda `refine` pobiera nazwę klasy i blok kodu. Wewnątrz bloku kodu definiujesz zachowania, które mają zostać zaadaptowane przez uszczegóławianą klasę. W przykładzie uszczegóławiana jest klasa `String` przez dodanie metody `shout`, która zwraca wersję łańcucha złożoną z dużych liter z następującymi po nich trzema wykrzyknikami.

Metoda `using` „przestawia przełącznik": po „użyciu" modułu, w którym zdefiniowano żądane doprecyzowanie, klasa docelowa adaptuje nowe zachowania. W przykładzie

moduł `Shout` używany jest w obrębie klasy `Person`. Oznacza to, że w czasie obowiązywania tej klasy (od instrukcji `using` do końca definicji klasy) łańcuchy będą „uszczegóławiane" tak, aby uwzględniać metodę `shout`.

Efekt „użycia" doprecyzowania zanika na końcu definicji klasy (lub modułu), w której deklarowane jest zastosowanie doprecyzowania. Właściwie możesz wykorzystać metodę `using` poza definicją klasy lub modułu. W tym przypadku efekt doprecyzowania utrzymuje się do końca pliku, w którym występuje wywołanie metody `using`.

Doprecyzowania mogą ułatwić wprowadzanie tymczasowych zmian w klasach podstawowych w stosunkowo bezpieczny sposób. Doprecyzowania nie będą wpływać na inne pliki programów i biblioteki używane przez program w czasie działania.

Rozdział zakończymy trochę dziwnym tematem, czyli omówieniem klasy `BasicObject`. Nie jest to zagadnienie dotyczące wyłącznie indywidualizacji obiektów (jak już wiesz po przeczytaniu w rozdziale 3. materiału wprowadzającego do klasy `BasicObject`). Klasa ta odnosi się do pochodzenia wszystkich obiektów, w tym tych, których zachowanie różni się od zachowania ich oryginalnych klas. Klasa `BasicObject` może odgrywać ważną rolę w wariancie dynamizmu umożliwianego przez język Ruby.

## 13.3. BasicObject jako przodek i klasa

Klasa `BasicObject` znajduje się na samej górze drzewa klas języka Ruby. W przypadku dowolnego obiektu *`obj`* języka Ruby prawdziwa jest następująca równość:

```
obj.class.ancestors.last == BasicObject
```

Inaczej mówiąc, najwyżej położonym w hierarchii przodkiem każdej klasy jest klasa `BasicObject` (chyba że dołączysz moduł do klasy `BasicObject`, ale jest to „naciągany" scenariusz).

Jak wspomniano w rozdziale 3., instancje klasy `BasicObject` mają niewiele metod, taki „zestaw na przetrwanie", jak można by to ująć. Metody te mogą być uwzględniane w działaniach dotyczących obiektów. Trudnym zadaniem będzie zażądanie od instancji klasy `BasicObject` informacji o jej możliwościach:

```
>> BasicObject.new.methods.sort
NoMethodError: undefined method `methods' for #<BasicObject:0x007fafa308b0d8>
```

Jednak `BasicObject` to klasa, która zachowuje się jak klasa. Informacje możesz uzyskać bezpośrednio z niej, korzystając ze znajomych metod na poziomie klasy:

```
>> BasicObject.instance_methods(false).sort
=> [:!, :!=, :==, :__id__, :__send__, :equal?, :instance_eval,
:instance_exec]
```

Jakie jest przeznaczenie klasy `BasicObject`?

### 13.3.1. Użycie klasy BasicObject

Klasa `BasicObject` umożliwia tworzenie obiektów nierealizujących żadnych działań. Oznacza to, że możesz „nauczyć" je robienia wszystkiego bez obaw, że będą one kolidować z istniejącymi metodami. Zwykle wiąże się to z intensywnym korzystaniem z metody `method_missing`. Definiując ją dla klasy `BasicObject` lub utworzonej przez

siebie klasy, która z niej dziedziczy, możesz uzyskać obiekty z w pełni kontrolowanym zachowaniem. Ponadto obiekty te dysponują niewielką ilością ustalonych z góry informacji o tym, jak mają się zachowywać, lub nie dysponują nimi wcale.

Najlepiej znanym przykładem użycia obiektu pozbawionego prawie zupełnie metod jest biblioteka `Builder` Jima Weiricha. Jest to narzędzie służące do tworzenia kodu XML, którego danymi wyjściowymi są znaczniki XML odpowiadające komunikatom wysyłanym do obiektu, który rozpoznaje niewiele komunikatów. „Magia", która ma miejsce, jest możliwa dzięki metodzie `method_missing`.

Poniżej zamieszczono prosty przykład zastosowania biblioteki `Builder` (wszystkie warianty jej użycia są nieskomplikowane; taki jest cel biblioteki). W przykładzie wstępnie założono, że zainstalowano pakiet *gem* o nazwie `builder`:

```
require 'builder'
xml = Builder::XmlMarkup.new(:target => STDOUT, :indent => 2) ←❶
xml.instruct!    ←❷
xml.friends do
  xml.friend(:source => "college") do
    xml.name("Jan Nowak")
    xml.address do
      xml.street("Polna 123")
      xml.city("Gdziekolwiek, Polska 00000")
    end
  end
end
```

`xml` to obiekt `Builder::XmlMarkup` ❶. Obiekt zaprogramowano w celu wysyłania swoich danych wyjściowych stałej `STDOUT` i stosowania wcięcia w postaci dwóch spacji. Metoda `instruct!` ❷ nakazuje bibliotece tworzącej kod XML rozpoczęcie go od deklaracji języka XML. Wszystkie nazwy metod instancji klasy `Builder::XmlMarkup` są zakończone wykrzyknikiem. Metody te nie mają swoich odpowiedników z nazwą pozbawioną wykrzyknika (w większości sytuacji takie odpowiedniki powinny być dostępne). Jednak w tym przypadku wykrzyknik służy do odróżnienia tych metod od metod o podobnych nazwach, których możesz użyć do generowania znaczników XML za pomocą metody `method_missing`. Przyjęto, że możesz wymagać elementu XML o nazwie `instruct`, ale nie będziesz potrzebować elementu `instruct!`. Oznacza to, że wykrzyknik służy do celów związanych ściśle z domeną, a ponadto sensowne jest w tym przypadku odejście od tradycyjnej konwencji języka Ruby.

Oto dane wyjściowe przykładowego skryptu biblioteki `Builder`:

```
<?xml version="1.0" encoding="UTF-8"?>
<friends>
  <friend source="college">
    <name>Jan Nowak</name>
    <address>
      <street>Polna 123</street>
      <city>Gdziekolwiek, Polska 00000</city>
    </address>
  </friend>
</friends>
```

Różne znaczniki XML pobierają swoje nazwy z wywołań metody. Każda brakująca metoda powoduje utworzenie znacznika, a bloki kodu reprezentują zagnieżdżanie kodu XML. Jeśli brakującej metodzie przekażesz argument w postaci łańcucha, zostanie on użyty jako kontekst tekstowy elementu. Atrybuty są zapewniane w argumentach tablicy asocjacyjnej.

Biblioteka `Builder` używa klasy `BasicObject` do realizowania własnych działań. Co interesujące, biblioteka ta istniała przed klasą `BasicObject`. W oryginalnych wersjach biblioteki używano dostosowanej klasy o nazwie `BlankSlate`, która prawdopodobnie stanowiła inspirację dla twórców klasy `BasicObject`.

W jaki sposób zaimplementujesz prostą klasę opartą na klasie `BasicObject`?

### 13.3.2. Implementowanie podklasy klasy BasicObject

Słowo *prostą* w zadanym powyżej pytaniu oznacza coś prostszego niż klasa `Builder::XmlMarkup` (upraszcza tworzenie kodu XML, ale sama w sobie jest dość złożona). Utwórzmy niewielką bibliotekę opartą na podobnej zasadzie i zwracającą zamierzoną listę elementów. Unikniemy konieczności zapewniania znaczników domykających, co wszystko znacznie uprości.

Klasa `Lister` przedstawiona w poniższym listingu 13.2 będzie dziedziczyć z klasy `BasicObject`. Klasa `Lister` zdefiniuje metodę `method_missing` w taki sposób, że każda brakująca metoda będzie traktowana jako nagłówek generowanej listy. Stosowaniem wcięć będą zarządzać zagnieżdżone bloki kodu.

**Listing 13.2. Klasa Lister: generuje listy z wcięciem na podstawie podklasy klasy BasicObject**

```
class Lister < BasicObject
  attr_reader :list
  def initialize          ←❶
    @list = ""
    @level = 0
  end
  def indent(string)      ←❷
    " " * @level + string.to_s
  end
  def method_missing(m, &block)  ←❸
    @list << indent(m) + ":"     ←❹
    @list << "\n"
    @level += 2           ←❺
    @list << indent(yield(self)) if block   ←❻
    @level -= 2
    @list << "\n"
    return ""             ←❼
  end
end
```

Podczas inicjowania ustawiane są dwie zmienne instancji ❶: `@list` (będzie akumulować łańcuchy dla całej listy) i `@level` (będzie zarządzać stosowaniem wcięć). Metoda `indent` ❷ pobiera łańcuch (lub cokolwiek, co może zostać przekształcone w łańcuch; metoda wywołuje metodę `to_s` w swoim argumencie) i zwraca go z wcięciem po prawej stronie o liczbę spacji określoną przez zmienną `@level`.

Większość działań odbywa się w metodzie `method_missing` ❸. Symbol `m` reprezentuje nazwę brakującej metody (przypuszczalnie odpowiadającą nagłówkowi lub elementowi listy). W związku z tym pierwszym krokiem jest dodanie symbolu `m` (z zastosowanym wcięciem, po którym następuje przecinek) do zmiennej `@list` wraz ze znakiem nowego wiersza ❹. W dalszej kolejności zwiększamy poziom wcięcia ❺ i stosujemy metodę `yield` ❻ (krok ten ma miejsce tylko wtedy, gdy warunek `block` nie ma wartości `nil`; zwykle możesz wykonać test sprawdzający obecność bloku za pomocą metody `block_given?`, ale instancje klasy `BasicObject` nie oferują tej metody!). Użycie metody `yield` może spowodować więcej wywołań brakujących metod. W takiej sytuacji są one przetwarzane, a ich wyniki dodawane do zmiennej `@list` na nowym poziomie wcięcia. Po uzyskaniu wyników operacji wykonywanej przez metodę `yield` zmniejszamy poziom wcięcia i do zmiennej `@list` dodajemy kolejny znak nowego wiersza.

Na końcu metoda `method_missing` zwraca pusty łańcuch ❼. W tym przypadku celem jest uniknięcie złączania zmiennej `@list` z samą sobą. Jeśli ostatecznie metoda `method_missing` określiłaby dla wyrażenia wartość w postaci zmiennej `@list` (np. `@list << "\n"`), zagnieżdżone wywołania metody `method_missing` wewnątrz instrukcji z metodą `yield` zwróciłyby zmienną `@list` i dołączyły ją do samych siebie. Pusty łańcuch przerywa ten cykl.

Oto przykład użycia klasy `Lister`:

```
lister = Lister.new
lister.groceries do |item|
  item.name { "Jabłka" }
  item.quantity { 10 }
  item.name { "Cukier" }
  item.quantity { "1 kg" }
end
lister.freeze do |f|
  f.name { "Lody" }
end
lister.inspect do |i|
  i.item { "samochód" }
end
lister.sleep do |s|
  s.hours { 8 }
end
lister.print do |document|
  document.book { "Rozdział 13" }
  document.letter { "do wydawcy" }
end
puts lister.list
```

Dane wyjściowe uzyskane po uruchomieniu tego kodu są następujące:

```
groceries:
  name:
    Jabłka
  quantity:
    10
  name:
    Cukier
quantity:
    1 kg
```

```
freeze:
  name:
    Lody
inspect:
  item:
    samochód
sleep:
  hours:
    8
print:
  book:
    Rozdział 13
  letter:
    do wydawcy
```

Wprawdzie nie jest to tak przyjemne jak w przypadku biblioteki `Builder`, ale możesz prześledzić operacje przekazania danych i wywołania brakujących metod, a następnie sprawdzić, jaką korzyść daje instancja klasy `BasicObject`. Jeśli przyjrzysz się nazwom metod używanym w przykładowym kodzie, zauważysz, że część z nich to metody wbudowane obiektów (innych niż podstawowe). Jeśli nie odbywa się dziedziczenie z klasy `BasicObject`, zostanie wygenerowany błąd przy próbie wywołania metody `freeze` lub `inspect`. Godne uwagi jest również to, że metody `sleep` i `print`, które są metodami prywatnymi modułu `Kernel`, a tym samym nie mogą być normalnie wywoływane z jawnym odbiorcą, aktywują metodę `method_missing` nawet pomimo tego, że nie są brakującymi metodami, lecz metodami prywatnymi.

Prezentacja klasy `BasicObject` doprowadziła nas do końca omówienia indywidualizacji obiektów. W dalszej kolejności zajmiemy się odmiennym zagadnieniem, które także jest mocno powiązane z dynamiką języka Ruby. Mowa o obiektach możliwych do wywołania i uruchomienia.

## 13.4. Podsumowanie

W rozdziale omówiono następujące zagadnienia:

- Klasy pojedynczych obiektów oraz sposób dodawania do nich metod i stałych.
- Metody klasy.
- Metoda `extend`.
- Kilka technik zmiany zachowania podstawowych elementów języka Ruby.
- Klasa `BasicObject` oraz sposób jej wykorzystania.

Przyjrzeliśmy się sposobom zapewniania zgodności z filozofią języka Ruby przez jego obiekty. Głosi ona, że to, co ma miejsce w czasie wykonywania programu, jest całkowicie powiązane z poszczególnymi obiektami i ich możliwościami dostępnymi w dowolnym danym miejscu. Obiekty języka Ruby tworzone są w konkretnej klasie, ale ich możliwości przechowywania poszczególnych metod w dedykowanej klasie pojedynczych obiektów oznaczają, że dowolny obiekt może realizować niemal każde działanie.

Pokazano, jak otwierać definicje klas pojedynczych obiektów i modyfikować zawartość poszczególnych obiektów, w tym obiekty klasy, które intensywnie korzystają z technik metod pojedynczego obiektu w połączeniu z metodami klasy (które w zasadzie

są metodami pojedynczego obiektu w obiektach klasy). Zaprezentowano też część funkcji, a także zagrożenia związane z zapewnianiem przez język Ruby możliwości wprowadzania zmian nie tylko we własnych klasach, ale też w podstawowych klasach języka. Coś takiego powinno być robione sporadycznie, jeśli w ogóle należy to robić. Ponadto trzeba być świadomym, że może to być także realizowane przez inne osoby, aby mieć możliwość oceny zagrożeń związanych z używaniem dowolnego kodu zewnętrznego, który zmienia zachowania podstawowych elementów języka.

Na końcu rozdziału omówiono klasę `BasicObject`, czyli ostatecznego przodka wszystkich klas. Klasa ta może być stosowana w sytuacjach, w których nawet zwykły obiekt języka Ruby nie jest wystarczająco zwykły.

W następnym rozdziale zajmiemy się obiektami możliwymi do wywołania i uruchomienia, czyli funkcjami (obiekty `Proc`), wątkami, blokami `eval` itp. To, że możesz tworzyć obiekty obejmujące uruchamialny kod, a ponadto masz możliwość modyfikowania tych obiektów, podobnie jak możesz modyfikować dowolny obiekt, zapewnia jeszcze jedną podstawową warstwę całościowego zagadnienia dynamiki języka Ruby.

# 14 Obiekty umożliwiające wywoływanie i uruchamianie

**W tym rozdziale:**

- Obiekty `Proc` jako funkcje anonimowe.
- Metoda `lambda` generująca funkcje.
- Bloki kodu.
- Metoda `Symbol#to_proc`.
- Obiekty metody.
- Powiązania.
- Rodzina metod `eval`.
- Wątki.
- Wykonywanie programów zewnętrznych.

Oprócz podstawowych wywołań metod, które odpowiadają za większość tego, co się dzieje w programie, język Ruby zapewnia rozbudowany zestaw narzędzi służących do realizowania działań na różne sposoby. Czy wymagasz, aby dwie lub większa liczba części kodu były uruchamiane równolegle? Utwórz kilka obiektów `Thread` i uruchamiaj je stosownie do potrzeb. Czy chcesz dokonać wyboru spośród zestawu funkcji możliwych do wykonania, a nie dysponujesz wcześniej ilością informacji wystarczającą do utworzenia dla nich metod? Utwórz tablicę obiektów `Proc` (funkcji anonimowych) i wywołaj ten obiekt, który jest potrzebny. Możesz nawet wyizolować metody jako obiekty lub wykonywać dynamicznie utworzone łańcuchy jako kod.

W tym rozdziale jest mowa o obiektach, które możesz wywoływać, wykonywać lub uruchamiać; są to: wątki, funkcje anonimowe, łańcuchy, a nawet metody przekształcone w obiekty. Przyjrzymy się wszystkim tym konstrukcjom, a także dodatkowym narzędziom (słowa kluczowe, powiązania zmiennych i bloki kodu), które umożliwiają dołączenie w kodzie Ruby obiektów możliwych do wywołania i uruchomienia.

Ostrzeżenie: w nowszych wersjach języka Ruby obiekty możliwe do uruchomienia były na pierwszym planie trudnych i zmieniających się tematów. Trzeba pogodzić się z tym, że wiele sporów towarzyszy temu, jak takie obiekty powinny działać, a ponadto kwestia ta cechuje się dużą złożonością. Obiekty umożliwiające wywoływanie i uruchamianie różnią się między sobą zarówno pod względem składni, jak i przeznaczenia. Wspólne omawianie ich w jednym rozdziale w jakimś stopniu jest wygodne. Jest to też pouczający sposób spojrzenia na te obiekty.

## 14.1. Podstawowe funkcje anonimowe: klasa Proc

W swojej najprostszej formie pojęcie *obiektu możliwego do wywołania* jest reprezentowane w języku Ruby za pośrednictwem obiektów, do których wysyłany jest komunikat `call` z oczekiwaniem, że zostanie wykonany jakiś kod powiązany z tymi obiektami. Główne obiekty możliwe do wywołania w języku Ruby to obiekty `Proc`, lambdy i obiekty metody. Obiekty `Proc` to odrębne sekwencje kodu, które możesz tworzyć, przechowywać i przekazywać jako argumenty metody, a w razie potrzeby wykonywać za pomocą metody `call`. Lambdy przypominają obiekty `Proc`. Prawdę powiedziawszy, lambda *to* obiekt `Proc`, lecz z trochę bardziej szczególną konstrukcją wewnętrzną. Różnice będą się uwidaczniać podczas omawiania kolejno każdego typu obiektu. Obiekty metody reprezentują metody wyodrębnione do obiektów, które również mogą być przechowywane, przekazywane i wykonywane.

Omówienie obiektów możliwych do wywołania zaczniemy od obiektów `Proc`.

**UWAGA** Termin *lambda* będzie wskazywać instancję obiektu `Proces` w stylu lambdy (nie martw się; wkrótce dowiesz się, co to oznacza). Termin *funkcja* to ogólny termin odnoszący się do odrębnych jednostek kodu pobierających dane wejściowe i zwracających wartość. W języku Ruby nie istnieje klasa `Function`. W rozdziale pojawi się jednak czasami termin *funkcja* używany w celu odwołania się do obiektów proc i lambda. Jest to po prostu kolejny, trochę bardziej abstrakcyjny sposób identyfikowania tych obiektów.

### 14.1.1. Obiekty Proc

Dokładne zrozumienie obiektów `Proc` oznacza zaznajomienie się z kilkoma rzeczami: podstawami tworzenia i używania obiektów `Proc`, obsługą przez te obiekty argumentów i powiązań zmiennych, pełnieniem przez obiekty `Proc` roli *domknięć*, relacją między obiektami `Proc` i blokami kodu oraz różnicą między tworzeniem obiektów `Proc` za pomocą metody `Proc.new`, metody `proc`, metody `lambda` i konstruktora lambda literału `->`. Choć podano tutaj tego mnóstwo, wszystko będzie do siebie pasować, jeśli związane z tym informacje będziesz poznawać stopniowo.

Zacznijmy od podstawowego obiektu możliwego do wywołania, czyli instancji obiektu `Proc` utworzonej za pomocą metody `Proc.new`. Obiekt ten jest definiowany przez utworzenie instancji klasy `Proc` z uwzględnieniem bloku kodu:

```
pr = Proc.new { puts "Wewnątrz bloku obiektu Proc" }
```

Blok kodu staje się treścią obiektu `Proc`, po którego wywołaniu wykonywany jest podany blok. A zatem jeśli wywołasz obiekt `pr`:

```
pr.call
```

zostanie wyświetlony następujący wiersz:

```
Wewnątrz bloku obiektu Proc
```

Jest to podstawowy scenariusz: blok kodu udostępniony wywołaniu metody `Proc.new` staje się treścią obiektu `Proc` i jest wykonywany w momencie wywołania tego obiektu za pomocą metody `call`. Wszystko inne, co ma miejsce lub może się wydarzyć, obejmuje rozszerzenia tego schematu lub jego warianty.

Pamiętaj, że `Proc` to obiekt. Oznacza to, że możesz go przypisywać do zmiennych, umieszczać wewnątrz tablic, wysyłać jako argumenty metody i generalnie traktować jak dowolny inny obiekt. Obiekty `Proc` mają informacje o porcji kodu (blok kodu, za pomocą którego zostały utworzone) i możliwość wykonania go, gdy zostanie to zażądane. Nadal jednak pozostają one obiektami.

**Metoda proc**

Metoda `proc` pobiera blok i zwraca obiekt `Proc`. Oznacza to, że możesz użyć wiersza `proc { puts "Witaj!" }` zamiast wiersza `Proc.new { puts "Witaj!" }` i uzyskać ten sam wynik. Wcześniej metody `Proc.new` i `proc` różniły się trochę od siebie, a przy tym metoda `proc` pełniła rolę synonimu metody `lambda` (więcej informacji zawarto w podrozdziale 14.2), natomiast metoda `proc`/`lambda` tworzyła specjalizowane obiekty `Proc`, które nie były do końca takie same jak to, co zostało wygenerowane przez metodę `Proc.new`. Zgadza się, powodowało to niejasności. Jednak obecnie, choć nadal istnieją dwa warianty obiektu `Proc`, metody `Proc.new` i `proc` realizują to samo, natomiast metoda `lambda` zapewnia inny wariant. Przynajmniej nazewnictwo jest bardziej przewidywalne.

Być może najważniejszym aspektem obiektów `Proc`, który pozwala je zrozumieć, jest relacja między nimi i blokami kodu. Relacja ta jest bliska i okazuje się być ważnym kluczem do dalszego zrozumienia zagadnienia.

### 14.1.2. Obiekty Proc i bloki oraz różnice między nimi

Podczas tworzenia obiektu `Proc` zawsze podajesz blok kodu. Jednak nie każdy blok kodu może stanowić bazę dla obiektu `Proc`. W następującym fragmencie kodu:

```
[1,2,3].each {|x| puts x * 10 }
```

znajduje się blok kodu, ale nie jest tworzony obiekt `Proc`. Cała związana z tym intryga jest jeszcze bardziej zagmatwana. Metoda może przechwytywać blok przekształcany w obiekt `Proc`, używając specjalnej składni parametrów, którą pobieżnie zaprezentowano w rozdziale 9.:

```
def call_a_proc(&block)
  block.call
end
call_a_proc { puts "Jestem blokiem..., obiektem Proc...lub czymś." }
```

Dane wyjściowe nie są zaskoczeniem:

```
Jestem blokiem..., obiektem Proc...lub czymś.
```

Możliwe jest też, aby obiekt `Proc` zastąpił blok kodu w wywołaniu metody, co umożliwia podobna specjalna składnia:

```
p = Proc.new {|x| puts x.upcase }
%w{ Dawid Nowak }.each(&p)
```

Oto dane wyjściowe wywołania metody `each`:

```
DAWID
NOWAK
```

Pozostaje jednak następująca kwestia: co dokładnie ma miejsce w przypadku obiektów `Proc` i bloków? Dlaczego i w jaki sposób obecność `(&p)` przekonuje metodę `each`, że nie potrzebuje prawdziwego bloku kodu?

W dużym stopniu relacja między blokami i obiektami `Proc` sprowadza się do porównania składni z obiektami.

### SKŁADNIA (BLOKI) I OBIEKTY (PROC)

Ważnym i często źle rozumianym faktem jest to, że blok kodu języka Ruby nie jest obiektem. W następującym znajomym i trywialnym przykładzie występuje odbiorca, operator kropki, nazwa metody i blok kodu:

```
[1,2,3].each {|x| puts x * 10 }
```

Odbiorca jest obiektem, ale blok kodu już nie. Zamiast tego blok kodu jest częścią składni wywołania metody.

W celu umiejscowienia bloków kodu w kontekście możesz posłużyć się analogią z listami argumentów.

W wywołaniu metody z argumentami:

```
puts c2f(100)
```

argumenty są obiektami, ale sama lista argumentów, czyli `(100)` w całości, nie jest obiektem. Nie istnieje klasa `ArgumentList` ani klasa `CodeBlock`.

Wszystko jednak komplikuje się bardziej w przypadku składni bloku niż w przypadku list argumentów z powodu sposobu współpracy bloków i obiektów `Proc`. Instancja klasy `Proc` jest obiektem. Blok kodu zawiera wszystko, co jest niezbędne do utworzenia obiektu `Proc`. Z tego właśnie powodu metoda `Proc.new` pobiera blok kodu: w ten oto sposób metoda stwierdza, co obiekt `Proc` ma zrealizować, gdy zostanie wywołany.

Ważną konsekwencją faktu, że blok kodu jest konstrukcją składniową, a nie obiektem, jest to, że bloki kodu nie są argumentami metody. Kwestia zapewniania argumentów metodzie jest niezależna od tego, czy blok kodu istnieje, tak jak obecność bloku jest niezależna od obecności lub nieobecności listy argumentów. Zapewniając blok kodu, nie wysyłasz go metodzie jako argumentu. Zapewnianie bloku kodu jest

czymś odrębnym. Przyjrzyjmy się bliżej mechanizmom konwersji, które umożliwiają przechwytywanie bloków kodu jako obiektów `Proc`, a także zastosowanie ich zamiast bloków kodu.

### 14.1.3. Konwersje dotyczące bloków i obiektów Proc

Konwersja między blokami i obiektami `Proc` jest prostą operacją. Nie jest to zbyt zaskakujące, ponieważ celem bloku kodu jest jego wykonanie, a obiekt `Proc` to obiekt, którego zadaniem jest zapewnienie dostępu z możliwością wykonania do wcześniej zdefiniowanego bloku kodu. Przyjrzymy się najpierw konwersjom między blokami i obiektami `Proc`, a następnie użyciu tych obiektów w miejsce bloków.

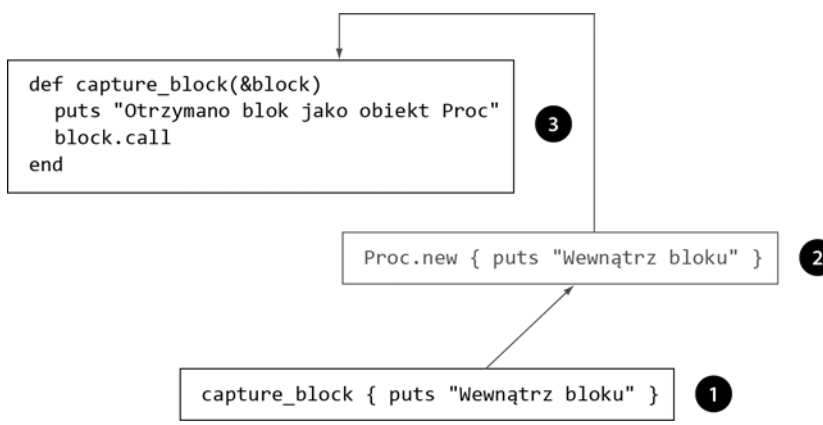
#### PRZECHWYTYWANIE BLOKU KODU JAKO OBIEKTU PROC

Zacznijmy od kolejnej prostej metody, która przechwytuje swój blok kodu jako obiekt `Proc`, a następnie wywołuje go:

```
def capture_block(&block)
  block.call
end
capture_block { puts "Wewnątrz bloku" }
```

To, co ma miejsce, to swego rodzaju niejawne wywołanie metody `Proc.new` przy użyciu tego samego bloku. A zatem tworzony obiekt `Proc` jest wiązany z parametrem `block`.

Na rysunku 14.1 w formie graficznej pokazano, jak blok kodu staje się obiektem `Proc`. Pierwszym zdarzeniem (widocznym na dole rysunku) jest wywołanie metody `capture_block` z blokiem kodu. Po drodze tworzony jest nowy obiekt `Proc` (krok 2.) z wykorzystaniem tego samego bloku. Jest to ten obiekt `Proc`, z którym powiązano zmienną `block` wewnątrz treści metody (krok 3.).




**Rysunek 14.1.** „Fikcyjne" tworzenie instancji klasy Proc ma miejsce między wywołaniem metody i metodą

A zatem element składniowy (blok kodu) stanowi bazę operacji tworzenia obiektu. „Fikcyjny" krok tworzenia obiektu `Proc` z bloku wyjaśnia również potrzebę użycia specjalnej składni opartej na operatorze `&`. Wywołanie metody może obejmować zarówno

listę argumentów, jak i blok kodu. Bez specjalnej flagi, takiej jak `&`, interpreter języka Ruby nie ma możliwości stwierdzenia, że chcesz przerwać wiązanie parametrów ze zwykłymi argumentami i zamiast tego przeprowadzić konwersję bloku do postaci obiektu `Proc` oraz zapisać wyniki.

Operator `&` pojawia się też, gdy chcesz dokonać konwersji w inny sposób, czyli używając obiektu `Proc` zamiast bloku kodu.

### UŻYCIE OBIEKTÓW PROC DLA BLOKÓW

Oto sposób, w jaki możesz wywołać metodę `capture_block`, używając obiektu `Proc`, a nie bloku kodu:

```
p = Proc.new { puts "Ten argument w postaci obiektu Proc będzie pełnić rolę bloku kodu." }
capture_block(&p)
```

Oto dane wyjściowe:

```
Ten argument w postaci obiektu Proc będzie pełnić rolę bloku kodu.
```

W przypadku użycia obiektu `Proc` jako bloku kluczowe jest to, że właściwie obiekt ten jest stosowany zamiast bloku. Obiekt `Proc` wysyłasz wywoływanej metodzie jako argument. Tak jak oznaczasz parametr w definicji metody za pomocą operatora `&` w celu wskazania, że powinien dokonać konwersji bloku do postaci obiektu `Proc`, tak też używasz tego znaku po stronie wywołania metody, aby wskazać, że obiekt `Proc` powinien realizować zadanie bloku kodu.

Pamiętaj o tym, że ponieważ obiekt `Proc` oznaczony za pomocą operatora `&` pełni rolę bloku kodu, nie możesz wysłać bloku w tym samym wywołaniu metody. Jeśli tak postąpisz, zostanie wygenerowany błąd. Wywołanie:

```
capture_block(&p) { puts "To jest jawny blok" }
```

spowoduje zwrócenie błędu informującego o tym, że podano zarówno argument bloku, jak i sam blok. Interpreter języka Ruby nie może zdecydować, jaki element (obiekt `Proc` lub blok) pełni rolę bloku, dlatego możesz użyć tylko jednego.

Pojawia się tutaj interesujący wątek poboczny. Podobnie jak wiele innych operatorów języka Ruby operator `&` w konstrukcji `&p` służy do opakowania metody `to_proc`. Wywołanie tej metody dla obiektu `Proc` powoduje zwrócenie samego obiektu, co trochę przypomina wywołanie metody `to_s` dla łańcucha lub metody `to_i` dla liczby całkowitej.

Zauważ jednak, że w dalszym ciągu niezbędny jest operator `&`. Jeśli użyjesz następującego wiersza:

```
capture_block(p)
```

lub wiersza:

```
capture_block(p.to_proc)
```

obiekt `Proc` będzie pełnił rolę zwykłego argumentu metody. Nie ma miejsca żadne specjalne działanie, które sprawia, że argument w postaci obiektu `Proc` realizuje zadanie bloku kodu.

A zatem operator & w wywołaniu capture_block(&p) wykonuje dwa zadania: powoduje wywołanie metody to_proc obiektu p, a ponadto informuje interpreter języka Ruby o tym, że wynikowy obiekt Proc zastępuje blok kodu. Ponieważ to_proc to metoda, możliwe jest użycie jej w bardziej ogólny sposób.

### UOGÓLNIANIE METODY TO_PROC

W teorii metodę to_proc możesz zdefiniować w dowolnej klasie lub dla każdego obiektu. W takiej sytuacji technika oparta na operatorze & będzie obowiązywać dla objętych tym obiektów. Prawdopodobnie nie będzie konieczne częste korzystanie z tego. Dwie klasy, w których metoda to_proc jest najbardziej przydatna, to Proc (wcześniej omówiona) i Symbol (zostanie omówiona w następnym podrozdziale). Zachowanie tej metody jest już wbudowane w te klasy. Zaznajomienie się jednak ze sposobem stosowania metody to_proc we własnych klasach może umożliwić zorientowanie się w możliwościach dynamicznych tkwiących „pod powierzchnią" języka.

Oto raczej dziwny, lecz pouczający fragment kodu:

```
class Person
  attr_accessor :name          ←❶
  def self.to_proc             ←❷
    Proc.new {|person| person.name }
  end
end
d = Person.new                 ←❸
d.name = "Dawid"
m = Person.new
m.name = "Matz"
puts [d,m].map(&Person)        ←❹
```

Jeśli chcesz prześledzić ten kod, najlepiej zacząć od ostatniego wiersza ❹. W tym wierszu znajduje się tablica złożona z dwóch obiektów Person. Dla tablicy wykonywana jest operacja metody map. Jak wiesz, metoda Array#map pobiera blok kodu. W tym przypadku zamiast bloku używamy obiektu Proc, który na liście argumentów reprezentowany jest przez &Person. Oczywiście Person nie jest obiektem Proc, lecz klasą. Aby rozpoznać, z czym ma do czynienia, interpreter języka Ruby żąda od obiektu Person reprezentowania siebie jako obiektu Proc, co oznacza niejawne wywołanie metody to_proc obiektu Person ❷.

Z kolei metoda ta tworzy prosty obiekt Proc, który pobiera jeden argument i wywołuje dla niego metodę name. Obiekty Person mają atrybuty name ❶. Obiekty Person utworzone w celu testowania kodu z całą pewnością zawierają imiona ❸. Wszystko to oznacza, że odwzorowanie tablicy obiektów Person ([d,m]) zgromadzi atrybuty name obiektów, a cała tablica wynikowa zostanie wyświetlona (dzięki metodzie puts).

Jest to rozbudowana technika, a ponadto trochę nieprecyzyjna. Przecież dowolna metoda pobierająca blok może użyć argumentu &Person, co może okazać się dziwne, jeśli będzie to obejmować obiekty niepowiązane z osobami, które są pozbawione metody name. W przykładzie pokazano jednak, że metoda to_proc może pełnić rolę narzędzia konwersji o dużych możliwościach. Jak się okaże w następnym punkcie, właśnie to ma miejsce w przypadku klasy Symbol.

### *14.1.4. Użycie metody Symbol#to_proc do zapewnienia zwięzłości*

Metoda wbudowana `Symbol#to_proc` pojawia się w sytuacjach podobnych do następującej:

```
%w{ dawid nowak }.map(&:capitalize)
```

Oto wynik:

```
["Dawid", "Nowak"]
```

Symbol `:capitalize` jest interpretowany jako komunikat, który kolejno ma zostać wysłany każdemu elementowi tablicy. A zatem poprzedni wiersz kodu jest równoznaczny następującemu:

```
%w{ dawid nowak }.map {|str| str.capitalize }
```

Jak jednak widać, kod jest bardziej zwięzły.

Jeśli w kodzie napotkałeś konstrukcję `&:capitalize` lub podobną do niej, mogłeś pomyśleć, że jest enigmatyczna. Wiedząc jednak, jak dokonuje ona analizy składni oraz że `:capitalize` to symbol, a operator `&` powoduje wyzwolenie metody `to_proc`, masz możliwość poprawnego interpretowania konstrukcji i docenienia jej wyrazistości.

Sytuacja, w której użyto metody `Symbol#to_proc`, jest odpowiednia do wyeliminowania nawiasów okrągłych:

```
%w{ dawid nowak }.map &:capitalize
```

Usuwając nawiasy okrągłe, możesz sprawić, że symbol w postaci obiektu `Proc` będzie wyglądać na znajdujący się w pozycji bloku kodu. Oczywiście takie działanie nie jest konieczne. Należy mieć świadomość tego, że używając indykatora `to_proc &`, wysyłasz obiekt `Proc` jako argument oznaczony za pomocą operatora `&`, a nie zapewniasz rzeczywistego bloku kodu.

Metoda `Symbol#to_proc` to między innymi znakomity przykład działania, jakie automatycznie realizuje interpreter języka Ruby. W razie potrzeby działanie to z łatwością możesz sam wykonać. Poniżej wyjaśniono, jak to zrobić.

#### IMPLEMENTOWANIE METODY SYMBOL#TO_PROC

Ponownie przeanalizujmy przypadek użycia metody `to_proc`:

```
%w{ dawid nowak }.map(&:capitalize)
```

Wiemy, że powyższy wiersz jest odpowiednikiem następującego:

```
%w{ dawid nowak }.map {|str| str.capitalize }
```

To samo możesz zapisać w następujący sposób:

```
%w{ dawid nowak }.map {|str| str.send(:capitalize) }
```

W normalnej sytuacji kod nie zostałby zapisany w ten sposób, ponieważ nie ma potrzeby kłopotliwego korzystania z metody `send`, jeśli możliwe jest wywołanie metody, która bazuje na zwykłej składni z kropką. Jednak wersja kodu oparta na metodzie `send` wskazuje sposób implementacji metody `Symbol#to_proc`. Zadaniem bloku w tym przykładzie jest wysłanie symbolu `:capitalize` do każdego elementu tablicy. Oznacza to, że obiekt

`Proc` utworzony przez metodę `:capitalize#to_proc` musi wysłać symbol `:capitalize` swojemu argumentowi. Uogólniając to, możemy przeprowadzić następującą prostą (prawie rozczarowującą, jak to można określić) implementację metody `Symbol#to_proc`:

```
class Symbol
  def to_proc
    Proc.new {|obj| obj.send(self) }
  end
end
```

Metoda ta zwraca obiekt `Proc` pobierający jeden argument i wysyła do niego obiekt `self` (będzie to dowolny używany symbol).

Nową implementację możesz wypróbować w narzędziu irb. Niech metoda wyświetla powitanie, aby jasne było, że używana wersja jest tą, którą właśnie zdefiniowano:

```
class Symbol
  def to_proc
    puts "Witaj w nowej wersji metody Symbol#to_proc!"
    Proc.new {|obj| obj.send(self) }
  end
end
```

Zapisz ten kod w pliku o nazwie *sym2proc.rb*, a następnie z poziomu katalogu, w którym go umieszczono, załaduj plik w narzędziu irb, używając flag `-I` (dołącza ścieżkę w ścieżce ładowania) i `-r` (flaga żądania):

```
irb --simple-prompt -I. -r sym2proc
```

Po zastosowaniu techniki opartej na *symbolu* `&:` sprawdzisz, jak działa nowa wersja metody `to_proc`:

```
>> %w{ dawid nowak }.map(&:capitalize)
Witaj w nowej wersji metody Symbol#to_proc!
=> ["Dawid", "Nowak"]
```

Nie jesteś zobligowany do użycia skrótu metody `Symbol#to_proc`, ale warto wiedzieć, jak to działa, aby możliwe było zdecydowanie, kiedy skorzystanie z niego będzie odpowiednie.

Jednym z najważniejszych aspektów obiektów `Proc` jest pełnienie przez nie roli *domknięć*, czyli funkcji anonimowych zachowujących powiązania zmiennych lokalnych, które obowiązują podczas tworzenia obiektów `Proc`. W następnym punkcie przyjrzymy się sposobowi działania obiektów `Proc` w roli domknięć.

### 14.1.5. Obiekty Proc w roli domknięć

Wspomniano już o tym, że zmienne lokalne używane wewnątrz treści metody nie są tymi samymi zmiennymi co zmienne lokalne zastosowane w zasięgu wywołania metody:

```
def talk
    a = "Witaj"
    puts a
  end
  a = "Żegnaj"
  talk      ←——— Dane wyjściowe: Witaj
  puts a    ←——— Dane wyjściowe: Żegnaj
```

Identyfikator a został przypisany dwukrotnie, ale te dwa przypisania (dwie zmienne a) nie są ze sobą powiązane.

Dowiedziałeś się też, że bloki kodu zachowują zmienne, które istniały w momencie ich tworzenia. Wszystkie bloki kodu wykonują następujący kod:

```
m = 10
[1,2,3].each {|x| puts x * m }
```

Takie zachowanie nabiera znaczenia, gdy blok kodu jest zawartością obiektu umożliwiającego wywołanie:

```
def multiply_by(m)
  Proc.new {|x| puts x * m }
end
mult = multiply_by(10)
mult.call(12)          ←——— Dane wyjściowe: 120
```

W tym przykładzie metoda multiply_by zwraca obiekt Proc, który może być wywoływany z dowolnym argumentem, ale zawsze mnożony jest przez liczbę wysłaną jako argument metodzie multiply_by. Zmienna m, niezależnie od jej wartości, jest zachowywana wewnątrz bloku kodu przekazywanego metodzie Proc.new. Oznacza to, że zmienna pełni rolę mnożnika każdorazowo przy wywoływaniu obiektu Proc zwracanego przez metodę multiply_by.

Obiekty Proc powodują, że zasięgi nabierają trochę odmiennego znaczenia. Gdy tworzysz blok kodu dla wywołania metody Proc.new, utworzone zmienne lokalne nadal znajdują się w zasięgu (jak w przypadku dowolnego bloku kodu). Zmienne te pozostają w zasięgu wewnątrz obiektu Proc, niezależnie od miejsca lub momentu jego wywołania.

Przyjrzyj się poniższemu listingowi 14.1 i zwróć uwagę na dwie zmienne o nazwie a.

**Listing 14.1. Obiekt Proc zachowujący kontekst lokalny**

```
def call_some_proc(pr)
  a = "Nieodpowiednia zmienna a w zasięgu metody"          ←——— ❶
  puts a
  pr.call    ←——— ❷
end
a = "Zmienna a, która zostanie użyta w bloku obiektu Proc"  ←——— ❸
pr = Proc.new { puts a }
pr.call
call_some_proc(pr)
```

Tak jak w poprzednim przykładzie w definicji metody ❶ występuje zmienna a, która znajduje się też w zasięgu zewnętrznym (wywołującym) ❸. Wewnątrz metody ma miejsce wywołanie obiektu Proc. Kod tego obiektu, który akurat już znamy, składa się z wiersza puts a. Zauważ, że gdy obiekt Proc jest wywoływany z wnętrza metody ❷, wyświetlana zmienna a nie jest zmienną a zdefiniowaną w metodzie. Jest to zmienna a z zasięgu, w którym oryginalnie utworzono obiekt Proc:

```
Zmienna a, która zostanie użyta w bloku obiektu Proc
Nieodpowiednia zmienna a w zasięgu metody
Zmienna a, która zostanie użyta w bloku obiektu Proc
```

Obiekt `Proc` zachowuje swój kontekst. Częścią tego kontekstu jest zmienna `a`, do której przypisano określony łańcuch. Zmienna ta znajduje się wewnątrz obiektu `Proc`.

Taka porcja kodu zachowująca własny kontekst tworzenia jest nazywana *domknięciem*. Tworzenie domknięcia przypomina pakowanie walizki: zawsze gdy ją otworzysz, będzie zawierać to, co zostało w niej umieszczone podczas pakowania. W momencie otwarcia domknięcia (przez wywołanie go) będzie ono zawierać to, co wstawiono do niego po jego utworzeniu. Domknięcia są ważne, ponieważ zachowują częściowy stan działania programu. Zmienna będąca poza zasięgiem w momencie zwracania wartości przez metodę może później przekazać coś interesującego. Dzięki domknięciu można zachować taką zmienną, aby w dalszym ciągu mogła zapewniać informacje lub wyniki obliczeń.

Klasycznym przykładem domknięcia jest licznik. Poniżej przedstawiono metodę zwracającą domknięcie (obiekt `Proc` z zachowanymi powiązaniami zmiennych lokalnych). Obiekt `Proc` pełni rolę licznika, który inkrementuje swoją zmienną każdorazowo po wywołaniu:

```
def make_counter
  n = 0
  return Proc.new { n += 1 }    ←❶
end
c = make_counter    ←❷
puts c.call
puts c.call
d = make_counter    ←❸
puts d.call
puts c.call         ←❹
```

Oto dane wyjściowe:

```
1
2
1
3
```

Logika obiektu `Proc` uwzględnia dodawanie liczby 1 do `n` ❶. A zatem przy pierwszym wywołaniu obiektu `Proc` określana jest dla niego liczba 1. Za drugim razem jest to liczba 2 itd. Wywołanie metody `make_counter`, a następnie zwracanego przez nią obiektu `Proc` jest tego potwierdzeniem: najpierw wyświetlana jest liczba 1, a następnie liczba 2 ❷. Jednak nowy licznik ponownie zaczyna od liczby 1. Drugie wywołanie metody `make_counter` ❸ generuje nową zmienną lokalną `n`, która jest zachowywana w innym obiekcie `Proc`. Różnica między dwoma licznikami staje się wyraźna przy trzecim wywołaniu pierwszego licznika, który wyświetla liczbę 3 ❹. Licznik ponawia działanie od miejsca, w którym zakończył, używając zmiennej `n` zachowanej wewnątrz licznika podczas jego tworzenia.

Podobnie jak dowolny blok kodu, tak i blok zapewniany w czasie tworzenia obiektu `Proc` może pobierać argumenty. Przyjrzyjmy się dokładnie argumentom i parametrom bloku podczas tworzenia obiektu `Proc`.

### 14.1.6. Argumenty i parametry obiektu Proc

Oto tworzenie instancji obiektu `Proc` za pomocą bloku pobierającego jeden argument:

```
pr = Proc.new {|x| puts "Wywołano z argumentem #{x}" }
pr.call(100)
```

Oto dane wyjściowe:

```
Wywołano z argumentem 100
```

Obiekty `Proc` różnią się od metod pod względem obsługi argumentów tym, że nie ma dla nich znaczenia, czy otrzymują właściwą liczbę argumentów. Następujący jednoargumentowy obiekt `Proc`:

```
>> pr = Proc.new {|x| p x }
=> #<Proc:0x000001029a8960@(irb):1>
```

może być wywoływany z dowolną liczbą argumentów (w tym bez żadnego). Jeśli obiekt zostanie wywołany bez żadnych argumentów, dla jego pojedynczego parametru zostanie ustawiony obiekt `nil`:

```
>> pr.call
nil
```

W przypadku wywołania z więcej niż jednym argumentem pojedynczy parametr jest wiązany z pierwszym argumentem, a pozostałe argumenty są odrzucane:

```
>> pr.call(1,2,3)
1
```

Pamiętaj, że pojedyncza wyświetlana wartość jest wartością zmiennej `x`.

Oczywiście możesz też użyć argumentów pobierających, a także całej reszty elementów z już poznanej listy parametrów. Pamiętaj jednak o tym, że obiekty `Proc` są trochę bardziej wyraziste niż metody w kwestii *liczby argumentów*. Niemniej język Ruby zapewnia sposób tworzenia bardziej wymyślnych funkcji w postaci metody `lambda`.

## 14.2. Tworzenie funkcji za pomocą metody lambda i konstruktora ->

Podobnie jak metoda `Proc.new` metoda lambda zwraca obiekt `Proc`, używając podanego bloku kodu jako treści funkcji:

```
>> lam = lambda { puts "lambda!" }
=> #<Proc:0x0000010299a1d0@(irb):2 (lambda)>
>> lam.call
lambda!
```

Jak sugeruje łańcuch inspekcji, obiekt zwracany przez metodę `lambda` jest klasy `Proc`. Zwróć jednak uwagę na notację `(lambda)`. Nie ma klasy `Lambda`, ale występuje odmienny wariant klasy `Proc` w stylu lambdy. Tego rodzaju obiekty `Proc` różnią się na trzy sposoby od swoich zwykłych „kuzynów”.

Po pierwsze, lambda wymaga jawnego tworzenia. Każdorazowo, gdy w języku Ruby niejawnie tworzone są obiekty `Proc`, są one zwykłymi tego typu obiektami, a nie lambdami. Oznacza to przede wszystkim, że po pobraniu bloku kodu w następującej metodzie:

```
def m(&block)
```

Pobrany obiekt `Proc` jest zwykłym tego typu obiektem, a nie lambdą.

Po drugie, lambdy różnią się od zwykłych obiektów `Proc` sposobem traktowania słowa kluczowego `return`. Umieszczone w obrębie lambdy powoduje ono wyjście poza treść lambdy do kontekstu kodu bezpośrednio zawierającego lambdę. Słowo kluczowe `return` wewnątrz obiektu `Proc` powoduje powrót z metody, w której wykonywany jest ten obiekt. Oto przykład ilustrujący różnicę:

```
def return_test
  l = lambda { return }
  l.call                                  ←❶
  puts "Nadal tutaj!"
  p = Proc.new { return }
  p.call                                  ←❷
  puts "Nie ujrzysz tego komunikatu!"     ←❸
end
return_test
```

Dane wyjściowe tego fragmentu kodu to komunikat `Nadal tutaj!`. Nigdy nie ujrzysz drugiego wyświetlanego komunikatu ❸, ponieważ wywołanie obiektu `Proc` ❷ powoduje powrót z metody `return_test`. Jednak wywołanie lambdy ❶ powoduje powrót (wyjście) z treści lambdy, a wykonywanie metody jest kontynuowane od miejsca, w którym zostało zakończone.

> **OSTRZEŻENIE** Ponieważ słowo kluczowe `return` w obrębie obiektu `Proc` (w zwykłym wariancie) powoduje powrót z metody obejmującej, wywołanie obiektu `Proc` zawierającego to słowo kluczowe poza obrębem dowolnej metody generuje błąd krytyczny. Aby zobaczyć demonstrację tego błędu z poziomu wiersza poleceń, spróbuj wykonać następujące polecenie: `ruby -e 'Proc.new { return }.call'`.

Ostatnią i najważniejszą rzeczą jest to, że obiekty `Proc` w stylu lambdy nie zezwalają na wywoływanie z niewłaściwą liczbą argumentów. Obiekty te są „wybredne”:

```
>> lam = lambda {|x| p x }
=> #<Proc:0x000001029901f8@(irb):3 (lambda)>
>> lam.call(1)
1
=> 1
>> lam.call
ArgumentError: wrong number of arguments (0 for 1)
>> lam.call(1,2,3)
ArgumentError: wrong number of arguments (3 for 1)
```

Oprócz metody `lambda` istnieje konstruktor literału lambda.

### KONSTRUKTOR LAMBDA ->

Konstruktor lambda działa w następujący sposób:

```
>> lam = -> { puts "witaj" }
=> #<Proc:0x0000010289f140@(irb):1 (lambda)>
>> lam.call
witaj
```

Aby lambda pobrała argumenty, parametry musisz umieścić w nawiasach okrągłych po konstruktorze ->, a *nie* w obrębie znaków potoku w bloku kodu:

```
>> mult = ->(x,y) { x * y }
=> #<Proc:0x00000101023c38@(irb):7 (lambda)>
>> mult.call(3,4)
=> 12
```

Trochę historii: konstruktor lambda istnieje przede wszystkim dlatego, że w starszych wersjach języka Ruby występował problem z analizowaniem składni argumentów w stylu metody w obrębie znaków potoku. Na przykład w wersji 1.8 języka Ruby nie było możliwe użycie w następujący sposób domyślnej składni argumentów:

```
lambda {|a,b=1| "Nie działa w wersji 1.8 języka Ruby -- błąd składni!" }
```

Problem polegał na tym, że interpreter języka Ruby nie był w stanie stwierdzić, czy drugi znak potoku był drugim separatorem, czy operatorem bitowym `OR`. Konstruktor lambda wprowadzono w celu umożliwienia użycia w przypadku lambdy pełnej postaci argumentów w stylu metod:

```
->(a, b=1) { "Działa w wersji 1.8 języka Ruby!" }
```

Ostatecznie poradzono sobie z ograniczeniem analizatora składni. Obecnie możesz w pełni korzystać ze składni argumentów metody między znakami potoku w bloku kodu. Mówiąc wprost, oznacza to, że konstruktor lambda nie jest dłużej potrzebny. Wzbudził jednak pewne zainteresowanie. Przekonasz się, że konstruktor ten był dość często używany.

W praktyce w języku Ruby najczęściej nie są wywoływane obiekty `Proc` lub lambdy, lecz metody. Do tej pory wywoływanie metod było traktowane jak coś realizowane na jednym poziomie przetwarzania: komunikaty wysyłamy do obiektów, które wykonują metodę o odpowiedniej nazwie. Możliwa jest jednak obsługa metod jako obiektów, o czym będzie mowa w następnym podrozdziale.

## 14.3. Metody jako obiekty

Metody nie prezentują się w postaci obiektów do momentu zażądania tego od nich. Traktowanie metod jak obiektów uwzględnia proces *prezentowania ich w postaci obiektów*.

### 14.3.1. Przechwytywanie obiektów Method

Aby uzyskać obiekt `Method`, użyj metody `method` z nazwą metody jako argumentem (w postaci łańcucha lub symbolu):

```
class C
  def talk
    puts "Test uzyskiwania obiektu Method! Obiekt self to #{self}."
  end
end
c = C.new
meth = c.method(:talk)
```

W tym przypadku występuje obiekt `Method`, a dokładniej mówiąc, *powiązany* obiekt `Method`: nie jest to metoda `talk` teoretycznie, lecz raczej metoda `talk` specjalnie powiązana z obiektem `c`. Jeśli komunikat `call` wysyłasz obiektowi `meth`, w efekcie sam się wywołuje z obiektem `c` w roli obiektu `self`:

```
meth.call
```

Oto dane wyjściowe:

```
Test uzyskiwania obiektu Method! Obiekt self to #<C:0x00000101201a00>.
```

Możliwe jest też usunięcie powiązania metody z jej obiektem, a następnie powiązanie jej z innym obiektem, pod warunkiem że ten obiekt jest tej samej klasy co obiekt oryginalny (lub podklasa):

```
class D < C
end
d = D.new
unbound = meth.unbind
unbound.bind(d).call
```

W tym przypadku dane wyjściowe informują o tym, że metoda faktycznie była powiązana z obiektem klasy `D` (`d`) w momencie wykonania tej metody:

```
Test uzyskiwania obiektu Method! Obiekt self to #<D:0x000001011d0220>.
```

Aby bezpośrednio uzyskać obiekt, który nie jest powiązany z metodą, bez konieczności wywoływania metody `unbind` dla powiązanej metody, możesz pobrać obiekt z klasy, a nie z jej konkretnej instancji, za pomocą metody `instance_method`. Następujący pojedynczy wiersz odpowiada wywołaniu metod `method` i `unbind`:

```
unbound = C.instance_method(:talk)
```

Po ujarzmieniu niepowiązanej metody, jak to można ująć, za pomocą metody `bind` możesz powiązać ją z dowolną instancją klasy `C` lub jej podklasy (np. `D`).

Dlaczego jednak należałoby to robić?

### 14.3.2. Powody używania metod jako obiektów

Bez wątpienia wiązanie metod i usuwanie ich powiązania to specjalizowana technika. Prawdopodobnie będziesz mógł ograniczyć się tylko do jej biernej znajomości. Pomijając jednak regułę, zgodnie z którą bierna znajomość czegokolwiek w języku Ruby nie może być złym pomysłem, w pewnych sytuacjach najlepsza odpowiedź na pytanie „w jaki sposób?" brzmi: „za pomocą niepowiązanych metod".

Poniżej zaprezentowano przykład. Co jakiś czas na forach poświęconych językowi Ruby poruszana jest następująca kwestia:

*Załóżmy, że istnieje hierarchia klas, w której ponownie definiowana jest metoda:*

```
class A
  def a_method
    puts "Definicja w klasie A"
  end
end
class B < A
```

```
  def a_method
    puts "Definicja w klasie B (podklasa klasy A)"
  end
end
class C < B
end
```

*Ponadto istnieje instancja podklasy:*

```
c = C.new
```

*Czy możliwe jest spowodowanie w jakiś sposób, aby instancja najniższej klasy odpowiedziała na komunikat (a_method) przez wykonanie wersji metody w klasie, która jest położona dwie klasy wyżej w łańcuchu?*

Oczywiście domyślnie instancja nie realizuje tego działania. Wykonuje najpierw pierwszą dopasowaną metodę, jaką znajduje podczas przechodzenia ścieżki wyszukiwania:

```
c.a_method
```

Oto dane wyjściowe:

```
Definicja w klasie B (podklasa klasy A)
```

Możesz jednak wymusić działanie za pomocą operacji usuwania powiązania i wiązania:

```
A.instance_method(:a_method).bind(c).call
```

Oto dane wyjściowe:

```
Definicja w klasie A
```

Działanie to możesz nawet umiejscowić wewnątrz metody w klasie `C`:

```
class C
  def call_original
    A.instance_method(:a_method).bind(self).call
  end
end
```

A następnie wywołać metodę `call_original` bezpośrednio dla obiektu `c`.

Jest to przykład techniki języka Ruby z paradoksalnym statusem: należy ona do rzeczy, jakie powinien zrozumieć ktoś, kto chce biegle opanować zagadnienie dynamiki języka Ruby, lecz znajduje się poza sferą czegokolwiek, co należy prawdopodobnie zrobić. Jeśli wymuszasz udzielenie przez obiekty języka Ruby odpowiedzi na metody, które zostały już ponownie zdefiniowane, należy przejrzeć projekt programu i znaleźć sposób na wymuszenie wykonania przez obiekty żądanego działania jako konsekwencji utworzonej hierarchii klas/modułów, a nie wbrew niej.

Niemniej jednak metody to obiekty, które można wywoływać i które mogą być odłączane (poddawane operacji usuwania powiązania) od swoich instancji. Jako osoba, która chce poznać zagadnienie dynamiki języka Ruby, powinieneś przynajmniej dysponować wiedzą o tego rodzaju operacji na poziomie orientacyjnym.

Pozostaniemy przez chwilę w „dynamicznej stratosferze", zajmując się rodziną metod `eval`. Jest to niewielka grupa metod o specjalnych możliwościach, które pozwalają na użycie łańcuchów jako kodu, a także na modyfikowanie zasięgu i obiektu `self` na różne interesujące sposoby zależne od przypadku użycia.

**Alternatywne techniki wywoływania obiektów możliwych do wywołania**

Do tej pory do wywoływania obiektów możliwych do wywołania używaliśmy wyłącznie metody `call`. Dostępnych jest jednak kilka innych opcji.

Jedną z nich jest metoda/operator w postaci nawiasów kwadratowych, który jest synonimem metody `call`. Wewnątrz nawiasów kwadratowych umieszczasz dowolne argumenty:

```
mult = lambda {|x,y| x * y }
twelve = mult[3,4]
```

W przypadku braku argumentów nawiasy kwadratowe należy pozostawić puste.

Obiekty możliwe do wywołania możesz również wywołać za pomocą metody `()`:

```
twelve = mult.(3,4)
```

Zwróć uwagę na kropkę przed otwierającym nawiasem okrągłym. Metoda `()` musi zostać wywołana za pomocą kropki. Nie możesz po prostu dołączyć nawiasów okrągłych do obiektu `Proc` lub `Method` w sposób, jaki jest możliwy w przypadku nazwy metody. Jeśli nie występują argumenty, nawiasy okrągłe pozostaw puste.

## 14.4. Rodzina metod eval

Podobnie jak wiele innych języków język Ruby oferuje narzędzie służące do wykonywania kodu przechowywanego w postaci łańcuchów podczas działania programu. Okazuje się, że w języku Ruby dostępna jest grupa umożliwiających to technik, z których każda realizuje konkretny cel, ale wszystkie bazują na podobnej zasadzie odnoszącej się do działającego programu: „Niezależnie od tego, jakie łańcuchy kodu mogą zostać wczytane z pliku programu przed rozpoczęciem wykonywania jego kodu, *te* łańcuchy kodu uruchom natychmiast".

Metoda `eval` to najprostszy sposób przetwarzania łańcucha jako kodu, a także najbardziej niebezpieczny. Inne metody z rodziny metod `eval` są trochę „łagodniejsze", nie dlatego, że nie przetwarzają łańcucha jako kodu, ale ponieważ mają one też inne możliwości. Metoda `instance_eval` powoduje tymczasowe przesunięcie w wartości obiektu `self`, a metoda `class_eval` (znana również jako synonim `module_eval`) pozwala sprawdzić kontekst bloku definicji klasy. Te metody z rodziny metod `eval` mogą przetwarzać łańcuchy, ale też mogą być wywoływane z blokiem kodu. A zatem nie zawsze metody te działają tak otwarcie jak metoda `eval`, która przetwarza łańcuchy.

Rozwińmy to omówienie, zajmując się bliżej metodą `eval` oraz innymi metodami z rodziny `eval`.

### 14.4.1. Przetwarzanie dowolnych łańcuchów jako kodu za pomocą metody eval

Metoda przetwarza przekazany jej łańcuch:

```
>> eval("2+2")
=> 4
```

Metoda `eval` stanowi odpowiedzi lub przynajmniej jedną odpowiedź na wiele często zadawanych pytań, takich jak: „W jaki sposób utworzyć metodę i nadać jej nazwę, która zostanie przez kogoś wpisana?”. Możesz to zrealizować w następujący sposób:

```
print "Nazwa metody: "
m = gets.chomp
eval("def #{m}; puts 'Witaj!'; end")
eval(m)
```

Oto dane wyjściowe tego kodu:

```
Witaj!
```

Zapisywana jest nowa metoda. Załóżmy, że uruchamiasz kod i wpisujesz łańcuch `abc`. Łańcuch, dla którego następnie używasz metody `eval`, to:

```
def abc; puts 'Witaj!'; end
```

Po zastosowaniu metody `eval` dla tego łańcucha tworzona jest metoda o nazwie `abc`. Druga metoda `eval` przetwarza łańcuch `abc`, który — biorąc pod uwagę tworzenie metody w poprzednim wierszu — ustanawia wywołanie metody `abc`. Po jej wywołaniu wyświetlany jest komunikat `Wewnątrz nowej metody!`.

**Klasa Binding oraz przetwarzanie kodu przez metodę eval z użyciem powiązania**

W języku Ruby dostępna jest klasa o nazwie `Binding`, której instancje hermetyzują powiązania zmiennych lokalnych obowiązujące w danym miejscu wykonywanego kodu. Metoda najwyższego poziomu `binding` zwraca to, czym jest bieżące powiązanie.

Obiekty `Binding` są najczęściej stosowane jako drugi argument metody `eval`. Jeśli w tej roli użyjesz powiązania, łańcuch przetwarzany przez metodę `eval` będzie wykonywany w kontekście danego powiązania. Wszystkie zmienne lokalne używane w obrębie łańcucha tej metody są interpretowane w kontekście tego powiązania.

Pora na przykład. Metoda `use_a_binding` pobiera obiekt `Binding` jako argument i używa go jako drugiego argumentu wywołania metody `eval`. A zatem w ramach operacji wykonywanej przez tę metodę wykorzystywane są powiązania zmiennych lokalnych reprezentowane przez obiekt `Binding`:

```
def use_a_binding(b)
  eval("puts str", b)
end
str = "Jestem łańcuchem w powiązaniu najwyższego poziomu!"
use_a_binding(binding)
```

Dane wyjściowe tego fragmentu kodu to komunikat `Jestem łańcuchem w powiązaniu najwyższego poziomu!`. Łańcuch ten jest powiązany ze zmienną najwyższego poziomu `str`. Choć zmienna ta nie należy do zasięgu w obrębie metody `use_a_binding`, widoczna jest dla metody `eval` dzięki temu, że uzyskuje ona argument powiązania najwyższego poziomu, w którym zmienną `str` zdefiniowano i powiązano.

A zatem łańcuch `"puts str"`, który w przeciwnym razie spowodowałby wygenerowanie błędu (ponieważ nie zdefiniowano zmiennej `str`), z powodzeniem może zostać przetworzony przez metodę `eval` w kontekście danego powiązania.

Metoda `eval` zapewnia duże możliwości, ale też kryje w sobie zagrożenia, które w opinii niektórych osób są na tyle duże, że wykluczają ją jako przydatną technikę.

### 14.4.2. Zagrożenia stwarzane przez metodę eval

Z przetwarzaniem dowolnych łańcuchów związane jest znaczne zagrożenie. Dotyczy to zwłaszcza (choć nie tylko) łańcuchów, które pochodzą od użytkowników prowadzących interakcję z programem. Na przykład nie stanowiłoby problemu umieszczenie w poprzednim przykładzie szkodliwego polecenia, którym być może jest wywołanie systemowe `rm -rf /*`.

Metoda `eval` może być kusząca. W pełni niemal odzwierciedla ona możliwości techniki programowania dynamicznego: przetwarzasz łańcuchy kodu, które prawdopodobnie nawet nie istniały po utworzeniu programu. Gdziekolwiek interpreter języka Ruby stwarza swego rodzaju barierę pełnym możliwościom prostego wprowadzania zmian w konfiguracji działającego programu, metoda `eval` wydaje się oferować sposób na „przecięcie czerwonej taśmy" i wykonanie żądanych działań.

Jak się jednak okazuje, metoda `eval` nie jest panaceum. Jeśli użyjesz jej dla utworzonego łańcucha, generalnie nie jest to mniej bezpieczne od uruchomienia utworzonego pliku programu. Jednak każdorazowo, gdy ma się do czynienia z niepewnym, dynamicznie wygenerowanym łańcuchem, zagrożenia zwiększają się znacząco.

W szczególności trudno uporządkować dane wprowadzone przez użytkownika (w tym dane pochodzące z formularzy internetowych i plików) do stanu pozwalającego poczuć się bezpiecznie podczas stosowania metody `eval` dla tego łańcucha. W języku Ruby utrzymywana jest zmienna globalna o nazwie `$SAFE`, którą możesz ustawić na większą liczbę (w skali od 0 do 4), aby uzyskać ochronę przez takimi zagrożeniami, jak szkodliwe żądania zapisu do pliku. Zmienna ta sprawia, że korzystanie z metody `eval` jest o wiele bezpieczniejsze. Niemniej jednak najlepsze będzie wyrobienie sobie nawyku rezygnowania z metody `eval`.

Nie jest problemem znalezienie doświadczonych programistów używających języka Ruby (dotyczy to również programistów posługujących się innymi językami), którzy nigdy nie użyli metody `eval` i nie zamierzają tego zmieniać. Musisz sam zdecydować, jak postąpisz w tej kwestii, bazując na własnej znajomości „pułapek".

Zajmijmy się teraz szerszą rodziną metod `eval`. Umożliwiają one tego samego rodzaju wymuszone przetwarzanie łańcuchów co zapewniane przez metodę `eval`, ale oferują też „milsze" i „łagodniejsze" zachowania, które sprawiają, że są przydatne i wartościowe.

### 14.4.3. Metoda instance_eval

Metoda `instance_eval` to specjalizowany „krewny" metody `eval`. Metoda ta przetwarza przekazany łańcuch lub blok kodu, zmieniając obiekt `self` tak, że będzie odbiorcą wywołania metody `instance_eval`:

```
p self
a = []
a.instance_eval { p self }
```

Powyższy fragment kodu zwraca następujące dwa różne obiekty `self`:

```
main
[]
```

Metoda `instance_eval` przydaje się najbardziej w przypadku uzyskiwania dostępu do tego, co normalnie byłoby danymi prywatnymi innego obiektu (w szczególności zmiennych instancji). Oto kod prezentujący sposób wyświetlania wartości zmiennej instancji należącej do dowolnego starego obiektu (w tym przypadku zmiennej instancji `@x` obiektu klasy `C`):

```
class C
  def initialize
    @x = 1
  end
end
c = C.new
c.instance_eval { puts @x }
```

Tego rodzaju ingerowanie w stan innego obiektu jest generalnie uznawane za nieuprzejme. Jeśli obiekt domaga się uzyskania przez użytkownika jakichś informacji o jego stanie, to zapewnia metody, które to umożliwiają. Niemniej jednak z tego względu, że dynamika języka Ruby bazuje na zmienianiu obiektu `self`, nie jest złym pomysłem, aby język oferował technikę bezpośredniego modyfikowania obiektu `self`.

**Metoda instance_exec**

Metoda `instance eval` ma bliskiego „kuzyna" o nazwie `instance exec`. Różnica między nimi polega na tym, że metoda `instance_exec` może pobierać argumenty. Dowolne przekazane jej argumenty zostaną z kolei przekazane blokowi kodu.

Umożliwia to zastosowanie następujących wierszy kodu:

```
string = "Przykładowy tekst"
string.instance_exec("t") {|delim| self.split(delim) }
```

**Dane wyjściowe:**
`["Przykładowy ", "eks"]`

Nie jest to konieczne, jeśli znasz już separator, ale jest to podstawowa technika.

Niestety, odróżnianie tych metod, a także tego, która z nich pobiera argumenty, a która nie, wymaga po prostu zapamiętania. Łańcuch `eval` lub `exec` nie będzie tutaj w żadnym stopniu pomocny. Niemniej jednak warto „mieć pod ręką" obie metody.

Być może najczęściej metoda `instance_eval` jest używana w celu umożliwienia zastosowania następującego uproszczonego kodu przypisania:

```
david = Person.new do
  name "Dawid"
  age 55
end
```

Kod ten przypomina trochę kod z używanymi akcesorami, z tą różnicą, że nie ma jawnego odbiorcy i znaków równości. Jak zapewniłbyś działanie tego kodu?

Oto możliwy kod definicji klasy `Person`:

```
class Person
  def initialize(&block)
    instance_eval(&block)      ❶
  end
  def name(name=nil)           ❷
    @name ||= name             ❸
  end
```

```
  def age(age=nil)
    @age ||= age
  end
end
```

W tym przypadku kluczem jest wywołanie metody `instance_eval` ❶, która ponownie używa bloku kodu przekazanego już metodzie `new`. Ponieważ blok kodu jest przetwarzany przez metodę `instance_eval` w nowym obiekcie `Person` (niejawny obiekt `self` w definicji metody `initialize`), wywołania metod `name` i `age` mają miejsce w obrębie klasy `Person`. Z kolei te metody pełnią rolę hybrydy w postaci metod ustawiających/pobierających ❷: pobierają one opcjonalny argument z wartością domyślną `nil` i warunkowo ustawiają odpowiednie zmienne instancji na wartość tego argumentu. Jeśli wywołasz te metody bez argumentu, po prostu zwrócą one bieżącą wartość ich zmiennych instancji ❸.

Efektem jest możliwość użycia konstrukcji `name "Dawid"` zamiast konstrukcji `person.`
`↪name = "Dawid"`. Wielu programistów używających języka Ruby uważa tego rodzaju miniaturowy język dziedzinowy za całkiem zwięzły i elegancki.

Metody `instance_eval` (i `instance_exec`) bez żadnego problemu pobiorą też łańcuch i przetworzą go w przełączonym kontekście obiektu `self`. Jednak z techniką tą związane są te same pułapki co z przetwarzaniem łańcuchów za pomocą metody `eval`. Z tego powodu należy korzystać z niej ostrożnie (jeśli w ogóle).

Ostatnim członkiem rodziny metod `eval` jest metoda `class_eval` (jej synonim to `module_eval`).

### 14.4.4. Użycie metody class_eval (inaczej module_eval)

Zasadniczo metoda `class_eval` zapewnia dostęp do zawartości definicji klasy:

```
c = Class.new
c.class_eval do
  def some_method
    puts "Utworzono w metodzie class_eval"
  end
end
c_instance = c.new
c_instance.some_method   ←——— Dane wyjściowe: Utworzono w metodzie class_eval
```

Metoda `class_eval` umożliwia jednak działania, jakie są niedostępne w przypadku zwykłego słowa kluczowego `class`:

- Przetwarzanie łańcucha w kontekście definicji klasy.
- Otwieranie definicji klasy anonimowej.
- Użycie istniejących zmiennych lokalnych w treści definicji klasy.

Ostatni z powyższych punktów zasługuje na szczególną uwagę.

Gdy otwierasz klasę za pomocą słowa kluczowego `class`, inicjujesz nowy zasięg zmiennej lokalnej. Jednak blok używany z metodą `class_eval` może mieć dostęp do zmiennych utworzonych w otaczającym go zasięgu. Przyjrzyjmy się różnicy w traktowaniu zmiennej lokalnej zasięgu zewnętrznego `var` w treści zwykłej definicji klasy i bloku przekazanego metodzie `class_eval`:

```
>> var = "zainicjowana zmienna"
=> "zainicjowana zmienna"
>> class C
>>   puts var
>> end
NameError: undefined local variable or method `var' for C:Class
        from (irb):3
>> C.class_eval { puts var }
zainicjowana zmienna
```

Zmienna `var` znajduje się poza zasięgiem wewnątrz standardowego bloku definicji klasy, ale nadal w zasięgu w bloku kodu przekazanego metodzie `class_eval`.

Wszystko się komplikuje w momencie zdefiniowania metody instancji wewnątrz bloku metody `class_eval`:

```
>> C.class_eval { def talk; puts var; end }
=> nil
>> C.new.talk
NameError: undefined local variable or method `var' for #<C:0x350ba4>
```

Jak każda instrukcja `def` ta znajdująca się w obrębie bloku inicjuje nowy zasięg, dlatego zmienna `var` nie jest dłużej widoczna.

Aby umiejscowić zmienną zewnętrznego zasięgu w metodzie instancji, musisz użyć innej techniki tworzenia metody, czyli metody `define_method`. Metodzie tej przekazujesz nazwę metody do utworzenia (jako symbol lub łańcuch) oraz blok kodu, który pełni rolę treści metody.

W celu umieszczenia zewnętrznej zmiennej `var` w metodzie instancji klasy `C` użyj następującego wiersza:

```
>> C.class_eval { define_method("talk") { puts var }  }
=> :talk
```

Wartością zwracaną metody `define_method` jest symbol reprezentujący nazwę nowo zdefiniowanej metody.

Na tym etapie metoda instancji `talk` klasy `C` będzie mieć dostęp do zmiennej zewnętrznego zasięgu `var`:

```
>> C.new.talk
zainicjowano zmienną
```

Nie spotkasz się z takimi technikami stosowanymi tak często jak standardowe techniki definicji klas i metod. Gdy jednak je napotkasz, będziesz wiedzieć, że wskazują one „spłaszczony" zasięg dla zmiennych lokalnych, a nie nowy zasięg utworzony przez powszechniejsze słowa kluczowe `class` i `def`.

`define_method` to metoda instancji klasy `Module`, dlatego możesz ją wywołać dla dowolnej instancji klasy `Module` lub `Class`. A zatem metody tej możesz użyć wewnątrz treści definicji zwykłej metody (gdzie domyślnym odbiorcą `self` jest obiekt klasy), aby umieścić w metodzie instancji zmienną lokalną względem treści. Choć nie jest to często spotykana sytuacja, zdarza się.

Język Ruby umożliwia uproszczone programowanie współbieżne z wykorzystaniem wątków. Wątki to następne zagadnienie, którym się zajmiemy.

## 14.5. Równoległe wykonywanie za pomocą wątków

Wątki języka Ruby pozwalają na jednoczesne wykonanie w programie więcej niż jednego działania, co jest możliwe dzięki wariantowi współużytkowania czasu: jeden wątek wykonuje co najmniej jedną instrukcję, a następnie przekazuje sterowanie kolejnemu wątkowi itd. To, jak dokładnie funkcjonuje współbieżność wątków, zależy od używanego systemu i implementacji języka Ruby. Choć interpreter języka Ruby spróbuje użyć wbudowanych narzędzi obsługi wątków dostępnych w systemie operacyjnym, jeśli ich nie ma, sięgnie po *zielone* wątki (wątki implementowane całkowicie w obrębie interpretera). W tym miejscu nie będziemy rozwijać tematu dotyczącego porównywania wątków „zielonych" i wbudowanych. Skoncentrujemy się przede wszystkim na technikach i składni związanych z obsługą wątków.

Tworzenie wątków w języku Ruby jest proste: sprowadza się to do utworzenia instancji klasy `Thread`. Nowy wątek od razu rozpoczyna działanie. Wykonywanie kodu związanego z wątkiem nie jest jednak przerywane. Jeśli program zakończy pracę w czasie działania jednego lub większej liczby wątków, zostaną one zakończone.

Oto swego rodzaju kompletny przykład, który umożliwi rozpoczęcie pracy z wątkami. Przykład prezentuje zachowanie wątków w momencie kończenia działania programu:

```
Thread.new do
  puts "Uruchamianie wątku"
  sleep 1
  puts "Na końcu wątku"
end
puts "Poza wątkiem"
```

Metoda `Thread.new` pobiera blok kodu, który stanowi kod wykonywalny wątku. W tym przykładzie wątek wyświetla komunikat, wstrzymuje pracę na jedną sekundę, a następnie wyświetla kolejny komunikat. Jednak poza obrębem wątku czas upływa: główna zawartość programu powoduje natychmiastowe wyświetlenie komunikatu (nie ma na to wpływu polecenie `sleep` wewnątrz wątku), po czym program kończy pracę. Jeśli wyświetlanie komunikatu zajmuje więcej niż sekundę, konieczne będzie sprawdzenie używanego sprzętu! Drugi komunikat z wątku nigdy nie zostanie wyświetlony. Ujrzysz następujące wiersze:

```
Uruchamianie wątku
Poza wątkiem
```

A co będzie, gdy zażądamy zezwolenia na zakończenie wykonywania wątku? W tym celu konieczne jest użycie metody instancji `join`. Najprostszym sposobem zastosowania tej metody jest zapisanie wątku w zmiennej i wywołanie dla niej metody `join`. Oto zmodyfikowana wersja poprzedniego przykładu:

```
t = Thread.new do
  puts "Uruchamianie wątku"
  sleep 1
  puts "Na końcu wątku"
end
puts "Poza wątkiem"
t.join
```

Ta wersja programu generuje poniższe dane wyjściowe z jednosekundową pauzą między wyświetleniem pierwszego komunikatu z wątku i ostatniego komunikatu:

```
Uruchamianie wątku
Poza wątkiem
Na końcu wątku
```

**Wstrzymanie podczas oczekiwania przez program na zakończenie wykonywania wątku**

Oprócz dołączania możesz modyfikować wątek na wiele innych sposobów uwzględniających kończenie jego działania, przełączanie go w stan uśpienia, wznawianie jego działania i wymuszanie na nim przekazania sterowania następnemu wątkowi zaplanowanemu do wykonania.

### *14.5.1. Kończenie działania, zatrzymywanie i uruchamianie wątków*

Aby zakończyć działanie wątku, wysyłany jest do niego komunikat `kill`, `exit` lub `ter↪minate`. Wszystkie trzy są równorzędne sobie. Jeśli ma miejsce działanie w obrębie wątku, możesz wywołać metodę `kill` (lub jeden z jej synonimów) w postaci metody klasy `Thread`.

Może być pożądane zakończenie działania wątku, jeśli wewnątrz niego wystąpi wyjątek. Poniżej zaprezentowano przykład, co prawda trochę nienaturalny, ale na tyle krótki, że pozwala efektywnie zilustrować proces. Zamiarem jest wczytanie zawartości trzech plików (`part00`, `part01` i `part02`) do łańcucha `text`. Jeśli dowolny z plików nie zostanie znaleziony, wątek zakończy działanie:

```
puts "Podjęcie próby wczytania plików..."
t = Thread.new do
  (0..2).each do |n|
   begin
    File.open("part0#{n}") do |f|
      text << f.readlines
    end
    rescue Errno::ENOENT
      puts "Komunikat z wątku: niepowodzenie dla n=#{n}"
      Thread.exit
    end
  end
end
t.join
puts "Zakończono!"
```

Dane wyjściowe są następujące (przy założeniu, że plik `part00` istnieje, lecz brak pliku `part01`):

```
Podjęcie próby wczytania plików...
Komunikat z wątku: niepowodzenie dla n=1
Zakończono!
```

Możliwe jest też zatrzymywanie i uruchamianie wątków oraz sprawdzanie ich stanu. Wątek może być przełączany w stan uśpienia lub aktywowany, a także aktywny lub „martwy”. Poniżej zamieszczono przykład, w którym wątek poddawany jest testom. Przykład ilustruje niektóre dostępne techniki sprawdzania i modyfikowania stanu wątku:

```
t = Thread.new do
  puts "[Uruchamianie wątku]"   <------ [Uruchamianie wątku]
  Thread.stop
  puts "[Wznawianie wątku]"
end
puts "Status wątku: #{t.status}"                  <------ Status wątku: uśpiony
puts "Czy wątek został zatrzymany? #{t.stop?}"    <------ Czy wątek został zatrzymany? true
puts "Czy wątek jest aktywny? #{t.alive?}"        <------ Czy wątek jest aktywny? true
puts
puts "Aktywowanie wątku i dołączanie go..."
t.wakeup
t.join                          <------ [Wznawianie wątku]
puts
puts "Czy wątek jest aktywny? #{t.alive?}"        <------ Czy wątek jest aktywny? false
puts "Łańcuch inspekcji dla wątku: #{t.inspect}"  <------ Łańcuch inspekcji dla wątku: #<Thread:0x28d20 dead>
```

**Obiekty Fiber: coś innego w wątkach**

Oprócz wątków język Ruby zapewnia klasę `Fiber`. Obiekty `Fiber` są jak powracające bloki kodu: mogą one wielokrotnie przełączać się do swojego kontekstu wywołującego.

Obiekt `Fiber` jest tworzony za pomocą konstruktora `Fiber.new`, który pobiera blok kodu. Nic się nie dzieje do momentu nakazania obiektowi `Fiber` użycia metody `resume`, która powoduje rozpoczęcie wykonywania bloku kodu. W obrębie bloku za pomocą metody klasy `Fiber.yield` możesz wstrzymać działanie obiektu `Fiber`, zwracając sterowanie kontekstowi wywołującemu.

Oto prosty przykład uwzględniający *przekazujący informacje* obiekt `Fiber`, który kilkakrotnie przełącza sterowanie do swojego kontekstu wywołującego:

```
f = Fiber.new do
  puts "Witaj."
  Fiber.yield
  puts "Miłego dnia."
  Fiber.yield
  puts "Żegnaj!"
end
f.resume
puts "Z powrotem do obiektu Fiber:"
f.resume
puts "Ostatni komunikat z obiektu Fiber:"
f.resume
puts "To wszystko!"
```

Oto dane wyjściowe tego fragmentu kodu:

```
Witaj.
Z powrotem do obiektu Fiber:
Miłego dnia.
Ostatni komunikat z obiektu Fiber:
Żegnaj!
To wszystko!
```

Obiekty `Fiber` stanowią między innymi techniczną bazę enumeratorów, które używają ich do implementowania własnych operacji zatrzymywania i uruchamiania.

Kontynuujmy poznawanie wątków, posługując się dwoma przykładami dotyczącymi sieci: mowa o serwerze dat i czymś bardziej ambitnym, czyli serwerze rozmów sieciowych.

### *14.5.2. Serwer dat z wątkami*

Serwer dat, który zostanie utworzony, jest zależny od nieomawianego jeszcze narzędzia języka Ruby, czyli klasy `TCPServer`. Jest to oparta na gniazdach klasa, która umożliwia uruchomienie serwera z prawie niebywałą łatwością. Operacja polega na utworzeniu instancji klasy i przekazaniu numeru portu. Oto prosty przykład zastosowania tej klasy zapewniającej bieżącą datę pierwszej osobie, która nawiąże z serwerem połączenie:

```
require 'socket'
s = TCPServer.new(3939)
conn = s.accept
conn.puts "Witaj. Oto data."
conn.puts `date`
conn.close
s.close
```

**Słowo `date` ujęte w odwrotne apostrofy powoduje wykonanie polecenia systemowego `date`**

Umieść ten kod w pliku o nazwie *dateserver.rb* i uruchom go z poziomu wiersza poleceń (jeśli port 3939 jest niedostępny, zmień numer portu na jakiś inny). Używając innej konsoli, nawiąż połączenie z serwerem:

```
telnet localhost 3939
```

Zostaną wyświetlone dane wyjściowe podobne do następujących:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Witaj. Oto data.
Sat Jan 18 07:29:11 EST 2014
Connection closed by foreign host.
```

Serwer obsłużył żądanie i odpowiedział.

Co będzie, gdy serwer ma obsłużyć wiele żądań? Nic trudnego: nie zamykaj gniazda i zachowaj możliwość akceptowania połączeń.

```
require 'socket'
s = TCPServer.new(3939)
while true
  conn = s.accept
  conn.puts " Witaj. Oto data."
  conn.puts `date`
  conn.close
end
```

Tym razem możesz zażądać daty więcej niż raz. Za każdym razem otrzymasz odpowiedź.

Wszystko się komplikuje, gdy chcesz wysłać informacje *do* serwera. Zrealizowanie tego w przypadku jednego użytkownika nie stanowi problemu. Serwer może akceptować dane wejściowe, wywołując metodę `gets`:

```
require 'socket'
s = TCPServer.new(3939)
while true
  conn = s.accept
  conn.print "Witaj. Jak masz na imię? "
  name = conn.gets.chomp
```

**Akceptuje od klienta wiersz danych wejściowych wprowadzonych za pomocą klawiatury**

```
  conn.puts "Witaj, #{name}. Oto data."
  conn.puts `date`
  conn.close
end
```

Jeśli jednak drugi klient nawiązuje połączenie z serwerem, gdy w dalszym ciągu oczekuje on na dane wejściowe od pierwszego klienta, drugi klient nie uzyska żadnych informacji (nawet komunikatu `Jak masz na imię?`), ponieważ serwer jest zajęty.

W tej sytuacji przydatna okazuje się obsługa wątków. Poniżej zaprezentowano kod serwera dat z wątkami, który przyjmuje dane wejściowe od klienta. Obsługa wątków zapobiega zablokowaniu całej aplikacji, gdy oczekuje ona na udostępnienie danych wejściowych przez jednego klienta:

```
require 'socket'
s = TCPServer.new(3939)
while (conn = s.accept)          ←❶
  Thread.new(conn) do |c|        ←❷
    c.print "Witaj. Jak masz na imię? "
    name = c.gets.chomp          ←❸
    c.puts "Witaj, #{name}. Oto data."
    c.puts `date`
    c.close
  end
end
```

W tej wersji serwer nieustannie nasłuchuje połączeń ❶. Każdorazowo, gdy otrzyma połączenie, tworzy nowy wątek ❷. Znaczenie argumentu dla konstruktora `Thread.new` jest takie, że jeśli zostanie podany, zostanie z powrotem przekazany jako parametr bloku. W tym przypadku oznacza to powiązanie połączenia z parametrem `c`. Choć technika ta może wyglądać dziwnie (wysyłanie argumentu metodzie tylko po to, aby uzyskać go z powrotem w momencie wywołania bloku), zapewnia, że każdy wątek ma odwołanie do swojego własnego połączenia. Dzięki temu nie jest konieczne rywalizowanie o zmienną `conn`, która znajduje się poza każdym wątkiem.

Jeśli nawet dany klient czeka kilka minut przed wpisaniem imienia ❸, serwer nadal nasłuchuje nowych połączeń, a nowe wątki w dalszym ciągu są tworzone. A zatem obsługa wątków umożliwia serwerowi skalowanie podczas stosowania dwukierunkowej transmisji między nim samym i co najmniej jednym klientem.

Serwer rozmów sieciowych reprezentuje następny poziom złożoności.

### 14.5.3. Tworzenie serwera rozmów sieciowych używającego gniazd i wątków

Tym razem zaczniemy najpierw od kodu. Listing 14.2 prezentuje kod serwera rozmów sieciowych. Spora część realizowanych przez ten kod działań przypomina to, co wykonuje serwer dat. Główną różnicą jest to, że serwer rozmów sieciowych utrzymuje listę (tablicę) wszystkich połączeń przychodzących, a ponadto używa jej do rozgłaszania otrzymywanych wiadomości rozmów sieciowych.

**Listing 14.2. Serwer rozmów sieciowych korzystający z klasy TCPServer i wątków**

```
require 'socket'                          ❶
def welcome(chatter)                      ❷
  chatter.print "Witaj! Wprowadź imię: "
  chatter.readline.chomp
end
def broadcast(message, chatters)          ❸
  chatters.each do |chatter|
    chatter.puts message
  end
end
s = TCPServer.new(3939)                   ❹
   chatters = []
while (chatter = s.accept)                ❺
  Thread.new(chatter) do |c|
    name = welcome(chatter)               ❻
    broadcast("Osoba o imieniu #{name} dołączyła.", chatters)
    chatters << chatter
    begin                                 ❼
      loop do
        line = c.readline                 ❽
        broadcast("#{name}: #{line}", chatters)
      end
    rescue EOFError                       ❾
      c.close
      chatters.delete(c)                  ❿
      broadcast("Osoba o imieniu #{name} odłączyła się.", chatters)
    end
  end
end
```

Ponieważ wiele dzieje się w kodzie z tego listingu, zostanie to omówione w kolejności wykonywania. Najpierw ma miejsce obowiązkowe ładowanie biblioteki gniazd ❶. W kilku następnych wierszach definiowane są niezbędne metody pomocnicze. Powrócimy do nich po wyjaśnieniu, w czym są one pomocne. Prawdziwym początkiem działań jest tworzenie instancji klasy `TCPServer` i inicjalizacja tablicy z rozmawiającymi osobami ❹.

Serwer rozpoczyna wykonywanie pętli `while` podobnej do pętli w przypadku serwera dat ❺. W momencie nawiązania połączenia przez osobę rozmawiającą serwer wyświetla jej powitanie (niezależnie od tego, czy będzie *to* kobieta, czy mężczyzna) ❻. Proces powitania uwzględnia metodę `welcome` ❷, która jako swój argument pobiera obiekt gniazda reprezentujący osobę rozmawiającą, wyświetla ładny komunikat powitania i zwraca wiersz danych wejściowych klienta. Pora powiadomić wszystkich bieżących rozmówców o pojawieniu się nowej osoby. Wiąże się to z użyciem metody `broadcast` ❸, która stanowi „serce" funkcjonalności programu związanej z rozmowami sieciowymi. Metoda ta odpowiada za przetwarzanie tablicy z osobami rozmawiającymi i wysyłanie komunikatu do każdej z nich. W tym przypadku komunikat informuje o dołączeniu do rozmów sieciowych nowego klienta.

Po ogłoszeniu go nowy rozmówca jest dodawany do tablicy z osobami rozmawiającymi. Oznacza to, że zostanie on uwzględniony w przyszłych operacjach rozgłaszania komunikatów.

Pora na część dotyczącą rozmów sieciowych. Składa się ona z nieskończonej pętli opakowanej w klauzuli `begin/rescue` ❼. Celem jest ciągłe akceptowanie komunikatów od tego klienta, z tym że ma zostać podjęte działanie, gdy gniazdo klienta zgłosi koniec pliku. Komunikaty są akceptowane za pośrednictwem metody `readline` ❽, która nad metodą `gets` (w każdym razie w tej sytuacji) ma tę przewagę, że zgłasza wyjątek w momencie napotkania końca pliku. Jeśli osoba rozmawiająca zakończy sesję, następna próba wczytania wiersza pochodzącego od tej osoby spowoduje zgłoszenie wyjątku `EOFError`. Gdy do tego dojdzie, sterowanie przekazywane jest blokowi `rescue` ❾, w którym odłączona osoba rozmawiająca jest usuwana z tablicy z osobami rozmawiającymi, a następnie rozgłaszane jest powiadomienie o opuszczeniu sesji przez rozmówcę ❿.

Jeśli nie został utworzony wyjątek `EOFError`, komunikat osoby rozmawiającej jest rozgłaszany do wszystkich rozmówców ❽.

W przypadku korzystania z wątków ważna jest znajomość tego, jak w ich obrębie funkcjonują reguły tworzenia zasięgu zmiennych i ich widoczność. Dzięki omówieniu tego zagadnienia, co nastąpi w kolejnym punkcie, dowiesz się również o specjalnej kategorii zmiennych specyficznych dla wątków.

### 14.5.4. Wątki i zmienne

Do działania wątki wykorzystują bloki kodu, które rozpoznają zmienne już utworzone w ich zasięgu lokalnym. Jeśli utworzysz zmienną lokalną i zmodyfikujesz ją wewnątrz bloku kodu wątku, zmiana będzie trwała:

```
>> a = 1
=> 1
>> Thread.new { a = 2 }
=> #<Thread:0x390d8c run>
>> a
=> 2
```

Interesujący i pouczający efekt możesz zaobserwować w przypadku zatrzymania wątku przed zmodyfikowaniem przez niego zmiennej, a następnie uruchomienia wątku:

```
>> t = Thread.new { Thread.stop; a = 3 }
=> #<Thread:0x3e443c run>
>> a
=> 2
>> t.run
=> #<Thread:0x3e443c dead>
>> a
=> 3
```

W obliczu wątków zmienne globalne przeważnie takimi pozostają. Dotyczy to zarówno wbudowanych zmiennych globalnych, takich jak `$/` (separator rekordu danych wejściowych), jak i tworzonych we własnym zakresie:

```
>> $/
=> "\n"
>> $var = 1
=> 1
>> Thread.new { $var = 2; $/ = "\n\n" }
=> #<Thread:0x38dbb4 run>
>> $/
```

```
=> "\n\n"
>> $var
=> 2
```

Jednak niektóre zmienne globalne są *lokalne względem wątku*. Dokładniej rzecz biorąc, są to zmienne globalne `$1`, `$2`, ..., `$n`, którym są przypisywane wartości przechwytywania w nawiasach okrągłych pochodzące z najbardziej aktualnego wyrażenia regularnego operacji dopasowywania. W każdym wątku uzyskujesz inną porcję tych zmiennych. Oto fragment kodu ilustrujący to, że zmienne `$n` w różnych wątkach nie kolidują ze sobą:

```
/(abc)/.match("abc")
t = Thread.new do
  /(def)/.match("def")
  puts "Zmienna $1 w wątku: #{$1}"
end.join
puts "Zmienna $1 poza wątkiem: #{$1}"
```

**Dane wyjściowe:**
`Zmienna $1 w wątku: def`

**Dane wyjściowe:**
`Zmienna $1 poza wątkiem: abc`

Powody takiego zachowania są oczywiste: zmienna `$1` jednego wątku nie może przesłaniać zmiennej `$1` z innego wątku, ponieważ w przeciwnym razie zostaną uzyskane wyjątkowo dziwne wyniki. Zmienne `$n` nie są w rzeczywistości zmiennymi globalnymi, co się okaże po wyświetleniu ich w kontekście języka zawierającego wątki.

Oprócz uzyskiwania dostępu do zwykłego zestawu zmiennych języka Ruby wątki dysponują też własnym magazynem zmiennych, a dokładniej rzecz biorąc, wbudowaną tablicą asocjacyjną, która umożliwia im skojarzenie symboli lub łańcuchów z wartościami. Takie klucze wątków mogą okazać się przydatne.

### 14.5.5. Modyfikowanie kluczy wątków

*Klucze wątków* to zasadniczo tablica asocjacyjna przechowująca wartości specyficzne dla wątku. Klucze muszą być symbolami lub łańcuchami. Dostęp do kluczy zapewnia bezpośrednie indeksowanie obiektu wątku za pomocą wartości w nawiasach kwadratowych. Przy użyciu metody `keys` możliwe jest też uzyskanie listy wszystkich kluczy (bez ich wartości).

Oto prosty scenariusz z operacjami ustawiania i pobierania oparty na kluczu wątku:

```
t = Thread.new do
  Thread.current[:message] = "Witaj"
end
t.join
p t.keys
puts t[:message]
```

Oto dane wyjściowe:

```
[:message]
Witaj
```

Wątki wydają się dominować w grach, dlatego użyjemy przykładu z grą, aby szerzej omówić klucze wątków. Będzie to gra sieciowa z wątkami w kamień, papier i nożyce. Zaczniemy od logiki gry (bez wątków) w klasie `RPS` i użyjemy wynikowej biblioteki gry jako bazy dla jej kodu.

### PODSTAWOWA IMPLEMENTACJA LOGIKI GRY W KAMIEŃ, PAPIER I NOŻYCE

Poniższy listing 14.3 prezentuje klasę `RPS` opakowaną w moduł `Games` (użyto takiej nazwy, ponieważ nazwa *RPS* może kolidować z nazwą innej klasy). Kod z listingu zapisz w pliku o nazwie *rps.rb*.

**Listing 14.3. Logika gry w kamień, papier i nożyce zawarta w klasie Games::RPS**

```
module Games
  class RPS
    include Comparable          ❶
    WINS = [%w{ rock scissors },
            %w{ scissors paper },  ❷
            %w{ paper rock }]
    attr_accessor :move         ❸
    def initialize(move)        ❹
      @move = move.to_s
    end
    def <=>(other)              ❺
      if move == other.move
        0
      elsif WINS.include?([move, other.move])
        1
      elsif WINS.include?([other.move, move])
        -1
      else
        raise ArgumentError, "Coś się nie zgadza."
      end
    end
    def play(other)             ❻
      if self > other
        self
      elsif other > self
        other
      else
        false
      end
    end
  end
end
```

Klasa `RPS` obejmuje moduł `Comparable` ❶. Moduł stanowi bazę do określenia, kto ostatecznie zwycięży w grze. Stała `WINS` zawiera w trzech tablicach wszystkie możliwe zwycięskie kombinacje. Pierwszy element każdej tablicy pokonuje drugi element ❷. Istnieje też atrybut `move`, który przechowuje ruch w grze dla danej instancji klasy `RPS` ❸. Metoda `initialize` ❹ przechowuje ruch w grze jako łańcuch (w sytuacji, gdy ruch pojawia się jako symbol).

Klasa `RPS` zawiera definicję metody operatora „statku kosmicznego” (`<=>`) ❺, który określa, co ma miejsce, gdy dana instancja tej klasy jest porównywana z inną instancją. Jeśli obie instancje mają jednakowe ruchy w grze, wynikiem jest zero (oznacza, że równe są dwa warunki porównania z użyciem operatora `<=>`). Reszta logiki szuka zwycięskich kombinacji przy użyciu tablicy `WINS`, zwracając wartość `-1` lub `1`, zależnie

od tego, czy wygrała pierwsza, czy druga instancja. Jeśli żaden gracz nie zostanie ustalony jako zwycięzca, wynikiem jest remis, co powoduje zgłoszenie wyjątku.

Gdy już obiekty `RPS` potrafią się porównywać, z łatwością można spowodować zagranie ich ze sobą. Odpowiada za to metoda `play` ❻. Reguły gry są proste: zwycięzcą jest dowolny gracz z silniejszym symbolem, a w przypadku remisu metoda zwraca wartość `false`.

Jesteśmy już gotowi do zastosowania klasy `RPS` w wersji sieciowej gry z wątkami, kluczach wątków oraz pozostałych elementach.

### UŻYCIE KLASY RPS W GRZE Z WĄTKAMI

Poniższy listing 14.4 prezentuje program gry sieciowej w kamień, papier i nożyce. Program oczekuje na dołączenie do gry dwóch osób, a następnie pobiera ich ruchy, wyświetla wynik i kończy działanie. Nie jest to efekciarski, ale dobry sposób pokazania, jak mogą okazać się pomocne klucze z wątkami.

**Listing 14.4. Program gry sieciowej z wątkami w kamień, papier i nożyce, który korzysta z kluczy wątków**

```
require 'socket'
require_relative 'rps'
s = TCPServer.new(3939)                                  ❶
threads = []                                             ❷
2.times do |n|
  conn = s.accept                                        ❸
  threads << Thread.new(conn) do |c|                     ❹
    Thread.current[:number] = n + 1
    Thread.current[:player] = c
    c.puts "Witaj, graczu #{n+1}!"
    c.print "Jaki jest Twój ruch? (kamień, papier, nożyce) "
    Thread.current[:move] = c.gets.chomp
    c.puts "Dzięki,... poczekaj."
  end
end
a,b = threads                                            Użycie składni przypisania równoległego
a.join                                                   w celu przypisania dwóch zmiennych z tablicy ❺
b.join
rps1, rps2 = Games::RPS.new(a[:move]), Games::RPS.new(b[:move])   ❻
winner = rps1.play(rps2)
if winner
  result = winner.move                                   ❼
else
  result = "REMIS!"
end
threads.each do |t|
  t[:player].puts "Zwycięzcą jest #{result}!"            ❽
end
```

Program ładuje klasę `Games::RPS` i jej używa, dlatego upewnij się, że plik *rps.rb* z kodem logiki gry znajduje się w tym samym katalogu co sam program.

Jak w przykładzie serwera rozmów sieciowych zaczniemy od serwera ❶ z tablicą, w której przechowywane są wątki ❷. Zamiast jednak wykonywać w nieskończoność pętlę, pobieramy tylko dwa wątki, co jest możliwe dzięki pętli `2.times` i metodzie `accept` serwera ❸. Dla każdego z dwóch połączeń tworzymy wątek ❹.

Kilka wartości przechowujemy w kluczach wątku: liczbę dla danego gracza (określaną przy użyciu pętli `times` i przez dodanie wartości 1, co powoduje, że nie ma gracza 0) i połączenia. Po powitaniu gracza jego ruch jest przechowywany w kluczu `:move` wątku.

Gdy obaj gracze wykonają swoje ruchy, dwa wątki są umieszczane w wygodnych zmiennych `a` i `b`, po czym oba wątki są łączone ❺. W dalszej kolejności dwa obiekty wątku z zapamiętanymi ruchami graczy są przekształcane w dwa obiekty `RPS` ❻. Zwycięzca jest ustalany przez rozegranie przez graczy partii. Końcowym wynikiem gry jest wyłonienie zwycięzcy lub remis, w sytuacji gdy w grze została zwrócona wartość `false` ❼.

Na końcu obaj gracze są informowani o wynikach ❽. Gra może zostać bardziej urozmaicona przez wprowadzanie imion graczy lub powtarzanie jej i zachowywanie wyniku. Jednak głównym celem tej wersji gry jest zilustrowanie przydatności kluczy wątków. Nawet po zakończeniu działania wątków zachowują one informacje. Umożliwia to rozegranie całej gry, a także wysyłanie dalszych komunikatów za pośrednictwem gniazd graczy.

Kończymy omawianie wątków języka Ruby. Warto zauważyć, że wątki to obszar, który ma już za sobą i nadal przed sobą wiele zmian oraz prac projektowych. Cokolwiek jednak się wydarzy, możesz bazować na uzyskanych tutaj podstawowych informacjach podczas dalszego poznawania i używania wątków.

Następną pozycją w programie i ostatnią w tym rozdziale jest zagadnienie związane z wykonywaniem poleceń systemowych z poziomu kodu Ruby.

## 14.6. Wykonywanie poleceń systemowych w obrębie programów Ruby

Język Ruby oferuje kilka sposobów wykonywania poleceń systemowych. Przyjrzymy się przede wszystkim dwóm z nich: metodzie system i technice opartej na odwróconych apostrofach (``). Inne sposoby komunikowania się z programami systemu obejmują programowanie na niższym poziomie. Ponadto są one w większym stopniu zależne od systemu, a tym samym w pewnym sensie wykraczają poza zakres książki. Niemniej jednak omówimy je pobieżnie. Jeśli uznasz, że wydają się być czymś, co może być potrzebne, możesz je bliżej poznać.

### 14.6.1. Metoda system i odwrócone apostrofy

Metoda `system` wywołuje program systemowy. Odwrócone apostrofy (``) powodują wywołanie programu systemowego i zwrócenie jego danych wyjściowych. To, co zostanie wybrane, zależy od tego, co chcesz zrealizować.

### WYKONYWANIE PROGRAMÓW SYSTEMOWYCH ZA POMOCĄ METODY SYSTEM

Aby użyć metody `system`, wyślij jej nazwę programu do uruchomienia z dowolnymi argumentami. Program używa bieżących wartości stałych `STDIN`, `STDOUT` i `STDERR`. Poniżej zamieszczono trzy proste przykłady. Polecenia `cat` i `grep` wymagają użycia kombinacji klawiszy *Ctrl+D* (lub dowolnej kombinacji klawiszy używanego systemu, która wskazuje „koniec pliku") w celu ich zakończenia i przekazania sterowania z powrotem narzędziu irb. Dla zwiększenia przejrzystości dane wyjściowe programu Ruby pogrubiono, a dla danych wprowadzonych przez użytkownika zastosowano zwykłą czcionkę:

```
>> system("date")
Sat Jan 18 07:32:11 EST 2014
=> true
>> system("cat")
Wpisuję dane na ekranie dla polecenia cat.
Wpisuję dane na ekranie dla polecenia cat.
=> true
>> system('grep "D"')
jeden
dwa
Dawid
Dawid
```

W przypadku użycia metody `system` dla zmiennej globalnej ustawiany jest obiekt `Pro`
`↪cess::Status` zawierający informacje o wywołaniu: dokładniej rzecz biorąc, jest to identyfikator właśnie uruchomionego procesu oraz jego status wyjścia. Poniżej zamieszczono wywołania poleceń `date` i `cat`, z których drugie jest kończone za pomocą kombinacji klawiszy *Ctrl+C*. Po każdym wywołaniu następuje sprawdzenie zmiennej `$?`:

```
>> system("date")
Sat Jan 18 07:32:11 EST 2014
=> true
>> $?
=> #<Process::Status: pid 28025 exit 0>
>> system("cat")
^C=> false
>> $?
=> #<Process::Status: pid 28026 SIGINT (signal 2)>
```

Oto wywołanie nieistniejącego programu:

```
>> system("datee")
=> nil
>> $?
=> #<Process::Status: pid 28037 exit 127>
```

Zmienna `$?` jest lokalna względem wątku: jeśli wywołasz program w jednym wątku, jego wartość zwracana będzie dotyczyć tylko zmiennej `$?` w tym wątku:

```
>> system("date")
Sat Jan 18 07:32:11 EST 2014
=> true
>> $?
=> #<Process::Status: pid 28046 exit 0>          ←❶
>> Thread.new { system("datee"); p $? }.join     ←❷
#<Process::Status: pid 28047 exit 127>           ←❸
```

```
=> #<Thread:0x3af840 dead>
>> $?
=> #<Process::Status: pid 28046 exit 0>        ←❹
```

Obiekt `Process::Status` raportujący dla wywołania polecenia `date` jest przechowywany w zmiennej `$?` w głównym wątku ❶. Nowy wątek tworzy wywołanie nieistniejącego programu ❷. Wersja zmiennej `$?` tego wątku odzwierciedla problem ❸. Jednak zmienna `$?` głównego wątku pozostaje niezmieniona ❹. Zachowanie zmiennej globalnej lokalnej względem wątku przypomina (z podobnych powodów) w dużym stopniu zachowanie zmiennych przechwytywania wyrażenia regularnego `$n`. W obu przypadkach nie chcesz, aby wątek reagował na warunek błędu, którego nie spowodował i który nie odzwierciedla rzeczywistego przepływu programu.

Technika oparta na odwróconych apostrofach jest blisko spokrewniona z metodą `system`.

### WYWOŁYWANIE PROGRAMÓW SYSTEMOWYCH ZA POMOCĄ ODWRÓCONYCH APOSTROFÓW

Aby wykonać polecenie systemowe przy użyciu odwróconych apostrofów, ujmij je w tego rodzaju apostrofy. Główną różnicą między metodą system i odwróconymi apostrofami jest to, że wartość zwracana wywołania z odwróconymi apostrofami ma postać danych wyjściowych działającego programu:

```
>> d = `date`
=> "Sat Jan 18 07:32:11 EST 2014\n"
>> puts d
Sat Jan 18 07:32:11 EST 2014
=> nil
>> output = `cat`
Wpisuję dane dla polecenia cat. Ponieważ używam odwróconych apostrofów,
podczas wpisywania każdego wiersza nie będzie on powtarzany na ekranie.
Dane wyjściowe polecenia cat zostaną skierowane do
danych wyjściowych zmiennej.
=> "Wpisuję dane dla polecenia cat. Ponieważ używam odwróconych apostrofów,\npodczas
wpisywania itd.
>> puts output
Wpisuję dane dla polecenia cat. Ponieważ używam odwróconych apostrofów,
podczas wpisywania każdego wiersza nie będzie on powtarzany na ekranie.
Dane wyjściowe polecenia cat zostaną skierowane do
danych wyjściowych zmiennej.
```

Odwrócone apostrofy powodują ustawienie zmiennej `$?` w taki sam sposób jak metoda `system`. Wywołanie nieistniejącej metody za pomocą odwróconych apostrofów kończy się wygenerowaniem błędu krytycznego:

```
>> `datee`
Errno::ENOENT: No such file or directory - datee
>> $?
=> #<Process::Status: pid 28094 exit 127>
>> `date`
=> "Sat Jan 18 07:35:32 EST 2014\n"
>> $?
=> #<Process::Status: pid 28095 exit 0>
```

**Dodatki związane z poleceniami systemowymi**

Istnieje jeszcze jeden sposób wykonywania poleceń systemowych z poziomu kodu Ruby, czyli operator `%x`. Na przykład kod `%x{date}` spowoduje wykonanie polecenia `date`. Podobnie jak odwrócone apostrofy operator `%x` zwraca łańcuchowe dane wyjściowe polecenia. Tak jak jego „krewni", czyli operatory `%w` i `%q` (między innymi), operator `%x` zezwala na użycie dowolnego separatora, pod warunkiem że występuje zgodność separatorów umieszczanych w nawiasach: konstrukcje `%x{date}`, `%x-date-` i `%x(date)` są synonimami.

Zarówno odwrócone apostrofy, jak i operator `%x` umożliwiają interpolację łańcucha:

```
polecenie = "date"
%x(#{polecenie})
```

Może to być wygodne, choć zapewne niewiele będzie okazji, w których dobrym pomysłem okaże się wywoływanie jako poleceń systemowych dynamicznie przetwarzanych łańcuchów.

Odwrócone apostrofy są wyjątkowo przydatne przy przechwytywaniu danych wyjściowych programu zewnętrznego, ale nie jest to jedyny sposób, który to umożliwia. Tak oto przechodzimy do trzeciego sposobu uruchamiania programów z poziomu programu Ruby. Mowa o metodach `open` i `Open3.popen3`.

### *14.6.2. Komunikacja z programami za pośrednictwem metod open i Open3.popen3*

Użycie rodziny metod `open` do wywoływania programów zewnętrznych jest znacznie bardziej złożone niż zastosowanie metody `system` i odwróconych apostrofów. Przyjrzymy się kilku prostym przykładom, ale nie będziemy zagłębiać się w to zagadnienie. Takie metody języka Ruby są odwzorowywane bezpośrednio na bazowe wywołania biblioteki systemowej, które je obsługują. Działanie tych metod może się różnić między poszczególnymi systemami w większym stopniu niż działanie samego języka Ruby.

Niemniej jednak zajmijmy się tym. Zostaną omówione dwie metody: metoda `open` i metoda klasy `Open3.popen3`.

#### KOMUNIKACJA Z PROGRAMAMI ZEWNĘTRZNYMI ZA POMOCĄ METODY OPEN

Metoda najwyższego poziomu `open` pozwala na dwukierunkową komunikację z programem zewnętrznym. Oto poczciwy przykład polecenia `cat`:

```
>> d = open("|cat", "w+")        ←❶
=> #<IO:fd 11>
>> d.puts "Witaj w poleceniu cat" ←❷
=> nil
>> d.gets                        ←❸
=> "Witaj w poleceniu\n"
>> d.close                       ←❹
=> nil
```

Wywołanie metody `open` jest ogólne. Może to być dowolny strumień operacji wejścia-wyjścia, ale w tym przypadku użyto dwukierunkowego połączenia z poleceniem systemowym ❶. Znak potoku przed słowem `cat` wskazuje, że żądana jest komunikacja z programem, a nie otwarcie pliku. Uchwyt programu zewnętrznego działa bardzo podobnie do gniazda operacji wejścia-wyjścia lub uchwytu pliku. Uchwyt ten otwierany

jest do odczytu i zapisu (tryb w+), dlatego w jego przypadku możliwe są operacje zapisywania ❷ i odczytywania ❸. Na końcu uchwyt jest zamykany ❹.

Możliwe jest też skorzystanie z blokowej formy metody open i zapisanie ostatniego kroku:

```
>> open("|cat", "w+") {|p| p.puts("witaj"); p.gets }
=> "witaj\n"
```

Trochę bardziej wyszukanym i oferującym duże możliwości sposobem realizowania dwukierunkowej komunikacji między programem Ruby i programem zewnętrznym jest użycie metody Open3.popen3.

### DWUKIERUNKOWA KOMUNIKACJA ZA POMOCĄ METODY OPEN3.POPEN3

Metoda Open3.popen3 inicjuje komunikację z programem zewnętrznym i zapewnia uchwyty dla jego standardowego strumienia danych wejściowych, standardowego strumienia danych wyjściowych i standardowego strumienia błędów. Oznacza to, że przy użyciu tych uchwytów możesz dokonywać zapisu i odczytu niezależnie od analogicznych strumieni własnego programu.

Oto prosty przykład metody Open3.popen3 oparty na poleceniu cat:

```
>> require 'open3'                                  ❶
=> true
>> stdin, stdout, stderr = Open3.popen3("cat")      ❷
=> [#<IO:fd 10>, #<IO:fd 11>, #<IO:fd 13>,          ❸
   #<Thread:0x000001011356f8 sleep>]
>> stdin.puts("Witaj.\nŻegnaj")                     ❹
=> nil
>> stdout.gets                                      ❺
=> "Witaj.\n"
>> stdout.gets
=> "Żegnaj\n"
```

Po załadowaniu biblioteki open3 ❶ wywoływana jest metoda Open3.popen3, której przekazywana jest nazwa programu zewnętrznego ❷. W odpowiedzi otrzymujemy trzy uchwyty operacji wejścia-wyjścia oraz wątek ❸, który możesz zignorować. Uchwyty te są przekazywane do programu zewnętrznego oraz z niego. Możliwy jest zatem zapis w uchwycie strumienia STDIN ❹ i odczyt wierszy z uchwytu strumienia STDOUT ❺. Uchwyty te nie są tymi samymi co w przypadku strumieni STDIN i STDOUT samej sesji narzędzia irb.

Następny przykład prezentuje trochę bardziej zaawansowane użycie metody Open3.
↪popen3. Bądź świadom tego, że sam w sobie przykład jest trywialny. Jego celem jest zilustrowanie podstawowego mechanizmu techniki. Ponieważ w przykładzie zastosowano wątki, prezentuje również wybrane techniki związane z wątkami. Poniższy listing 14.5 zawiera kod.

**Listing 14.5. Użycie metody Open3.popen3 i wątków do modyfikowania sesji polecenia cat**

```
require 'open3'
stdin, stdout, stderr = Open3.popen3("cat")
t = Thread.new do                                   ❶
```

```
  loop { stdin.puts gets }
end
u = Thread.new do          ←❷
  n = 0
  str = ""
  loop do
     str << stdout.gets
     n += 1
     if n % 3 == 0         ←❸
       puts "--------\n"
       puts str
       puts "--------\n"
       str = ""
     end
  end
end
t.join
u.join
```

Program tworzy dwukierunkowy potok dla polecenia `cat` i używa dwóch wątków na potrzeby komunikacji z tym potokiem i nasłuchiwania go. Pierwszy wątek `t` ❶ działa w ramach nieskończonej pętli, prowadząc nasłuch strumienia `STDIN` (powiązanego z używanym programem, a nie z poleceniem `cat`), a także zapisując każdy wiersz w uchwycie strumienia `STDIN` w procesie polecenia `cat`. Drugi wątek `u` ❷ utrzymuje licznik (`n`) oraz akumulator łańcuchów (`str`). Gdy licznik osiągnie wielokrotność liczby 3, co wskazuje test `modulo` ❸, wątek `u` wyświetla linię poziomą, trzy zakumulowane dotąd wiersze tekstu oraz kolejną linię poziomą. Licznik resetuje następnie akumulator łańcuchów do postaci pustego łańcucha i powraca do nasłuchiwania.

Jeśli uruchomisz ten program, pamiętaj, że będzie bez końca wykonywany, dlatego konieczne jest zakończenie jego działania za pomocą kombinacji klawiszy *Ctrl+C* (lub dowolnych klawiszy służących w używanym systemie do wygenerowania sygnału przerwania działania). Zgodnie z przewidywaniami dane wyjściowe nie są czymś ekscytującym, ale pozwalają dobrze zorientować się, w jaki sposób wątki prowadzą interakcję ze sobą oraz z uchwytami operacji wejścia-wyjścia. W następujących danych wyjściowych wiersze wprowadzone przez użytkownika wyróżniono kursywą:

```
Jeden
Dwa
Trzy
--------
Jeden
Dwa
Trzy
--------
Cztery
Pięć
Sześć
--------
Cztery
Pięć
Sześć
--------
```

Jak już wspomniano, nie zamierzamy zajmować się szczegółowo metodą `Open3.popen3`. Możesz jednak i powinieneś skorzystać z przedstawionych tutaj informacji w sytuacjach, gdy wymagana będzie maksymalna elastyczność w zakresie wykonywania operacji odczytywania i zapisywania dla programu zewnętrznego.

## 14.7. Podsumowanie

W rozdziale omówiono następujące zagadnienia:

- Obiekty `Proc`.
- Wariant procesu oparty na metodzie `lambda`.
- Przekształcanie bloku kodu w obiekt `Proc` (i odwrotnie).
- Metoda `Symbol#to_proc`.
- Obiekty metody.
- Powiązania.
- Metody `eval`, `instance_eval` i `class_eval`.
- Użycie i modyfikowanie wątków.
- Zmienne „globalne" lokalne względem wątku.
- Metoda `system`.
- Wywoływanie poleceń systemowych za pomocą odwróconych apostrofów.
- Podstawowe informacje o metodach `open` i `Open3.popen3`.

Obiekty w języku Ruby są produktami wykonywania kodu w środowisku wykonawczym, ale same mogą zapewniać możliwości wykonywania kodu. W rozdziale omówiono kilka sposobów funkcjonowania ogólnego pojęcia obiektów umożliwiających wywoływanie i uruchamianie. Przyjrzeliśmy się obiektom `Proc` i `lambda`, czyli funkcjom anonimowym tkwiącym w samym środku składni bloków języka Ruby. Omówiono też metody pełniące rolę obiektów, a także sposoby wiązania metod i usuwania ich powiązania oraz stosowania ich niezależnie od wywołujących je obiektów. Rodzina metod `eval` wprowadziła nas w obszar przetwarzania dowolnych łańcuchów, a ponadto umożliwiła poznanie wybranych technik cechujących się elegancją i dużymi możliwościami, które służą do modyfikowania hierarchii klas i obiektów działającego programu. W tym celu używana jest nie tylko metoda `eval`, ale nawet metody `class_eval` i `instance_eval` z ich operacjami dotyczącymi bloków.

Wśród obiektów wykonywalnych języka Ruby wątki wybijają się na pierwszy plan. Każdy program działa w głównym wątku, nawet jeśli nie utworzy żadnych innych wątków. Objaśniliśmy składnię i semantykę wątków, a także zaprezentowaliśmy, jak ułatwiają realizowanie takich projektów, jak komunikacja sieciowa między wieloma użytkownikami.

Na końcu przyjrzeliśmy się różnym sposobom zapewnianym przez język Ruby do wykonywania programów zewnętrznych. Uwzględniono stosunkowo prostą w użyciu metodę `system` i technikę opartą na odwróconych apostrofach oraz trochę bardziej szczegółowe i złożone metody `open` i `Open3.popen3`.

Nie istnieje żadna konkretna definicja obiektu umożliwiającego wywoływanie i uruchamianie. W rozdziale celowo posłużono się dość płynną techniką pozwalającą

zrozumieć pojęcia. Z jednej strony, taka płynność powoduje zestawienie zagadnień, którym — jak można sobie wyobrazić — mogłyby zostać poświęcone osobne rozdziały (trudno polemizować z tym, że na przykład między metodami `instance_eval` i `Open3.popen3` występuje jakiekolwiek bliskie i bezpośrednie pokrewieństwo). Z drugiej strony, szczegóły języka Ruby w dużej mierze manifestują podstawowe i dodatkowe zasady. Ponadto ważne jest pojęcie obiektów, które uczestniczą bezpośrednio w dynamice elementów języka Ruby. Choć zagadnienia poruszone w rozdziale mogą pod pewnymi względami być zasadniczo odmienne, wszystkie są zbieżne z podstawowymi zasadami. Dobre opanowanie podstaw związanych z tymi zagadnieniami w znaczącym stopniu zwiększy możliwości w zakresie korzystania z języka Ruby.

Pora przejść do następnego, już ostatniego, podstawowego zagadnienia, czyli mechanizmu refleksji w czasie wykonywania kodu, introspekcji i wywołań zwrotnych.

# 15 Wywołania zwrotne, „haki” i introspekcja w czasie wykonywania kodu

**W tym rozdziale:**

- Wywołania zwrotne środowiska wykonawczego: `inherited`, `included` i inne.
- Metody `respond_to?` i `method_missing`.
- Introspekcja list obiektów i metod klasy.
- Wychwytywanie nierozstrzygniętych odwołań do stałych.
- Sprawdzanie zmiennych i stałych w zasięgu.
- Analizowanie informacji śledzenia stosu i elementu wywołującego.

Przy swojej dynamicznej naturze i wspieraniu elastycznego projektu dotyczącego obiektów i programów język Ruby zapewnia dużą liczbę sposobów pozwalających sprawdzać, co ma miejsce w czasie działania programu, a także definiować oparte na zdarzeniach wywołania zwrotne i „haki” (zasadniczo są to „przewody” pociągane w konkretnych momentach i z określonych powodów) w postaci metod o specjalnych, zarezerwowanych nazwach, dla których w razie potrzeby możesz udostępnić definicje. A zatem możesz tak przygotować moduł, że określona metoda będzie wywoływana

każdorazowo w momencie dołączenia tego modułu przez klasę. Możesz też utworzyć metodę wywołania zwrotnego dla klasy, która będzie wywoływana każdorazowo w przypadku dziedziczenia klasy itd.

Oprócz wywołań zwrotnych środowiska wykonawczego język Ruby umożliwia wykonywanie bardziej pasywnych, lecz często kluczowych działań sprawdzających: możesz zażądać od obiektów informacji o wykonywanych przez nie metodach (korzystając z jeszcze większej liczby sposobów niż dotychczas zaprezentowano) lub zmiennych instancji, jakie zawierają. Możliwe jest odpytywanie klas i modułów w odniesieniu do ich stałych i metod instancji. Istnieje możliwość sprawdzania danych śledzenia stosu w celu określenia, jakie wywołania metod miały miejsce do konkretnego miejsca w programie. Możesz nawet uzyskać dostęp do nazw plików i numerów wierszy wszystkich wywołań metod, jakie wystąpiły.

Podsumowując, język Ruby zaprasza na przyjęcie: dowiesz się bardzo dokładnie, co się dzieje, za pośrednictwem technik introspekcji środowiska wykonawczego. Możesz nakazać językowi Ruby „naciśnięcie konkretnych przycisków" w odpowiedzi na zdarzenia środowiska wykonawczego. W tym ostatnim rozdziale książki objaśnimy różnorodność tych technik wywołań zwrotnych i introspekcji, a ponadto umożliwimy Ci jeszcze lepsze wykorzystanie rozwiązań oferowanych przez ten niezwykły i niezwykle dynamiczny język.

## 15.1. Wywołania zwrotne i „haki"

Użycie *wywołań zwrotnych* i *„haków"* to dość powszechna technika metaprogramowania. Metody te są wywoływane, gdy wystąpi konkretne zdarzenie w czasie działania programu Ruby. Może to być jedno z następujących zdarzeń:

- Wywoływanie dla obiektu nieistniejącej metody.
- Dołączanie modułu do klasy lub innego modułu.
- Rozszerzanie obiektu za pomocą modułu.
- Tworzenie podklasy dla klasy (przez dziedziczenie).
- Tworzenie odwołania do nieistniejącej stałej.
- Dodawanie metody instancji do klasy.
- Dodawanie do obiektu metody pojedynczego obiektu.

Dla każdego zdarzenia z powyższej listy możesz (jeśli uznasz to za konieczne) utworzyć metodę wywołania zwrotnego, która będzie wykonywana w momencie wystąpienia zdarzenia. Takie metody wywołania zwrotnego dotyczą poszczególnych obiektów lub klas, a nie są globalne. Aby metoda została wywołana w przypadku tworzenia podklasy dla klasy `Ticket`, musisz utworzyć odpowiednią metodę ściśle dla tej klasy.

Poniżej opisano każdy z tych „haków" zdarzeń środowiska wykonawczego. Przyjrzymy się im w kolejności wyszczególnienia na wcześniejszej liście.

### 15.1.1. Przechwytywanie nierozpoznanych komunikatów za pomocą metody method_missing

W rozdziale 4. (podrozdział 4.3) dość sporo napisano o metodzie `method_missing`. Oto podsumowanie: w momencie wysyłania komunikatu do obiektu wykonuje on pierwszą metodę z taką samą nazwą jak nazwa komunikatu, jaką znajdzie w swojej ścieżce wyszukiwania metod. Jeśli obiektowi nie uda się znaleźć żadnej takiej metody, zgłasza wyjątek `NoMethodError`, chyba że udostępniono obiekt z metodą o nazwie `method_missing` (powróć do podrozdziału 4.3, jeśli chcesz przypomnieć sobie, jak działa ta metoda).

Oczywiście metoda `method_missing` również w tym rozdziale zasługuje na swoje miejsce, ponieważ bez wątpienia jest najczęściej stosowanym „hakiem” środowiska wykonawczego w języku Ruby. Zamiast jednak powtarzać omówienie z rozdziału 4., przyjrzyjmy się kilku specyficznym niuansom tej metody. Rozważymy użycie metody `method_missing` w roli techniki delegowania. Dowiemy się, jak działa ta metoda i co się stanie w przypadku przesłonięcia jej na samej górze hierarchii klas.

#### DELEGOWANIE ZA POMOCĄ METODY METHOD_MISSING

Metoda `method_missing` umożliwia automatyczne rozszerzenie tego, jak zachowuje się obiekt. Dla przykładu załóżmy, że modelujesz obiekt, który pod pewnymi względami jest kontenerem, ale też wyróżnia się innymi właściwościami. Być może jest to książka kucharska. Chcesz mieć możliwość zdefiniowania dla niej kodu ustanawiającego ją kolekcją przepisów. Książka ma też mieć określone właściwości (tytuł, autor, być może lista osób, którym książkę udostępniono lub które brały udział w jej tworzeniu), które wymagają zapisania i obsługi niezależnie od przepisów. A zatem książka kucharska jest zarówno kolekcją, jak i repozytorium metadanych dotyczących kolekcji.

Aby zrealizować to w oparciu o metodę `method_missing`, należałoby zachować tablicę przepisów, a następnie przekazać do niej wszystkie nierozpoznane komunikaty. Prosta implementacja może wyglądać następująco:

```
class Cookbook
  attr_accessor :title, :author
  def initialize
    @recipes = []
  end
  def method_missing(m,*args,&block)
    @recipes.send(m,*args,&block)
  end
end
```

Możemy teraz wprowadzić modyfikacje w kolekcji przepisów, wykorzystując dowolne żądane metody tablicy. Załóżmy, że istnieje klasa `Recipe` niezależna od klasy `Cookbook`, a ponadto utworzono już kilka obiektów `Recipe`:

```
cb = Cookbook.new
cb << recipe_for_cake
cb << recipe_for_chicken
beef_dishes = cb.select {|recipes| recipe.main_ingredient == "beef" }
```

Instancja `cb` klasy `Cookbook` nie ma metod `<<` i `select`, dlatego komunikaty te są przekazywane tablicy `@recipes` dzięki metodzie `method_missing`. W dalszym ciągu możemy

zdefiniować dowolne żądane metody bezpośrednio w klasie `Cookbook`. Możliwe jest nawet przesłonięcie metod tablicy, jeśli zachowanie dowolnej z nich ma być bardziej powiązane z klasą `Cookbook`. Jednak metoda `method_missing` eliminuje konieczność definiowania równoległego zestawu metod do obsługi stron jako uporządkowanej kolekcji.

**Techniki delegowania metod w języku Ruby**

W przedstawionym przykładzie użycia metody `method_missing` przetwarzanie komunikatów (nieznanych) *oddelegowano* do tablicy `@recipes`. Język Ruby oferuje kilka mechanizmów delegowania działań między obiektami. Nie będziemy się tutaj nimi zajmować, ale przy dalszym poznawaniu języka możesz natrafić na klasy `Delegator` i `SimpleDelegator`.

Choć takie użycie metody `method_missing` jest bardzo proste (możliwe jest jednak uzupełnienie jej o wybrane dodatki z rozdziału 4.), zapewnia bardzo duże możliwości. W zamian za niewielki nakład pracy metoda dodaje do klasy sporo „inteligencji". Przyjrzyjmy się teraz drugiemu krańcowi spektrum, czyli metodzie `method_missing`, której nie ma w konkretnej klasie, lecz jest na samej górze drzewa klas i na najwyższym poziomie kodu.

### ORYGINAŁ: METODA BASICOBJECT#METHOD_MISSING

Metoda `method_missing` jest jedną z niewielu metod zdefiniowanych na samej górze drzewa klas w klasie `BasicObject`. Dzięki temu, że wszystkie klasy ostatecznie wywodzą się z klasy `BasicObject`, każdy obiekt dysponuje metodą `method_missing`.

W domyślnym wariancie metoda `method_missing` jest raczej inteligentna. Przyjrzyj się różnicy między komunikatami o błędzie w przypadku następujących dwóch wierszy w sesji narzędzia irb:

```
>> a
NameError: undefined local variable or method `a' for main:Object
>> a?
NoMethodError: undefined method `a?' for main:Object\
```

Nieznany identyfikator `a` może być metodą lub zmienną (w sytuacji, gdyby nie miał takiego statusu). Choć jest on obsługiwany przez metodę `method_missing`, komunikat o błędzie odzwierciedla to, że interpreter języka Ruby nie może ostatecznie stwierdzić, czy identyfikatora użyto jako wywołania metody, czy jako odwołania do zmiennej. Drugi nieznany identyfikator `a?` może być tylko metodą, ponieważ nazwy zmiennych nie mogą być zakończone pytajnikiem. Metoda `method_missing` stwierdza ten fakt i podaje właściwy komunikat o błędzie (zapewnia nawet wariant wyjątku, który zostanie zgłoszony).

Możliwe są dwa sposoby przesłonięcia domyślnego wariantu metody `method_missing`. Po pierwsze, możesz otworzyć klasę `BasicObject` i zmodyfikować definicję metody `method_missing`. Po drugie, możesz skorzystać z częściej używanej (choć nie aż tak często) techniki, która polega na zdefiniowaniu metody `method_missing` na najwyższym poziomie, co powoduje zainstalowanie jej jako prywatnej metody instancji klasy `Object`.

Jeśli skorzystasz z tej drugiej techniki, wszystkie obiekty, z wyjątkiem faktycznych instancji samej klasy `BasicObject`, znajdą nową wersję metody `method_missing`:

```
>> def method_missing(m,*args,&block)
>>   raise NameError, "Co właściwie masz na myśli, używając #{m}?"
>> end
=> nil
>> a
NameError: Co właściwie masz na myśli, używając a?
      from (irb):2:in `method_missing'
>> BasicObject.new.a
NoMethodError: undefined method `a' for #<BasicObject:0x4103ac>
```

Wywołanie metody `super` możesz umieścić wewnątrz nowej wersji metody `method_mis`
`↪sing`, jeśli zamiast zgłaszać wyjątek samemu, być może po zarejestrowaniu błędu, chcesz, aby zostało to obsłużone przez wersję metody `method_missing` w klasie `Basic`
`↪Object`.

Pamiętaj, że jeśli zdefiniujesz własną wersję metody `method_missing`, utracisz możliwość skorzystania z tego, że może ona odróżniać nazwy zmiennych od nazw metod:

```
>> a?
NameError: Co właściwie masz na myśli, używając a??
```

Prawdopodobnie nie ma to znaczenia, a zwłaszcza w sytuacji, gdy i tak zamierzasz wywołać metodę `super`. Jeżeli naprawdę tego oczekujesz, możesz sam sprawdzić szczegóły symbolu `m`. Jest to jednak interesujący wgląd w subtelności hierarchii klas i semantyki przesłaniania.

### METODY METHOD_MISSING, RESPOND_TO? I RESPOND_TO_MISSING?

Często przytaczanym problemem w przypadku metody `method_missing` jest to, że nie jest ona zgodna z metodą `respond_to?`. Przeanalizuj przykład. W klasie `Person` przechwytywane są komunikaty o nazwach zaczynających się łańcuchem `set_`, a następnie są one przekształcane w metody ustawiające: metoda `set_age(n)` staje się metodą `age=n` itd.

Oto przykład:

```
class Person
  attr_accessor :name, :age
  def initialize(name, age)
    @name, @age = name, age
  end
  def method_missing(m, *args, &block)
    if /set_(.*)/.match(m)
      self.send("#{$1}=", *args)
    else
      super
    end
  end
end
```

A zatem czy obiekt `Person` posiada metodę `set_age`, czy nie? Cóż, możesz wywołać tę metodę, ale obiekt `Person` utrzymuje, że nie udziela dla niej odpowiedzi:

```
person = Person.new("Dawid", 54)
person.set_age(55)              ←——— 55
p person.age
p person.respond_to?(:set_age)  ←——— wartość false
```

Sposób pozwalający zapewnić wzajemną zgodność metod `method_missing` i `respond_to?` polega na zdefiniowaniu specjalnej metody `respond_to_missing?`. Oto definicja, jaką możesz dodać do wcześniejszej klasy `Person`:

```
def respond_to_missing?(m, include_private = false)
    /set_/.match(m) || super
  end
```

Nowy obiekt `Person` będzie teraz udzielać różnych odpowiedzi dla tych samych zapytań:

```
55
true
```

Za pomocą drugiego argumentu metody `respond_to?` możesz kontrolować to, czy są dołączane metody prywatne. Argument ten będzie przekazywany metodzie `respond_to_missing?`. W przykładzie argument przyjmuje domyślnie wartość `false`.

Dodatkowo metody, które stają się widoczne dzięki metodzie `respond_to_missing?`, mogą być też przekształcane w obiekty metod za pomocą metody `method`:

```
person = Person.new("Dawid", 55)
p person.method(:set_age)     ←——— #<Method: Person#set_age>
```

Generalnie metoda `method_missing` jest bardzo przydatnym narzędziem wychwytującym zdarzenia. Zdecydowanie jednak nie jest jedynym.

### 15.1.2. Wychwytywanie operacji dołączania i wstawiania na początku

Wiesz, jak dołączyć moduł do klasy lub innego modułu, a także w jaki sposób umieścić moduł na początku klasy lub modułu. Aby wychwytywać takie zdarzenia (w celu wyzwalania wywołania zwrotnego w momencie występowania zdarzeń), możesz zdefiniować specjalne metody o nazwach `included` i `prepended`. Każda z tych metod otrzymuje jako swój pojedynczy argument nazwę modułu lub klasy, dla których wykonywana jest operacja dołączania lub umieszczania na początku.

Przyjrzyjmy się bliżej metodzie `included`, wiedząc, że metoda `prepended` działa w bardzo podobny sposób. Możesz przeprowadzić szybki test metody `included`, powodując wyświetlenie przez nią komunikatu, a następnie wykonanie operacji instrukcji `include`:

```
module M
  def self.included(c)
    puts "Właśnie dołączono mnie do klasy #{c}."
  end
end
class C
  include M
end
```

Wyświetlany jest komunikat `"Właśnie dołączono mnie do klasy C."` jako wynik wykonania metody `M.included`, gdy moduł `M` został dołączony do klasy `C` (ponieważ moduły mogą być też dołączane do innych modułów, przykład ten sprawdzi się również w sytuacji, gdyby `C` było modułem).

Kiedy przydatne byłoby dla modułu przechwytywanie w taki sposób własnego włączenia? Często omawiany przypadek dotyczy różnicy między metodami instancji

i metodami klasy. Gdy dołączasz moduł do klasy, zapewniasz, że wszystkie metody instancji zdefiniowane w module stają się dostępne dla instancji klasy. Nie dotyczy to jednak obiektu klasy. Pojawia się często następujące pytanie: co będzie, jeśli dodasz metody klasy do klasy przez dołączenie modułu oraz dodanie metod instancji?

Dzięki metodzie `included` możesz wychwycić operację instrukcji `include` i skorzystać z okazji, aby dodać metody klasy do klasy, która przeprowadza dołączanie. Poniższy listing 15.1 prezentuje przykładowy kod.

**Listing 15.1. Użycie metody included do dodania metody klasy w ramach operacji dołączania**

```
module M
  def self.included(cl)
    def cl.a_class_method
      puts "Klasa dysponuje teraz nową metodą klasy."
    end
  end
  def an_inst_method
    puts "Moduł ten zapewnia tę metodę instancji."
  end
end
class C
  include M
end
c = C.new
c.an_inst_method
C.a_class_method
```

Oto dane wyjściowe kodu z tego listingu:

```
Moduł ten zapewnia tę metodę instancji.
Klasa dysponuje teraz nową metodą klasy.
```

W momencie dołączania modułu `M` przez klasę `C` mają miejsce dwa zdarzenia. Po pierwsze, metoda instancji o nazwie `an_inst_method` pojawia się w ścieżce wyszukiwania instancji klasy (np. `c`). Po drugie, dzięki wywołaniu zwrotnemu metody `included` modułu `M` dla obiektu klasy `C` definiowana jest metoda klasy o nazwie `a_class_method`.

Metoda `Module#included` zapewnia przydatny sposób uzyskania dostępu do struktury klasy/modułu programu. Przyjrzyjmy się kolejnemu wywołaniu zwrotnemu z tego samego ogólnego obszaru zainteresowań, czyli metodzie `Module#extended`.

### 15.1.3. Przechwytywanie operacji rozszerzania

Jak wiesz z rozdziału 13., rozszerzanie poszczególnych obiektów za pomocą modułów to jedna z technik o największych możliwościach, jakie są dostępne w języku Ruby. Techniki te wykorzystują elastyczność obiektów oraz ich zdolność dostosowywania. Technika korzysta też z „haka” środowiska wykonawczego: używając metody `Module#extended`, możesz określić wywołanie zwrotne, które będzie wyzwalane każdorazowo, gdy obiekt wykonuje operację metody `extend` uwzględniającą dany moduł.

Listing 15.2 zawiera zmodyfikowaną wersję kodu z listingu 15.1, która ilustruje mechanizmy działania metody `Module#extended`.

**Listing 15.2. Wyzwalanie wywołania zwrotnego z poziomu zdarzenia metody extend**

```
module M
  def self.extended(obj)
    puts "Moduł #{self} jest używany przez #{obj}."
  end
  def an_inst_method
    puts "Moduł ten zapewnia tę metodę instancji."
  end
end
my_object = Object.new
my_object.extend(M)
my_object.an_inst_method
```

Oto dane wyjściowe kodu z tego listingu:

```
Moduł M jest używany przez #<Object:0x007f8e2a95bae0>.
Moduł ten zapewnia tę metodę instancji.
```

Warto przyjrzeć się zasadom działania wywołań zwrotnych `included` i `extended` w powiązaniu z klasami pojedynczych obiektów. Nie ma tutaj nic szczególnie zaskakującego. Dowiesz się jedynie, jaka jest spójność modelu klas i obiektów języka Ruby.

### ZACHOWANIE KLASY POJEDYNCZYCH OBIEKTÓW W PRZYPADKU WYWOŁAŃ ZWROTNYCH INCLUDED I EXTENDED

W rzeczywistości rozszerzanie obiektu za pomocą modułu odpowiada dołączaniu go do klasy pojedynczych obiektów danego obiektu. Niezależnie od tego, jakie z tych działań jest opisywane, ich rezultatem jest dodanie modułu do ścieżki wyszukiwania metod obiektu. Moduł jest umieszczany w ścieżce tuż za klasą pojedynczych obiektów danego obiektu.

Te dwie operacje wyzwalają jednak różne wywołania zwrotne: `included` i `extended`. Poniższy listing 15.3 demonstruje odpowiednie zachowania.

**Listing 15.3. Rozszerzanie obiektu i dołączanie go do jego klasy pojedynczych obiektów**

```
module M
  def self.included(c)                                          ❶
    puts "Moduł #{self} został dołączony przez klasę #{c}."
  end
  def self.extended(obj)                                        ❷
    puts "Moduł #{self} został rozszerzony przez obiekt #{obj}."
  end
end
obj = Object.new
puts "Dołączanie modułu M do klasy pojedynczych obiektów danego obiektu:"
class << obj                                                    ❸
  include M
end
puts
obj = Object.new
puts "Rozszerzanie obiektu za pomocą modułu M:"                 ❹
obj.extend(M)
```

Oba wywołania zwrotne, `included` ❶ i `extended` ❷, są zdefiniowane w module `M`. Każde wywołanie zwrotne wyświetla raport o tym, co ma miejsce. Zaczynając od zupełnie nowego, ogólnego obiektu, dołączamy moduł `M` do klasy pojedynczych obiektów danego obiektu ❸, a następnie powtarzamy proces, stosując kolejny nowy obiekt i rozszerzając go bezpośrednio za pomocą modułu `M` ❹.

Oto dane wyjściowe kodu z tego listingu:

```
Dołączanie modułu M do klasy pojedynczych obiektów danego obiektu:
Moduł M został dołączony przez klasę #<Class:#<Object:0x0000010193c978>>.
Rozszerzanie obiektu za pomocą modułu M:
Moduł M został rozszerzony przez obiekt #<Object:0x0000010193c310>.
```

Z całą pewnością operacja dołączenia wyzwala wywołanie zwrotne `included`, a operacje rozszerzenia wywołanie zwrotne `extended`, nawet pomimo tego, że w tym konkretnym scenariuszu wyniki tych dwóch operacji są takie same. Dla danego obiektu moduł `M` jest dodawany do ścieżki wyszukiwania metod obiektu. Jest to dobra ilustracja niektórych subtelności i precyzji architektury języka Ruby, a także przydatne przypomnienie tego, że bezpośrednie modyfikowanie klasy pojedynczych obiektów danego obiektu nie *do końca* odpowiada wykonywaniu operacji na poziomie pojedynczego obiektu wprost dla obiektu.

Tak jak moduły mogą przechwytywać operacje dołączania i rozszerzania, tak klasy mają możliwość określenia, kiedy są dla nich tworzone podklasy.

### 15.1.4. Przechwytywanie dziedziczenia za pomocą metody Class#inherited

Aby mieć możliwość tworzenia podklas klasy, należy zdefiniować dla niej specjalną metodę klasy o nazwie `inherited`. Jeśli została ona zdefiniowana dla danej klasy, w momencie tworzenia dla niej podklasy metoda ta jest wywoływana z nazwą nowej klasy jako jej pojedynczym argumentem.

Oto prosty przykład, w którym klasa `C` zgłasza fakt, że została dla niej utworzona podklasa:

```
class C
  def self.inherited(subclass)
    puts "Dla klasy #{self} została właśnie utworzona podklasa #{subclass}."
  end
end
class D < C
end
```

Tworzenie podklasy `D` klasy `C` automatycznie powoduje wyzwolenie wywołania metody `inherited` i wygenerowanie następujących danych wyjściowych:

```
Dla klasy C została właśnie utworzona podklasa D.
```

Metoda `inherited` to metoda klasy, dlatego potomkowie definiującej ją klasy również mają możliwość wywołania tej metody. Działania definiowane w kaskadzie metody `inherited` są następujące: jeśli dziedziczenie odbywa się z podklasy, wyzwala ona metodę `inherited`. Proces jest powtarzany na kolejnych niższych poziomach łańcucha dziedziczenia. W przypadku użycia następującego kodu:

```
class E < D
end
```

zostanie wyświetlony komunikat: `Dla klasy D została właśnie utworzona podklasa E`. Podobny rezultat zapewni tworzenie podklasy klasy `E` itd.

**Ograniczenia wywołania zwrotnego inherited**

Wszystko to ma swoje ograniczenia, w tym wywołanie zwrotne `inherited`. Gdy klasa `D` dziedziczy z klasy `C`, jest ona nadklasą klasy `D`. Dodatkowo jednak klasa pojedynczych obiektów klasy `C` jest nadklasą klasy pojedynczych obiektów klasy `D`. Dzięki temu klasa `D` ma możliwość wywołania metod klasy `C`. Nie jest jednak wyzwalane żadne wywołanie zwrotne. Jeśli nawet zdefiniujesz metodę `inherited` w klasie pojedynczych obiektów klasy `C`, nie zostanie ona wywołana.

Poniżej zaprezentowano środowisko testowe. Zauważ, jak metodę `inherited` zdefiniowano w obrębie klasy pojedynczych obiektów klasy `C`. Jednak nawet wtedy, gdy klasa `D` dziedziczy z klasy `C`, a nawet po jawnym utworzeniu klasy pojedynczych obiektów klasy `D`, wywołanie zwrotne nie jest wyzwalane:

```
class C
  class << self
    def self.inherited
      puts "Dokonano właśnie dziedziczenia klasy pojedynczych obiektów klasy C!"
      puts "Jednak komunikat ten nie zostanie nigdy wyświetlony."
    end
  end
end
class D < C
  class << self
    puts "Klasa pojedynczych obiektów klasy D teraz istnieje, ale bez wywołania
zwrotnego!"
  end
end
```

Oto dane wyjściowe tego kodu programu:

```
Klasa pojedynczych obiektów klasy D teraz istnieje, ale bez wywołania zwrotnego!
```

Choć wyjątkowo mało prawdopodobne jest spotkanie się kiedykolwiek z sytuacją, w której takie zachowanie będzie istotne, pozwala ono dokładnie zorientować się w sposobie współdziałania modelu klas języka Ruby z jego warstwą wywołań zwrotnych.

Przyjrzyjmy się teraz temu, jak przechwycić odwołanie do nieistniejącej stałej.

### *15.1.5. Metoda Module#const_missing*

`Module#const_missing` to kolejne powszechnie używane wywołanie zwrotne. Jak nazwa wskazuje, metoda ta jest wywoływana każdorazowo, gdy w obrębie danego modułu lub klasy występuje odwołanie do niemożliwej do zidentyfikowania stałej:

```
class C
  def self.const_missing(const)
    puts "Stała #{const} nie jest zdefiniowana. Ustawiana jest dla niej wartość 1."
    const_set(const,1)
  end
end
puts C::A
puts C::A
```

Oto dane wyjściowe tego kodu:

```
Stała A nie jest zdefiniowana. Ustawiana jest dla niej wartość 1.
1
1
```

Dzięki wywołaniu zwrotnemu stała `C::A` jest definiowana automatycznie, gdy zostanie użyta bez utworzenia dla niej definicji. Jest to realizowane w ten sposób, że instrukcja `puts` może wyświetlić wartość stałej. Instrukcja nigdy nie musi dysponować informacją o tym, że stała nie została wcześniej zdefiniowana. W drugim wywołaniu instrukcji `puts` stała jest już zdefiniowana, a metoda `const_missing` nie jest wywoływana.

Jednym z narzędzi wywołań zwrotnych zdarzeń w języku Ruby oferujących największe możliwości jest metoda `method_added`, która umożliwia wyzwalanie zdarzenia w momencie definiowania nowej metody instancji.

### 15.1.6. Metody method_added i singleton_method_added

Jeśli zdefiniujesz metodę `method_added` jako metodę klasy w dowolnej klasie lub module, będzie ona wywoływana podczas definiowania dowolnej metody instancji. Oto prosty przykład:

```
class C
  def self.method_added(m)      <------ Definiuje wywołanie zwrotne
    puts "Zdefiniowano właśnie metodę #{m}."
  end
  def a_new_method              <------ Wyzwalane przez definiowanie metody instancji
  end
end
```

Oto dane wyjściowe kodu tego programu:

```
Zdefiniowano właśnie metodę a_new_method.
```

Wywołanie zwrotne `singleton_method_added` realizuje bardzo podobne działanie, ale w przypadku metod pojedynczego obiektu. Być może zaskakujące będzie to, że wywołanie to wyzwala nawet samo siebie. Jeśli uruchomisz następujący fragment kodu:

```
class C
  def self.singleton_method_added(m)
    puts "Zdefiniowano właśnie metodę #{m}."
  end
end
```

okaże się, że wywołanie zwrotne, które jest metodą pojedynczego obiektu dla obiektu klasy `C`, powoduje wykonanie samego siebie:

```
Zdefiniowano właśnie metodę singleton_method_added.
```

Wywołanie zwrotne będzie również wyzwalane przez definicję innej metody pojedynczego obiektu (klasy). Rozszerzmy poprzedni przykład w celu dołączenia następującej definicji:

```
class C
  def self.singleton_method_added(m)
    puts "Zdefiniowano właśnie metodę #{m}."
  end
```

```
  def self.new_class_method
  end
end
```

Oto nowe dane wyjściowe:

```
Zdefiniowano właśnie metodę singleton_method_added.
Zdefiniowano właśnie metodę new_class_method.
```

W większości przypadków należy użyć metody `singleton_method_added` z obiektami innymi niż obiekty klasy. W przypadku ogólnego obiektu metoda ta może zostać zastosowana w następujący sposób:

```
obj = Object.new
def obj.singleton_method_added(m)
  puts "Zdefiniowano właśnie metodę pojedynczego obiektu #{m}."
end
def obj.a_new_singleton_method
end
```

W tym przypadku dane wyjściowe są następujące:

```
Zdefiniowano właśnie metodę pojedynczego obiektu singleton_method_added.
Zdefiniowano właśnie metodę pojedynczego obiektu a_new_singleton_method.
```

Ponownie masz do czynienia z trochę zaskakującym efektem polegającym na tym, że definiowanie metody `singleton_method_added` wyzwala wykonanie samego wywołania zwrotnego.

Połączenie ze sobą rozwiązań opartych na klasach i obiektach pozwala osiągnąć efekt specyficzny dla obiektów przez zdefiniowanie odpowiednich metod w klasie pojedynczych obiektów danego obiektu:

```
obj = Object.new
class << obj
  def singleton_method_added(m)
    puts "Zdefiniowano właśnie metodę pojedynczego obiektu #{m}."
  end
  def a_new_singleton_method
  end
end
```

Dane wyjściowe dla tego fragmentu kodu są dokładnie takie same jak w przypadku poprzedniego przykładu. Ostatecznie, wracając do punktu wyjścia, możesz zdefiniować metodę `singleton_method_added` jako zwykłą metodę instancji klasy. W tym przypadku każda instancja tej klasy będzie zgodna z regułą określającą, że wywołanie zwrotne będzie wyzwalane przez tworzenie metody pojedynczego obiektu:

```
class C
  def singleton_method_added(m)  <------- ❶
    puts "Zdefiniowano właśnie metodę pojedynczego obiektu #{m}."
  end
end
c = C.new
def c.a_singleton_method         <------- ❷
end
```

W tym przypadku definicja wywołania zwrotnego ❶ zarządza każdą instancją klasy `C`. A zatem definicja metody pojedynczego obiektu w takiej instancji ❷ powoduje wyzwolenie wywołania zwrotnego, generując następujące dane wyjściowe:

```
Zdefiniowano właśnie metodę pojedynczego obiektu a_singleton_method.
```

Możliwe jest, że w tworzonych aplikacjach Ruby nie będziesz często używać metody `method_added` ani metody `singleton_method_added`. Jednak eksperymentowanie z nimi stanowi znakomity sposób na dokładniejsze zorientowanie się w tym, jak dopasowują się do siebie różne „fragmenty obrazów" powiązane z klasami, instancjami i klasami pojedynczych obiektów.

Zajmiemy się teraz zagadnieniem sprawdzania możliwości obiektów (metoda `"abc".methods` i powiązane z nią elementy). Ogólne informacje na ten temat umieszczono w rozdziale 7., w którym omówiono wbudowane elementy podstawowe. Jak w nim obiecano, w niniejszym rozdziale zagadnienie zostanie obszerniej zaprezentowane.

## 15.2. Interpretowanie zapytań dotyczących możliwości obiektów

Na tym etapie znajomości języka Ruby możesz zdecydować się na więcej działań związanych z listami metod obiektów niż tylko sprawdzanie ich i odrzucanie. W tym podrozdziale przyjrzymy się kilku przykładom (później będziesz mieć jeszcze wiele możliwości tworzenia kolejnych wraz ze zwiększającymi się wymaganiami i potrzebami) sposobów używania i interpretowania informacji zawartych na listach metod. Wiedza, jaką zdobyłeś o języku Ruby od czasu, gdy w książce ostatnio zajmowano się bezpośrednio tym zagadnieniem, okaże się bardzo przydatna. Dowiesz się też o kilku świetnych kwestiach związanych z samymi metodami, służących do odpytywania metod.

Zacznijmy od najbardziej znajomego punktu wyjścia, czyli wyświetlenia listy metod nieprywatnych za pomocą metody `methods`.

### 15.2.1. Wyświetlanie metod nieprywatnych obiektu

Aby wyświetlić metody nieprywatne (czyli publiczne lub chronione), które są znane obiektowi, używasz metody `methods` zwracającej tablicę symboli. W przypadku tablic możesz wykonać kilka przydatnych zapytań dotyczących wyników początkowego zapytania. Zwykle w pewien sposób będziesz filtrować tablicę, aby uzyskać podzbiór metod.

Oto na przykład sposób umożliwiający uzyskanie od łańcucha informacji o znanych mu metodach, który uwzględnia modyfikację instrukcji `case`:

```
>> string = "Łańcuch testowy"
=> "Łańcuch testowy"
>> string.methods.grep(/case/).sort
=> [:casecmp, :downcase, :downcase!, :swapcase, :swapcase!, :upcase,
:upcase!]
```

Metoda `grep` odfiltrowuje dowolny symbol, który nie zawiera metody `case` (pamiętaj o tym, że choć symbole nie są łańcuchami, cechują się kilkoma podobnymi do nich zachowaniami, takimi jak możliwość przetwarzania ich przez metodę `grep`). Umieszczone

na końcu polecenie `sort` jest przydatne w przypadku większości operacji wyświetlania listy metod. W tym przykładzie nie robi to większej różnicy, ponieważ występuje w nim tylko siedem metod. Gdy jednak zostaną użyte tablice liczące 100 lub więcej symboli, sortowanie ich może okazać się bardzo pomocne.

Uzyskanie wyniku w przypadku użycia polecenia `grep` dla łańcucha `case` zależy oczywiście od założenia, że metody powiązane z wielkością znaków będą w nazwie zawierać łańcuch `case`. W przypadku wielu zapytań dotyczących możliwości metod zdecydowanie występuje element oceny, któremu często towarzyszy kwestia opierania się na posiadanych informacjach odnośnie do tego, co uważasz, że znajdziesz. Jednak na ogół wszystko kończy się pomyślnie, ponieważ w kwestii doboru nazw metod język Ruby cechuje się wystarczającą spójnością i konwencjonalnością.

Niektóre metody z łańcuchem `case` w nazwie dysponują też wersją metody z wykrzyknikiem w nazwie. Podążając tym tropem, znajdźmy wszystkie metody z wykrzyknikiem w nazwie, które zawiera łańcuch, ponownie używając metody `grep`:

```
>>string.methods.grep(/.!/).sort
=> [:capitalize!, :chomp!, :chop!, :delete!, :downcase!, :encode!, :gsub!, :lstrip!, :next!,
↪:reverse!, :rstrip!, :scrub!, :slice!, :squeeze!, :strip!, :sub!, :succ!, :swapcase!,
↪:tr!, :tr_s!, :upcase!]
```

Dlaczego w wyrażeniu regularnym przed znakiem `!` znajduje się kropka? Jej celem jest zapewnienie, że w nazwie metody przed wykrzyknikiem występuje co najmniej jeden znak. Oznacza to wykluczenie metod `!`, `!=` i `!~`, które zawierają w nazwie wykrzyknik, ale zwyczajowo nie są zaliczane do grupy metod z wykrzyknikiem w nazwie. Wymagamy metod z nazwą zakończoną znakiem `!`, ale nie tych, które zaczynają się od niego.

Użyjmy metody `methods` w trochę szerszym zakresie. Oto pytanie, na jakie możemy udzielić odpowiedzi, interpretując wyniki zapytania dotyczącego metod: czy łańcuchy dysponują jakimikolwiek metodami z wykrzyknikiem w nazwie, które nie mają odpowiedników w postaci metod bez znaku `!` w nazwie?

```
string = "Łańcuch testowy"
methods = string.methods
bangs = string.methods.grep(/.!/)                                  ❶
unmatched = bangs.reject do |b|
  methods.include?(b[0..-2].to_sym)                                ❷
end
if unmatched.empty?
  puts "Wszystkie metody z wykrzyknikiem w nazwie są dopasowywane
  ↪przez metody bez tego znaku w nazwie."                          ❸
else
  puts "Niektóre metody z wykrzyknikiem w nazwie nie mają odpowiednika w postaci metody bez
tego znaku w nazwie: "
  puts unmatched
end
```

**Dane wyjściowe: Wszystkie metody z wykrzyknikiem w nazwie są dopasowywane przez metody bez tego znaku w nazwie**

Działanie kodu polega na gromadzeniu wszystkich metod publicznych łańcucha, a także osobno wszystkich jego metod z wykrzyknikiem w nazwie ❶. Następnie operacja metody `reject` odfiltrowuje wszystkie nazwy metod z wykrzyknikiem, dla których na większej liście nazw metod ❷ można znaleźć odpowiadającą im nazwę metody bez

znaku !. Indeks [0..-2] pobiera wszystko, z wyjątkiem ostatniego znaku symbolu. Inaczej mówiąc, jest to nazwa metody pozbawiona znaku!. Wywołanie metody to_sym powoduje przekształcenie wynikowego łańcucha z powrotem w symbol, aby metoda include? wykonująca test mogła go znaleźć w tablicy metod. Jeśli filtrowana lista jest pusta, oznacza to, że nie znaleziono żadnych niedopasowanych nazw metod z wykrzyknikiem. Jeśli lista nie jest pusta, została znaleziona co najmniej jedna taka nazwa, która może zostać wyświetlona ❸.

Jeśli uruchomisz skrypt w takiej postaci, zawsze pobierze pierwszą (dla wartości prawda) gałąź instrukcji if. Aby wyświetlić listę niedopasowanych metod z wykrzyknikiem w nazwie, do kodu programu możesz dodać następujący wiersz (tuż za pierwszym wierszem):

```
def string.surprise!; end
```

Po uruchomieniu zmodyfikowanej wersji skryptu ujrzysz następujące dane wyjściowe:

```
Niektóre metody z wykrzyknikiem w nazwie nie mają odpowiednika w postaci metody bez tego
➥znaku w nazwie:
surprise!
```

Jak już pokazano, tworzenie metod z wykrzyknikiem w nazwie bez towarzyszących im metod bez wykrzyknika w nazwie to zazwyczaj zła praktyka. Jest to jednak dobry sposób na sprawdzenie działania metody methods.

Oczywiście możesz zażądać od obiektów klas i modułów informacji o ich metodach. Są to w końcu tylko obiekty. Pamiętaj jednak, że metoda methods zawsze wyświetla listę metod nieprywatnych samego obiektu. W przypadku klas i modułów oznacza to, że nie otrzymujesz listy metod, które mogą wywołać instancje klasy lub instancje klas dołączające moduł. Uzyskujesz metody znane samej klasie lub modułowi. Oto (częściowy) wynik wywołania metody methods dla nowo utworzonego obiektu klasy:

```
>> class C; end
=> nil
>> C.methods.sort
=> [:!, :!=, :!~, :<, :<=, :<=>, :==, :===, :=~, :>, :>=, :__id__, :__send__,
:allocate, :ancestors, :autoload, :autoload?, :class, :class_eval,
:class_exec, :class_variable_defined?, :class_variable_get,
:class_variable_set, :class_variables, itd.
```

Obiekty klas i modułów współużytkują niektóre metody ze swoimi instancjami, ponieważ są to obiekty, które generalnie współużytkują określone metody. Jednak widoczne metody są tymi, które może wywołać sama klasa lub moduł. Możesz również zażądać od klas i modułów informacji o definiowanych przez nie metodach instancji. Wkrótce powrócimy do tej techniki. Najpierw jednak przyjrzymy się w skrócie procesowi wyświetlania listy metod prywatnych i chronionych obiektu.

### 15.2.2. Wyświetlanie listy metod prywatnych i chronionych

Każdy obiekt (z wyjątkiem instancji klasy BasicObject) zawiera metody private_methods i protected_methods. Ich działanie jest zgodne z oczekiwaniami: zapewniają one tablice symboli, które jednak przechowują odpowiednio nazwy metod prywatnych i chronionych.

Zupełnie nowe obiekty języka Ruby dysponują wieloma metodami prywatnymi, ale nie dysponują żadną metodą chronioną:

```
$ ruby -e 'o = Object.new; p o.private_methods.size'
72
$ ruby -e 'o = Object.new; p o.protected_methods.size'
0
```

Czym są metody prywatne? Są to prywatne metody instancji zdefiniowane głównie w module `Kernel`, a dodatkowo w klasie `BasicObject`. Możesz to sprawdzić w następujący sposób:

```
$ ruby -e 'o = Object.new; p o.private_methods -
BasicObject.private_instance_methods(false) -
Kernel.private_instance_methods(false)'
[]
```

Zauważ, że po wyłączeniu metod prywatnych zdefiniowanych w module `Kernel` i klasie `BasicObject` oryginalny obiekt nie ma żadnych metod prywatnych do wyświetlenia. Metody prywatne zdefiniowane w module `Kernel` są metodami traktowanymi jako „najwyższego poziomu" (np. `puts`, `binding` i `raise`). Poeksperymentuj trochę z prezentowanymi tutaj technikami wyświetlania listy metod, a niektóre z tych metod okażą się znajome.

Oczywiście jeśli sam zdefiniujesz metodę prywatną, również pojawi się ona na liście metod prywatnych. Oto przykład: prosta klasa `Person`, w której przypisanie imienia osobie za pośrednictwem metody `name=` powoduje wyzwolenie metody normalizującej imiona. Metoda ta usuwa z imienia wszystko z wyjątkiem liter i wybranych znaków interpunkcji. Metoda `normalize_name` jest metodą prywatną:

```
class Person
  attr_reader :name
  def name=(name)                    ← Definiuje niedomyślny akcesor zapisu
    @name = name
    normalize_name                   ← Normalizuje przypisane imię
  end
  private
  def normalize_name
    name.gsub!(/[^-a-z'.\s]/i, "")   ← Usuwa z imienia niepożądane znaki
  end
end
david = Person.new
david.name = "123Dawid!! Now%a9k"
raise "Problem" unless david.name == "Dawid Nowak"   ← Sprawdza, czy normalizacja działa poprawnie
puts "Imię poddano normalizacji."    ← Wyświetla komunikat powodzenia
p david.private_methods.sort.grep(/normal/)   ← Wynik inspekcji metody prywatnej: [:normalize_name]
```

Metody chronione mogą być sprawdzane w bardzo podobny sposób przy użyciu metody `protected_methods`.

Oprócz żądania od obiektów informacji o znanych im metodach często przydatne jest uzyskiwanie od klas i modułów informacji dotyczących zapewnianych przez nie metod.

### 15.2.3. Uzyskiwanie metod instancji klas i modułów

Klasy i moduły są wyposażone we w pewnym stopniu „podrasowany" zestaw metod służących do odpytywania metod. Sprawdzenie takich metod dostępnych w klasie `String` zapewnia ich kompletną listę. **Pogrubieniem** wyróżniono metody specyficzne dla klas i modułów:

```
>> String.methods.grep(/methods/).sort
=> [:instance_methods, :methods, :private_instance_methods,
:private_methods,:protected_instance_methods, :protected_methods,
:public_instance_methods,:public_methods, :singleton_methods]
```

Pogrubione nazwy metod zapewniają różnego rodzaju listy metod instancji zdefiniowanych w klasie lub module. Cztery metody działają w następujący sposób:

- Metoda `instance_methods` zwraca wszystkie publiczne i chronione metody instancji.
- Metoda `public_instance_methods` zwraca wszystkie publiczne metody instancji.
- Metody `protected_instance_methods` i `private_instance_methods` zwracają odpowiednio wszystkie chronione i prywatne metody instancji.

Podczas wywoływania dowolnej z tych metod masz możliwość przekazania argumentu. Jeśli przekażesz argument `false`, otrzymana w odpowiedzi lista metod będzie uwzględniać tylko te zdefiniowane w odpytywanej klasie lub module. W przypadku przekazania dowolnego argumentu z wartością prawdy boolowskiej (dowolna inna niż wartość `false` lub `nil`) lub w sytuacji, gdy nie przekazano żadnego argumentu, lista metod będzie obejmować metody zdefiniowane w odpytywanej klasie lub module oraz wszystkich ich klasach i modułach potomnych.

Na przykład w następujący sposób możesz stwierdzić, jakie metody instancji zdefiniowano w klasie `Range`:

```
>> Range.instance_methods(false).sort
=> [:==, :===, :begin, :bsearch, :cover?, :each, :end, :eql?, :exclude_end?,
:first, :hash, :include?, :inspect, :last, :max, :member?, :min, :size,
:step, :to_s]]
```

Idąc krok dalej, co będzie, jeśli wymagasz informacji o tym, które z metod zdefiniowanych w module `Enumerable` są przesłaniane w klasie `Range`? W celu uzyskania takiej informacji możesz wykonać operację z użyciem operatora `&` dla dwóch list metod instancji zdefiniowanych w module `Enumerable` i klasie `Range`:

```
>> Range.instance_methods(false) & Enumerable.instance_methods(false)
=> [:first, :min, :max, :member?, :include?]
```

Jak widać, klasa `Range` ponownie definiuje pięć metod, które zostały już zdefiniowane w module `Enumerable`.

Przyjrzymy się wkrótce ostatniej z metod zawierających łańcuch `methods` w nazwie, czyli metodzie `singleton_methods`.

Stwórzmy jednak najpierw program, który generuje listę wszystkich przesłonięć wszystkich klas dołączających moduł `Enumerable`.

### UZYSKIWANIE WSZYSTKICH PRZESŁONIĘĆ MODUŁU ENUMERABLE

Przedstawiona tutaj strategia będzie polegać na stwierdzeniu, jakie klasy dołączają moduł `Enumerable`, a następnie wykonaniu, tak jak w poprzednim przykładzie, dla każdej z nich operacji opartej na operatorze `&`. Wyniki zostaną zapisane i ostatecznie wyświetlone. Poniższy listing 15.4 prezentuje kod.

**Listing 15.4. Przesłonięcia metod instancji modułu Enumerable dla jego obiektów potomnych**

```
overrides = {}                                                    ❶
enum_classes = ObjectSpace.each_object(Class).select do |c|       ❷
  c.ancestors.include?(Enumerable)
end
enum_classes.sort_by {|c| c.name}.each do |c|
  overrides[c] = c.instance_methods(false) &                      ❸
                 Enumerable.instance_methods(false)
end
overrides.delete_if {|c, methods| methods.empty? }                ❹
overrides.each do |c,methods|                                     ❺
  puts "Klasa #{c} przesłania: #{methods.join(", ")}"
end
```

Najpierw tworzymy pustą tablicę asocjacyjną w zmiennej `overrides` ❶. Następnie uzyskujemy listę wszystkich klas dołączających moduł `Enumerable`. Technika pobierania tej listy uwzględnia moduł `ObjectSpace` i jego metodę `each_object` ❷. Metoda ta pobiera pojedynczy argument reprezentujący klasę obiektów do znalezienia. W tym przypadku interesują nas obiekty klasy `Class`, i to tylko te, które wśród swoich obiektów potomnych mają moduł `Enumerable`. Metoda `each_object` zwraca enumerator. Wywołanie metody `select` dla tego enumeratora powoduje pożądany efekt w postaci odfiltrowania listy wszystkich klas wyłącznie do tych, które dołączyły moduł `Enumerable`.

Pora wypełnić tablicę asocjacyjną zmiennej `overrides`. Dla każdej klasy w obiekcie `enum_classes` (ładnie posortowanym według nazwy klasy) w tablicy tej umieszczamy wpis. Kluczem jest klasa, a wartością tablica nazw metody modułu `Enumerable`, które są przesłaniane przez tę klasę ❸. Po usunięciu wszystkich wpisów reprezentujących klasy, które nie zostały przesłonięte przez żadne metody tego modułu ❹, wyświetlane są wyniki. Aby zapewnić spójność i przejrzystość danych wyjściowych ❺, w następujący sposób wykonywane są operacje oparte na metodach `sort` i `join`:

```
Class ARGF.class overrides: to_a
Class Array overrides: to_a, to_h, first, reverse_each, find_index, sort,
collect, map, select, reject, zip, include?, count, cycle, take,
take_while, drop, drop_while
Class Enumerator overrides: each_with_index, each_with_object
Class Enumerator::Lazy overrides: map, collect, flat_map, collect_concat,
select, find_all, reject, grep, zip, take, take_while, drop, drop_while,
lazy, chunk, slice_before
Class Hash overrides: to_h, to_a, select, reject, include?, member?
Class ObjectSpace::WeakMap overrides: include?, member?
Class Range overrides: first, min, max, member?, include?
Class Struct overrides: to_a, to_h, select
```

Pierwszy wiersz odnosi się do trochę nietypowego obiektu w postaci `ARGF.class`, który jest unikatowym, specjalnie skonstruowanym obiektem zaangażowanym w przetwarzanie danych wejściowych programu. Inne wiersze dotyczą kilku znajomych klas dołączających moduł `Enumerable`. W każdym przypadku widoczne jest, jakie metody tego modułu zostały przesłonięte przez daną klasę.

Przyjrzyjmy się temu, jak odpytywać obiekt w odniesieniu do jego metod pojedynczego obiektu.

### 15.2.4. Wyświetlanie listy metod pojedynczego obiektu danego obiektu

Jak wiesz, metoda pojedynczego obiektu to metoda definiowana na wyłączne potrzeby określonego obiektu (a jeśli obiektem jest klasa — w celu użycia przez obiekt i jego podklasy) i przechowywana w klasie pojedynczych obiektów tego obiektu. Metoda `singleton_methods` umożliwia wyświetlenie listy wszystkich takich metod. Zauważ, że metoda ta wyszczególnia publiczne i chronione metody pojedynczego obiektu, lecz nie prywatne. Oto przykład:

```
class C
end
c = C.new          ❶
class << c         ❷
  def x
  end
  def y
  end
  def z            ❸
  end
  protected :y
  private :z
end
p c.singleton_methods.sort   ❹
```

Tworzona jest instancja klasy `C` ❶ oraz otwierana jej klasa pojedynczych obiektów ❷. W tej klasie są definiowane trzy metody, po jednej na poziomie publicznym (`x`), chronionym (`y`) i prywatnym (`z`) ❸. Po wyświetleniu dane wyjściowe metod pojedynczego obiektu dla obiektu `c` ❹ mają następującą postać:

```
[:x, :y]
```

Metody pojedynczego obiektu są też traktowane jak zwykłe metody. Metody `:x` i `:y` zostaną wyświetlone również w przypadku wywołania metody `c.methods`. Dla klasy pojedynczych obiektów możesz użyć opartych na klasie metod odpytujących metody. Na końcu poprzedniego przykładowego kodu dodaj następujący kod:

```
class << c
  p private_instance_methods(false)
end
```

Po uruchomieniu tego kodu ujrzysz następujące dane wyjściowe:

```
[:z]
```

Metoda `:z` jest metodą pojedynczego obiektu dla obiektu `c`. Oznacza to, że jest to metoda instancji (w tym przypadku w wariancie prywatnym) klasy pojedynczych obiektów dla obiektu `c`.

Od klasy możesz zażądać informacji o jej metodach pojedynczego obiektu. W odpowiedzi uzyskasz tego rodzaju metody zdefiniowane dla tej klasy i wszystkich jej obiektów nadrzędnych. Oto odpowiednia ilustracja bazująca na sesji narzędzia irb:

```
>> class C; end
=> nil
>> class D < C; end
=> nil
>> def C.a_class_method_on_C; end
=> nil
>> def D.a_class_method_on_D; end
=> nil
>> D.singleton_methods
=> [:a_class_method_on_D, :a_class_method_on_C]
```

Po nabraniu wprawy w posługiwaniu się różnymi metodami zawierającymi łańcuch `methods` w nazwie stwierdzisz, że przydają się one podczas analizowania i badania tego, jak i gdzie metody są definiowane. Na przykład za pomocą zapytań dotyczących metod możesz sprawdzić, w jaki sposób tworzone są metody klasy `File`. Na początek dowiedz się, jakie metody klasy są dziedziczone przez klasę `File` z jej obiektów nadrzędnych (w przeciwieństwie do tych, które są definiowane przez samą klasę):

```
>> File.singleton_methods - File.singleton_methods(false)
=> [:new, :open, :sysopen, :for_fd, :popen, :foreach, :readlines,
:read, :select, :pipe, :try_convert, :copy_stream]
```

Wywołanie metody `singleton_methods(false)` zapewnia tylko metody pojedynczego obiektu zdefiniowane w klasie `File`. Wywołanie bez argumentu `false` pozwala uzyskać wszystkie metody pojedynczego obiektu zdefiniowane w klasie `File` oraz jej obiektach nadrzędnych. Różnicą są metody zdefiniowane przez obiekty nadrzędne.

Nadklasą klasy `File` jest klasa `IO`. Interesujące, choć nie zaskakujące, jest to, że każdą z dwunastu metod pojedynczego obiektu obiektów nadrzędnych dostępnych dla klasy `File` zdefiniowano w klasie `IO`. Możesz to potwierdzić za pomocą kolejnego zapytania:

```
>> IO.singleton_methods(false)
=> [:new, :open, :sysopen, :for_fd, :popen, :foreach, :readlines,
:read, :select, :pipe, :try_convert, :copy_stream]
```

Relacja między klasami — w tym przypadku dotyczy to faktu, że klasa `File` jest podklasą klasy `IO`, a zatem współużytkuje jej metody pojedynczego obiektu (metody klasy) — jest wprost widoczna w tablicach nazw metody. Różne metody zawierające w nazwie łańcuch `methods` oferują prawie nieograniczone możliwości tego rodzaju inspekcji i eksploracji.

Jak widać, rozwiązania języka Ruby służące do odpytywania metod są w stanie zapewnić sporo informacji o obsługiwanych obiektach, klasie i modułach. Musisz jedynie wyciągnąć wnioski, stosując techniki przetwarzania tekstu i odpytywania kolekcji dla udostępnianych przez nie list. Interpretowanie zapytań dotyczących metod stanowi

dobry przykład zapewnianego przez język Ruby wariantu pętli sprzężenia zwrotnego procesu uczenia: im bardziej poznajesz język, tym lepiej *możesz* się go nauczyć.

W dalszej kolejności zajmiemy się kwestią mechanizmu refleksji w czasie wykonywania kodu w przypadku zmiennych i stałych.

## 15.3. Introspekcja zmiennych i stałych

Język Ruby pozwala uzyskać kilka informacji o tym, do jakich zmiennych i stałych masz dostęp w danym momencie wykonywania kodu. Możliwe jest otrzymanie zestawienia zmiennych lokalnych lub globalnych, zmiennych instancji obiektu, zmiennych klasy lub modułu oraz stałych klasy lub modułu.

### 15.3.1. Wyświetlanie listy zmiennych lokalnych lub globalnych

Inspekcje zmiennych lokalnych lub globalnych są proste do przeprowadzenia: używane są metody najwyższego poziomu `local_variables` i `global_variables`. W każdym przypadku w odpowiedzi uzyskujesz tablicę symboli odpowiadających aktualnie zdefiniowanym zmiennym lokalnym lub globalnym:

```
x = 1
p local_variables
[:x]
p global_variables.sort
[:$!, :$", :$$, :$&, :$', :$*, :$+, :$,, :$-0, :$-F, :$-I, :$-K, :$-W, :$-a,
:$-d, :$-i, :$-l, :$-p, :$-v, :$-w, :$., :$/, :$0, :$1, :$2, :$3, :$4, :$5,
:$6, :$7, :$8, :$9, :$:, :$;, :$<, :$=, :$>, :$?, :$@, :$DEBUG, :$FILENAME,
:$KCODE, :$LOADED_FEATURES, :$LOAD_PATH, :$PROGRAM_NAME, :$SAFE, :$VERBOSE,
:$\, :$_, :$`, :$stderr, :$stdin, :$stdout, :$~]
```

Lista zmiennych globalnych obejmuje takie zmienne jak: `$:` (ścieżka ładowania bibliotek, która jest dostępna też jako zmienna `$LOAD_PATH`), `$~` (obiekt globalny `MatchData` oparty na najbardziej aktualnej operacji dopasowywania wzorca), `$0` (nazwa pliku, w którym zainicjowano wykonywanie bieżącego programu), `$FILENAME` (nazwa aktualnie uruchomionego pliku) oraz inne zmienne. Lista zmiennych lokalnych zawiera wszystkie aktualnie zdefiniowane zmienne lokalne.

Zauważ, że metody `local_variables` i `global_variables` nie zapewniają wartości zmiennych, o których informują. Podają one jedynie nazwy. To samo dotyczy metody `instance_variables`, którą możesz wywołać dla dowolnego obiektu.

### 15.3.2. Wyświetlanie listy zmiennych instancji

Oto kolejna wersja prostej klasy `Person`, która ilustruje, co ma miejsce w operacji odpytywania dotyczącej zmiennych instancji:

```
class Person
  attr_accessor :name, :age
  def initialize(name)
    @name = name
  end
end
```

```
david = Person.new("Dawid")
david.age = 55
p david.instance_variables
```

Oto dane wyjściowe:

```
[:@name, :@age]
```

Obiekt `david` ma dwie zmienne instancji zainicjowane w momencie wykonywania zapytania. Jednej z nich, o nazwie `@name`, przypisano wartość w chwili tworzenia obiektu. Druga zmienna `@age` istnieje dzięki atrybutowi akcesora `age`. Atrybuty są implementowane jako metody odczytu i/lub zapisu na bazie zmiennych instancji. Z tego powodu nawet pomimo tego, że zmienna `@age` nie pojawia się jawnie nigdzie w programie, jest inicjowana po przypisaniu obiektowi wieku.

Nazwy wszystkich zmiennych instancji zaczynają się od znaku `@`, a zmiennych globalnych od znaku `$`. Możesz oczekiwać, że interpreter języka Ruby nie będzie zwracać uwagi na te znaki, udostępniając listy nazw zmiennych. Nazwy pojawiające się na listach uwzględniają jednak te znaki początkowe.

**Zmienna w postaci znaku podkreślenia w narzędziu irb**

Jeśli w nowej sesji narzędzia irb uruchomisz metodę `local_variables`, zostanie wyświetlony znak podkreślenia:

```
>> local_variables
=> [: ]
```

Znak podkreślenia identyfikuje specjalną zmienną narzędzia irb: zmienna reprezentuje wartość ostatniego wyrażenia, dla którego narzędzie określiło wartość. Zmiennej tej możesz użyć do pobrania wartości, które w przeciwnym razie nie będą dostępne:

```
>> Person.new("Dawid")
=> #<Person:0x000001018ba360 @name="Dawid">
>> david = _
=> #<Person:0x000001018ba360 @name="Dawid">
```

Obecnie obiekt `Person` jest powiązany ze zmienną `david`.

Przyjrzyjmy się teraz technikom śledzenia przebiegu wykonywania, które ułatwiają ustalanie historii wywoływania metod w danym miejscu wykonywanego kodu.

## 15.4. Śledzenie wykonywania kodu

Niezależnie od tego, na jakim etapie jest wykonywanie kodu programu, w jakiś sposób został on osiągnięty. Może to być najwyższy poziom w hierarchii kodu lub miejsce poprzedzone jednym lub większą liczbą wywołań metody. Język Ruby zapewnia informacje o tym, jak osiągnięto dane miejsce. Podstawowym narzędziem służącym do sprawdzania historii wywołań metod jest metoda najwyższego poziomu `caller`.

### 15.4.1. Sprawdzanie danych śledzenia stosu za pomocą metody caller

Metoda `caller` zapewnia tablicę łańcuchów. Każdy z nich reprezentuje jeden krok śledzenia stosu: jest to opis pojedynczego wywołania metody oraz przebiegu działań do bieżącego miejsca w wykonywanym kodzie. Łańcuchy zawierają informacje o pliku

lub programie, w którym wywołanie metody miało miejsce, wierszu identyfikującym miejsce wystąpienia tego wywołania oraz metodzie (jeśli taka istnieje) inicjującej wywołanie bieżącej metody.

Poniżej zamieszczono przykład. Następujące wiersze umieść w pliku o nazwie *tracedemo.rb*:

```
def x
  y
end
def y
  z
end
def z
  puts "Dane śledzenia stosu: "
  p caller
end
x
```

Działanie tego programu sprowadza się do wniknięcia w stos wywołań metody: metoda `x` wywołuje metodę `y`, która wywołuje metodę `z`. Wewnątrz metody `z` uzyskujemy dane śledzenia stosu dzięki metodzie `caller`. Oto dane wyjściowe uzyskane po uruchomieniu pliku *tracedemo.rb*:

```
Stacktrace:
["tracedemo.rb:6:in `y'", "tracedemo.rb:2:in `x'", "tracedemo.rb:14:in
`<main>'"]
```

Każdy łańcuch w tablicy danych śledzenia stosu zawiera jeden odnośnik w łańcuchu wywołań metody, który kieruje nas do miejsca wywołania metody `caller`. Pierwszy łańcuch reprezentuje najnowsze wywołanie w historii: znaleźliśmy się w wierszu 6. kodu pliku *tracedemo.rb* wewnątrz metody `y`. Drugi łańcuch pokazuje, że metodę `y` osiągnięto za pośrednictwem metody `x`. Trzeci i ostatni łańcuch potwierdza napotkanie nazwy metody `<main>`, co oznacza, że wywołanie metody `x` zostało wykonane na najwyższym poziomie, a nie w obrębie metody.

Składnię danych śledzenia stosu możesz rozpoznać na podstawie komunikatów widocznych w błędach krytycznych. Jeśli przebudujesz metodę `z` do następującej postaci:

```
def z
  raise
end
```

dane wyjściowe będą następujące:

```
tracedemo.rb:10:in `z': unhandled exception
     from tracedemo.rb:6:in `y'
     from tracedemo.rb:2:in `x'
     from tracedemo.rb:13:in `<main>'
```

Jest to oczywiście tylko nieznacznie „upiększona” wersja tablicy danych śledzenia stosu, jaką uzyskano za pierwszym razem z metody `caller`.

Dane śledzenia stosu w języku Ruby są przydatne, ale wzbudzają też nieufność, ponieważ składają się wyłącznie z łańcuchów. Aby w jakiś sposób wykorzystać informacje zapewniane przez dane śledzenia stosu, musisz dokonać skanowania lub analizy

łańcucha i wyodrębnić wartościowe informacje. Innym rozwiązaniem jest utworzenie narzędzia Ruby przeznaczonego do analizowania danych śledzenia stosu i przekształcania ich w obiekty.

### 15.4.2. Tworzenie narzędzia do analizowania danych śledzenia stosu

Dysponując danymi śledzenia stosu (tablica łańcuchów), chcemy wygenerować tablicę obiektów, z których każdy zawiera informacje o programie lub nazwie pliku, numerze wiersza i nazwie metody (lub `<main>`). Utworzymy klasę `Call`, która będzie reprezentować jeden krok śledzenia stosu dla obiektu, a także klasę `Stack` reprezentującą wszystkie dane śledzenia stosu i złożoną z jednego lub większej liczby obiektów `Call`. Aby zminimalizować ryzyko kolizji nazw, umieśćmy obie klasy wewnątrz modułu `CallerTools`. Zacznijmy od bardziej szczegółowego opisania przeznaczenia każdej z tych dwóch klas.

Klasa `CallerTools::Call` będzie dysponować trzema atrybutami odczytu: `program`, `line` i `meth` (lepiej użyć łańcucha `meth` niż łańcucha `method` jako nazwy trzeciego atrybutu, ponieważ klasy zawierają już metodę o nazwie `method`, która nie ma zostać przesłonięta). W momencie inicjalizacji obiekt tej klasy dokona analizy łańcucha śledzenia stosu i zapisze odpowiednie podłańcuchy we właściwych zmiennych instancji w celu późniejszego pobierania za pośrednictwem metod wczytujących atrybuty.

Klasa `CallerTools::Stack` będzie przechowywać jeden lub większą liczbę obiektów `Call` w tablicy, która z kolei będzie składowana w zmiennej instancji `@backtrace`. Zostanie też utworzona metoda `report`, która wygeneruje (dość) ładną, wyświetlaną reprezentację wszystkich informacji w tym konkretnym stosie wywołań.

Utwórzmy teraz klasy.

#### KLASA CALLERTOOLS::CALL

Poniższy listing 15.5 prezentuje klasę `Call` wraz z pierwszym wierszem całego programu, który całą resztę opakowuje w module `CallerTools`.

**Listing 15.5. Początek modułu CallerTools uwzględniający klasę Call**

```
module CallerTools
  class Call
    CALL_RE = /(.*):(\d+):in `(.*)'/          ←❶
    attr_reader :program, :line, :meth        ←❷
    def initialize(string)
      @program, @line, @meth = CALL_RE.match(string).captures  ←❸
    end
    def to_s
      "%30s%5s%15s" % [program, line, meth]   ❹
    end
  end
end
```

Wymagane jest wyrażenie regularne, za pomocą którego zostanie dokonana analiza łańcuchów śledzenia stosu. Wyrażenie to jest przechowywane w stałej `CALL_RE` ❶. Zawiera ona trzy grupy przechwyceń w nawiasach okrągłych, które są oddzielone podłańcuchami literałów bez przechwyceń. Poniżej pokazano, jak wyrażenie regularne dopasowuje typowy łańcuch śledzenia stosu. Pogrubiona czcionka wyróżnia

podłańcuchy przechwytywane przez odpowiednie podwzorce wyrażenia regularnego. Znaki bez pogrubienia nie są uwzględniane w przechwyceniach, lecz dopasowywane dosłownie:

```
myrubyfile.rb:234:in `a_method'
   .*        :\d+:in '  .*    '
```

Jak podano, klasa zawiera trzy atrybuty odczytu dla trzech składników wywołania ❷. Inicjalizacja wymaga umieszczenia za pomocą przypisania równoległego ❸ argumentu w postaci łańcucha, łańcucha dopasowywanego względem stałej `CALL_RE` oraz wyników dostępnych za pośrednictwem metody `captures` obiektu `MatchData` w trzech zmiennych instancji, które odpowiadają atrybutom (jeśli nie występuje dopasowanie, przy próbie wywołania metody `captures` dla obiektu `nil` pojawia się błąd krytyczny; w razie potrzeby możesz zmodyfikować kod pod kątem bezpośredniej obsługi tego warunku).

Definiujemy również metodę `to_s` dla obiektów `Call` ❹. Metoda ta pojawia się w sytuacjach, w których przydatne jest wyświetlenie raportu dotyczącego konkretnego elementu `backtrace`. Uwzględnia to pomocną technikę języka Ruby opartą na znaku `%`. Po lewej stronie tego znaku znajduje się łańcuch formatujący w stylu metody `sprintf`, natomiast po prawej stronie jest tablica wartości zastępujących. Jeśli preferujesz inny styl danych wyjściowych, możesz zmodyfikować długości pól w łańcuchu zastępującym lub w tym przypadku utworzyć własną metodę `to_s`.

Pora zająć się klasą `Stack`.

### KLASA CALLERTOOLS::STACK

Poniższy listing 15.6 prezentuje klasę `Stack` wraz z instrukcją domykającą `end` całego modułu `CallerTools`.

**Listing 15.6. Klasa CallerTools::Stack**

```
class Stack
    def initialize
      stack = caller                  ←— ❶
      stack.shift
      @backtrace = stack.map do |call|  ←— ❷
        Call.new(call)
      end
    end
    def report                        │
      @backtrace.map do |call|        │
        call.to_s                     ├— ❸
      end                             │
    end
    def find(&block)                  ←— ❹
      @backtrace.find(&block)
    end
  end
```

W momencie inicjalizacji nowy obiekt `Stack` wywołuje metodę `caller` i zapisuje wynikową tablicę ❶. Następnie obiekt zmienia tę tablicę, usuwając pierwszy łańcuch, który informuje o wywołaniu samej metody `Stack.new`, dlatego stanowi jedynie „szum”.

Przechowywana zmienna `@backtrace` powinna zawierać jeden obiekt `Call` dla każdego łańcucha w tablicy `my_caller`. Odpowiada za to metoda `map` ❷. Zauważ, że istnieje atrybut odczytu `backtrace`. W tym przypadku niezbędna jest jedynie zmienna instancji na wewnętrzny użytek obiektu.

W dalszej kolejności pojawia się metoda `report`, która używa metody `map` w tablicy zmiennej `@backtrace` w celu wygenerowania tablicy łańcuchów dla wszystkich obiektów `Call` w stosie ❸. Ta tablica raportująca jest odpowiednia w przypadku wyświetlania, a jeśli taka zaistnieje potrzeba — wyszukiwania i filtrowania.

Klasa `Stack` uwzględnia jedną metodę finalną `find` ❹. Działanie klasy polega na przekazywaniu swojego bloku kodu metodzie `find` tablicy zmiennej `@backtrace`. Metoda ta działa bardzo podobnie do niektórych metod wcześniej zaprezentowanych w ramach przykładu z talią kart. Metody te przekazują metodę tablicy zawierającej karty tworzące talię. Takie techniki umożliwiają dostrajanie interfejsu obiektów, używając bazowych obiektów do zapewniania im dokładnie takich funkcji, jakich wymagają (wkrótce dowiesz się o specyficznej przydatności metody `find`).

Wypróbujmy teraz moduł `CallerTools`.

### UŻYCIE MODUŁU CALLERTOOLS

Aby wypróbować moduł `CallerTools`, możesz użyć zmodyfikowanej wersji demonstracji (z metodami `x`, `y` i `z`) z punktu 15.4.1. Następujący kod umieść w pliku o nazwie *callertest.rb*:

```
require_relative 'callertools'
def x
  y
end
def y
  z
end
def z
  stack = CallerTools::Stack.new
  puts stack.report
end
x
```

Po uruchomieniu programu zostaną wyświetlone następujące dane wyjściowe:

```
callertest.rb   12          z
callertest.rb    8          y
callertest.rb    4          x
callertest.rb   16      <main>
```

Choć nie wyróżnia się niczym szczególnym, jest to wygodny sposób programistyczny adresowania danych śledzenia stosu zamiast każdorazowego bezpośredniego modyfikowania łańcuchów (na początku wierszy jest wiele wolnego miejsca, ale byłoby go mniej, gdyby ścieżki do plików były dłuższe; oczywiście możesz dostosować formatowanie wedle własnego uznania).

Następnym w „programie”, a zarazem ostatnim „przystankiem” w tym rozdziale jest projekt, który wiąże ze sobą kilka wcześniej omawianych technik: śledzenie stosu, odpytywanie metod i wywołania zwrotne, a także niektóre techniki znane z różnych miejsc w książce. Utworzymy środowisko testowe.

## 15.5. Wywołania zwrotne i inspekcja metod w praktyce

W tym podrozdziale zaimplementujemy niewielkie środowisko testowe MicroTest. Nie zawiera ono wielu funkcji, lecz takie, które zademonstrują niektóre możliwości i wyrazistość właśnie poznanych technik wywołań zwrotnych i inspekcji.

Najpierw trochę informacji wprowadzających.

### 15.5.1. Fundament środowiska MicroTest: MiniTest

Język Ruby jest wyposażony w środowisko testowe o nazwie MiniTest. Za jego pomocą możesz utworzyć klasę, która dziedziczy z klasy `MiniTest::Unit::TestCase`, a ponadto zawiera metody o nazwach rozpoczynających się łańcuchem `string`. Możesz następnie określić, jakie metody testowe mają zostać wykonane, lub zapewnić (zostanie to zaprezentowane poniżej) automatyczne wykonanie po uruchomieniu pliku każdej metody zawierającej w nazwie łańcuch `test`. Wewnątrz tych metod tworzysz *asercje*. Prawdziwość lub fałszywość asercji określa, czy testy zakończą się pomyślnie.

Zaprezentowane poniżej ćwiczenie ma na celu utworzenie prostego narzędzia testującego opartego na części tych samych zasad co środowisko MiniTest. Aby ułatwić zorientowanie się w tym, przyjrzymy się najpierw pełnemu przykładowi użycia środowiska MiniTest, a następnie wykonamy ćwiczenie związane z implementacją.

Test będzie dotyczyć rozdawania kart do gry. Poniższy listing 15.7 prezentuje wersję klasy dla talii kart. Talii kart odpowiada tablica 52 łańcuchów przechowywanych w zmiennej instancji `@cards`. Rozdanie jednej lub większej liczby kart oznacza zdjęcie ich z góry talii.

**Listing 15.7. Implementacja talii kart oferująca możliwość rozdawania kart**

```
module PlayingCards
  RANKS = %w{ 2 3 4 5 6 7 8 9 10 J Q K A }
  SUITS = %w{ trefle kara kiery piki }
  class Deck
    def initialize  ←——❶
      @cards = []
      RANKS.each do |r|
        SUITS.each do |s|
          @cards << "#{r} of #{s}"
        end
      end
      @cards.shuffle!
    end
    def deal(n=1)  ←——❷
      @cards.pop(n)
    end
    def size
      @cards.size
```

```
    end
  end
end
```

Tworzenie nowej talii ❶ wiąże się z inicjalizowaniem zmiennej `@cards`, do której wstawiane są 52 łańcuchy, a także z przestawianiem zawartości tablicy. Każdy łańcuch przyjmuje postać *rangi koloru*, gdzie *ranga* to jedna z rang w tablicy stałej `RANKS`, a *kolor* to jeden z kolorów w tablicy stałej `SUITS`. Podczas rozdawania kart z talii ❷ zwracana jest tablica `n` kart, gdzie `n` reprezentuje liczbę rozdanych kart z wartością domyślną `1`.

Na razie wygląda to dobrze. Przeprowadźmy teraz test. Uruchom środowisko MiniTest. Poniższy listing 15.8 prezentuje kod testu dla klasy kart. W przypadku tego kodu przyjęto, że kod związany z kartami zapisano w osobnym pliku o nazwie *cards.rb* w tym samym katalogu co plik kodu testu (możesz mu nadać nazwę *cardtest.rb*).

**Listing 15.8. Plik cardtest.rb: testowanie precyzji rozdawania kart obsługiwanej przez klasę PlayingCards::Deck**

```
require 'minitest/unit'                    ←❶
require 'minitest/autorun'
require_relative 'cards'
class CardTest < MiniTest::Unit::TestCase  ←❷
def setup                                  ←❸
    @deck = PlayingCards::Deck.new
  end
  def test_deal_one                        ←❹
    @deck.deal
    assert_equal(51, @deck.size)           ←❺
  end
  def test_deal_many                       ←❻
    @deck.deal(5)
    assert_equal(47, @deck.size)
  end
end
```

Podstawową kwestią jest wymóg zarówno biblioteki `minitest/unit`, jak i pliku `cards.rb` ❶. Wymagana jest też funkcja `minitest/autorun`, która powoduje uruchamianie przez środowisko MiniTest napotykanych metod testu bez konieczności tworzenia jawnych wywołań metod. Następnie tworzona jest klasa `CardTest` dziedzicząca z klasy `MiniTest::Unit::`
`↪TestCase` ❷. W klasie `CardTest` definiowane są trzy metody. Pierwsza z nich ma nazwę `setup` ❸. Nazwa tej metody ma „magiczne" znaczenie dla środowiska MiniTest. Jeśli metoda zostanie zdefiniowana, będzie wykonywana przed każdą metodą testu w klasie testu. Uruchomienie metody `setup` przed każdą metodą testu ma wpływ na utrzymywanie wzajemnej niezależności metod testu, która stanowi ważną część architektury pakietów testów.

W dalszej kolejności pojawiają się dwie metody testu: `test_deal_one` ❹ i `test_`
`↪deal_many` ❻. Definiują one rzeczywiste testy. W każdym przypadku rozdawane są karty z talii, a następnie wykonywana jest asercja dotycząca wielkości talii po rozdaniu.

Pamiętaj, że metoda `setup` jest wykonywana przed każdą metodą testu. Oznacza to, że zmienna `@deck` zawiera dla każdej metody pełną talię liczącą 52 karty.

Asercje są wykonywane za pomocą metody `assert_equal` ❺, która pobiera dwa argumenty. Jeśli są one równe (w celu wykonania operacji porównania stosowany jest operator ==), asercja kończy się powodzeniem. W przeciwnym razie ma miejsce niepowodzenie.

Z poziomu wiersza poleceń uruchom plik *cardtest.rb*. Oto dane wyjściowe, jakie uzyskasz (używając prawdopodobnie innej wartości inicjującej i pomiarów czasu):

```
$ ruby cardtest.rb
Run options: --seed 39562
# Running tests:
..
Finished tests in 0.000784s, 2551.0204 tests/s, 2551.0204 assertions/s.
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

Ostatni wiersz informuje o tym, że wystąpiły dwie metody, których nazwy zaczynały się od łańcucha `test` (`2 tests`), a ponadto wskazuje, że w sumie istnieją dwie asercje (dwa wywołania metody `assert_equal`). Poza tym wiersz zawiera informacje o tym, że obie asercje powiodły się (brak niepowodzeń), a ponadto wszystko przebiegło bez większych problemów (brak błędów; błąd jest czymś nienaprawialnym, takim jak odwołanie do nieznanej zmiennej, natomiast niepowodzenie to niepoprawna asercja). Wiersz podaje też informacje o tym, że nie zostały pominięte żadne testy (pominięcie testu jest czymś możliwym do osiągnięcia w sposób jawny za pomocą wywołania metody `skip`).

W przypadku uruchamiania tego pliku testu najbardziej znamienne jest to, że w żadnym momencie nie ma potrzeby *tworzenia instancji* klasy `CardTest` albo jawnego wywoływania metod testu lub metody `setup`. Dzięki ładowaniu funkcji `autorun` środowisko MiniTest stwierdza, że ma uruchomić wszystkie metody o nazwach zaczynających się łańcuchem `test`, uruchamiając metodę `setup` przed każdą z nich. Takie automatyczne wykonywanie jest tym, co zostanie zaimplementowane (lub przynajmniej częściowo) w wykonywanym ćwiczeniu.

### 15.5.2. Określanie i implementowanie narzędzia MicroTest

Od narzędzia MicroTest będziemy oczekiwać następujących rzeczy:

- Automatyczne wykonywanie metody `setup` i metod testu w oparciu o dziedziczenie klasy.
- Prosta metoda asercji, której działanie kończy się pomyślnie lub niepowodzeniem.

Pierwsza specyfikacja będzie uwzględniać większość działań.

Wymagamy klasy, która gdy jest dziedziczona, „stwierdza” istnienie nowej podklasy, a ponadto wykonuje jej metody po ich zdefiniowaniu. Aby zachować (względną) prostotę, metody będą wykonywane w kolejności ich definiowania. Oznacza to, że metoda `setup` powinna zostać zdefiniowana jako pierwsza.

Oto bardziej szczegółowy opis kroków niezbędnych do zaimplementowania narzędzia MicroTest:

1. Zdefiniuj klasę `MicroTest`.
2. Zdefiniuj metodę `MicroTest.inherited`.
3. Wewnątrz metody `inherited` klasa dziedzicząca powinna...
4. Zdefiniuj własne wywołanie zwrotne `method_added`, które powinno...
5. Utwórz instancję klasy i wykonaj nową metodę, jeśli jej nazwa zaczyna się od łańcucha `test`, lecz najpierw...
6. Wykonaj metodę `setup` (jeśli istnieje).

Oto niedziałający model klasy `MicroTest` z komentarzami w postaci kodu Ruby:

```
class MicroTest
  def self.inherited(c)
    c.class_eval do
      def self.method_added(m)
        # Jeśli metoda m zaczyna się od łańcucha test
        #     Utwórz instancję klasy c
        #     Jeśli istnieje metoda setup
        #         Wykonaj metodę setup
        #     Wykonaj metodę m
      end
    end
  end
end
```

W tym przypadku występuje coś w rodzaju kaskady logicznej. Wewnątrz klasy `MicroTest` definiujemy metodę `self.inherited`, która jako swój argument odbiera klasę dziedziczącą (nowa podklasa). Używając metody `class_eval`, uzyskujemy następnie dostęp do zasięgu definicji klasy. W obrębie tego zasięgu implementujemy metodę `method_added`, która będzie wywoływana każdorazowo w momencie definiowania nowej metody w klasie.

Tworzenie pełnego kodu wynika bezpośrednio z komentarzy znajdujących się w modelu kodu. Poniższy listing 15.9 prezentuje pełną wersję pliku *micro_test.rb*. Plik umieść w tym samym katalogu co plik *callertools.rb*.

**Listing 15.9. Klasa testowa MicroTest, która emuluje niektóre funkcje środowiska MiniTest**

```
require_relative 'callertools'
class MicroTest
  def self.inherited(c)
    c.class_eval do
      def self.method_added(m)
        if m =~ /^test/          ←— ❶
          obj = self.new         ←— ❷
          if self.instance_methods.include?(:setup)  ←— ❸
            obj.setup
          end
         obj.send(m)
        end
      end
    end
  end
  def assert(assertion)          ←— ❹
```

```
    if assertion
      puts "Asercja powiodła się"
      true
    else
      puts "Asercja nie powiodła się:"
      stack = CallerTools::Stack.new
      failure = stack.find {|call| call.meth !~ /assert/ }   ❺
      puts failure
      false
    end
  end
  def assert_equal(expected, actual)   ❻
    result = assert(expected == actual)
    puts "(#{actual} to nie #{expected})" unless result   ❼
    result
  end
end
```

Wewnątrz zasięgu definicji klasy (`class_eval`) nowej podklasy definiujemy metodę `method_added`, w której ma miejsce większość działań. Jeśli nazwa definiowanej metody zaczyna się od łańcucha `test` ❶, tworzymy nową instancję klasy ❷. Jeśli zdefiniowano metodę `setup` ❸, wywołujemy ją dla tej instancji. W dalszej kolejności (niezależnie od tego, czy istnieje opcjonalna metoda `setup`) wywołujemy nowo dodaną metodę za pomocą metody `send`, ponieważ nie znamy nazwy metody.

> **UWAGA** Choć może wydać się to dziwne (z punktu widzenia tradycyjnego pojmowania dopasowywania wzorca, które uwzględnia łańcuchy), `m` w operacji dopasowywania wzorca `m =~ /^test/` jest symbolem, a nie łańcuchem. Możliwość dopasowywania się obiektów symboli względem wyrażeń regularnych jest częścią ogólnego działania, o którym już wspomniano, mającego na celu sprawienie, że symbole w większym stopniu będą wymienne z łańcuchami. Pamiętaj jednak o istotnych różnicach między nimi (była o tym mowa w rozdziale 8.).

Metoda `assert` sprawdza prawdziwość swojego pojedynczego argumentu ❹. Jeśli jest on prawdziwy (w sensie boolowskim; nie musi to oznaczać, że jest faktycznym obiektem `true`), wyświetlany jest komunikat wskazujący powodzenie. Jeśli asercja nie powiedzie się, wyświetlanie komunikatu staje się trochę bardziej złożone. Tworzymy obiekt `CallerTools::Stack` i określamy pierwszy obiekt `Call` w stosie, którego metoda nie zawiera w nazwie łańcucha `assert` ❺. Celem jest zapewnienie, że niepowodzenie nie jest zgłaszane jako takie, które wystąpiło w metodzie `assert` czy w metodzie `assert_equal` (zostanie poniżej opisana). Nie jest to coś solidnego. Może istnieć metoda zawierająca metodę `assert`, z której miał być zgłaszany błąd. Ilustruje to jednak rodzaj modyfikacji, jaką umożliwia metoda `find` klasy `CallerTools::Stack`.

Druga metoda asercji `assert_equal` sprawdza równość swoich dwóch argumentów ❻. W tym celu w ramach porównania wywoływana jest metoda `assert`. Jeśli wynik nie jest prawdziwy, zostanie wyświetlony komunikat o błędzie prezentujący dwa porównywane obiekty ❼. Niezależnie od tego, czy wystąpi powodzenie, czy niepowodzenie, wynik wywołania metody `assert` jest zwracany z metody `assert_equal`.

Aby wypróbować klasę `MicroTest`, następujący kod umieść w pliku o nazwie *microcardtest.rb* i uruchom go z poziomu wiersza poleceń:

```
require_relative 'microtest'
require_relative 'cards'
class CardTest < MicroTest
  def setup
    @deck = PlayingCards::Deck.new
  end
  def test_deal_one
    @deck.deal
    assert_equal(51, @deck.size)
  end
  def test_deal_many
    @deck.deal(5)
    assert_equal(47, @deck.size)
  end
end
```

Jak widać, kod ten jest prawie taki sam jak napisany wcześniej kod pliku testu środowiska MiniTest. Jedyną różnicą są nazwy biblioteki testu i klasy nadrzędnej testu. Po uruchomieniu kodu uzyskasz następujące trochę niewyraźne, lecz bardzo obiecujące wyniki:

```
Asercja powiodła się
Asercja powiodła się
```

Aby zobaczyć niepowodzenie, w metodzie `test_deal_one` wartość `51` zmień na `50`:

```
Asercja nie powiodła się:
              microcardtest.rb   11  test_deal_one
(51 to nie 50)
Asercja powiodła się
```

Środowisko testowe MicroTest nie zastąpi wkrótce środowiska MiniTest, ale podobnie jak ono realizuje kilka spośród najbardziej „magicznych" rzeczy. Wszystko to jest możliwe dzięki narzędziom introspekcji i wywołań zwrotnych języka Ruby, czyli technik, które udostępniają niezwykłe możliwości i elastyczność.

## 15.6. Podsumowanie

W rozdziale omówiono następujące zagadnienia:

- Przechwytywanie metod za pomocą metody `method_missing`.
- „Haki" i wywołania zwrotne środowiska wykonawczego przeznaczone dla obiektów, klas i modułów.
- Odpytywanie obiektów w odniesieniu do ich metod z wykorzystaniem różnych kryteriów.
- Wychwytywanie odwołań do nieznanych stałych.
- Dane śledzenia stosu.
- Tworzenie środowiska MicroTest.

W rozdziale zaprezentowano wiele podstawowych informacji. Praktyczne wykorzystanie omówionych tutaj technik będzie mieć znaczący wpływ na ugruntowanie posiadanej wiedzy na temat języka Ruby. Przyjrzeliśmy się przechwytywaniu nieznanych komunikatów za pomocą metody `method_missing`, a także innym „hakom" i wywołaniom zwrotnym środowiska wykonawczego, takim jak `Module.included`, `Module.extended` i `Class.inherited`. W rozdziale zaprezentowano również różne niuanse operacji odpytywania metod: na poziomie publicznym, chronionym i prywatnym oraz klasy, instancji i pojedynczego obiektu. Pokazano kilka przykładów, jak tego rodzaju odpytywanie może ułatwić uzyskiwanie informacji o sposobie organizowania przez język Ruby własnych klas, modułów i metod.

Ostatnim ogólnym zagadnieniem była obsługa danych śledzenia stosu, które są wykorzystywane w module `CallerTools`. Rozdział zakończono rozbudowanym ćwiczeniem złożonym z implementowania klasy `MicroTest`, która połączyła kilka tematów i wątków z tego oraz innych rozdziałów książki.

Materiał zaprezentowano w sposób metodyczny i dokładny, jak przystało na zdobywanie podstawowej wiedzy lub przygotowywanie się. Jeśli jednak spojrzysz na wyniki, a zwłaszcza na klasę `MicroTest`, możesz zauważyć, jak duże możliwości oferuje język Ruby przy stosunkowo niewielkim nakładzie pracy. Z tego właśnie powodu warto dowiedzieć się nawet o tym, co może się wydawać „magicznymi" lub „meta" elementami języka Ruby. W rzeczywistości takimi nie są. Wszystko to tworzy ten język. Po przyswojeniu sobie zasad dotyczących struktury obiektów i klas oraz relacji między nimi z wszystkim innym sobie poradzisz.

I to tyle! Korzystaj ze zdobytej wiedzy o języku Ruby oraz z wielu struktur, jakie stworzysz na bazie podstawowych informacji uzyskanych po przeczytaniu tej książki.

# Skorowidz

## A


analiza argumentów metody, 80
analizator kodu, 197
anatomia instalacji, 45
argumenty, 38
    metody, 80, 91
    obiektu Proc, 494
    opcjonalne, 80
    wymagane, 80
ASCII, 275
asercje
    wsteczne, 405
    wyprzedzające, 404
    wyrażeń regularnych, 403
atrybuty, 108
automatyzowanie tworzenia atrybutów, 108


## B


biblioteka, 48
    Active Support, 471
    drb, 46
    open3, 519
biblioteki
    programistyczne, 48
    standardowe, 46
bloki kodu, 209, 218, 487
błędy, 221
    operacji wejścia-wyjścia, 432
    Errno, 432
    składni, 40


## C


cechy charakterystyczne symboli, 283
czas, 292
czyszczenie tablicy asocjacyjnej, 322


## D


dane
    śledzenia stosu, 546
    wprowadzane z klawiatury, 42
data, 292
definiowanie
    metody najwyższego poziomu, 187
    operatorów, 236
    wielokrotne metody, 138
    zachowania obiektu, 68
delegowanie, 525
destrukcyjność, 241
dodatki, 518
dodawanie
    metod klasy, 473
    modułu do klasy, 131, 151
    obiektu do zbioru, 332
dokument miejscowy, 267
dołączanie, 528
    danych, 237
    modułu, 457
    wielokrotne modułu, 140
    symboli, 285
domknięcia, 491, 493
domyślny odbiorca komunikatów, 162
dopasowania, 201
    podrzędne, 392
    warunkowe, 405
dopasowanie
    pomyślne, 394
    zakończone niepowodzeniem, 394
doprecyzowania, 475
dostęp do metody, 156, 182
dostosowywanie operatorów
  jednoargumentowych, 238
dostrajanie wyrażeń regularnych, 398
druga iteracja konwertera, 41
dynamika języka, 449
działania niedozwolone, 85

działanie
enumeratorów, 377
narzędzia irb, 58
dziedziczenie, 111, 126, 151
dziedziczenie pojedyncze, 113

## E

elementy tablicy, 308
enumeratory, 364, 380
leniwe, 380

## F

fałsz, 253
FIFO, First In, First Out, 129
filtrowanie kolekcji, 301
flaga FNM_DOTMATCH, 438
formatowanie daty i czasu, 296
funkcje anonimowe, 484

## G

generowanie nowego zasięgu lokalnego, 172
gra w kamień, papier i nożyce, 513

## H

haki, 523, 524
hermetyzowanie zachowań, 129
hierarchia
dziedziczenia, 462
klas, 111, 178

## I

identyfikator, 35
identyfikatory liczbowe, 78
identyfikowanie obiektów, 77
iloczyn zbiorów, 332
implementowanie
metody SYMBOL#TO_PROC, 490
metody times, 213
narzędzia MicroTest, 551
podklasy, 478
indeksowanie obiektów wyliczeniowych, 377
indywidualizacja obiektów, 451
informacje o pliku, 435
inicjowanie
elementów tablicy, 305
obiektu, 102
inspekcja możliwości obiektów, 258
instalowanie
języka, 45
pakietów, 62
instancje, 96, 124
instrukcja
case, 200, 202, 203, 204
each, 61, 431
if, 192, 193, 196
prepend, 141, 142
unless, 195
when, 202
interpolacja łańcuchów, 73, 271
interpreter języka, 40, 57
interpretowanie zapytań, 535
introspekcja, 258, 523
zmiennych i stałych, 543
iteratory, 209

## J

jednowymiarowość, 263

## K

katalog
archdir, 46
gems, 47
rubylibdir, 46
site_ruby, 47
vendor_ruby, 47
katalogi instalacji języka, 46
klasa
ArgumentList, 486
BasicObject, 114, 137, 138, 476, 478
CALLERTOOLS::CALL, 546
CALLERTOOLS::STACK, 547
CargoHold, 133, 151
Class, 116, 127, 138
CodeBlock, 486
Dir, 436, 439
Enumerator, 337
Errno, 433
Fiber, 507
File, 428, 434

File::Stat, 435
IO, 420
Object, 113, 128, 138
Pathname, 442
Proc, 484
Rainbow, 339
RPS, 514
Stack, 131, 132, 133
String, 260
StringIO, 444
StringScanner, 413
Suitcase, 133
TCPServer, 510
klasy, 95, 96, 124
jako obiekty, 115, 126
liczbowe, 290
pojedynczych obiektów, 453, 456
wyjątków, 226, 228
wyliczeniowe, 340
znaków, 390, 391
klauzula
ensure, 227
rescue, 221, 223, 225
when, 201
klawiatura, 423
klucze tablicy asocjacyjnej, 286
kodowanie
ASCII, 280
łańcuchów, 280, 281
pliku źródłowego, 280
UTF-8, 280
kolejka
FIFO, 129
LIFO, 129
kolejność
kodów znaków, 275
parametrów i argumentów, 82
sortowania, 363
kolekcje, 301, 335
komentarze, 35
komunikacja dwukierunkowa, 519
komunikat puts, 38
komunikaty, 37
komunikaty do obiektów, 66, 79
konstruktor, 96
->, 494
Date.today, 293
hash.new, 316
lambda ->, 495
konstruktory
literałów, 234, 304
tablic %I I %I, 307
tablic %W I %W, 307
kontenery, 301
konwencja nazewnicza, 36
konwersja
daty i czasu, 297
liczb, 247
łańcucha, 243, 279
tablic, 246
konwerter temperatur, 40, 43, 44
konwertowanie konwertera, 119
kwantyfikatory, 398, 400

## L

lambda, 484
leniwe enumeratory, 380
LIFO, Last In, First Out, 129
lista
argumentów, 85
metod pojedynczego obiektu, 541
metod prywatnych, 537
wartości, 209
literały, 234
logika
klasy Person, 148
włączenia do zakresów, 328
lukier składniowy, 237

## Ł

ładowanie
pakietów, 63
plików, 48, 49
rozszerzeń zewnętrznych, 48
łańcuchy, 263, 264
łączenie
łańcuchów, 270, 271
opcji, 56
tablic, 311
tablic asocjacyjnych, 320

## M

metaklasy, 462
metoda, 65, 68
[], 316
Array, 304, 306
Array.new, 304
attr_accessor, 110
caller, 544
capture_block, 488
cards, 372
Class#inherited, 531
class_eval, 503
concat, 311
cover?, 328
cycle, 354
display, 245
drop, 350
dup, 91
each, 214, 216, 338, 342, 368, 370
each.with_index, 352
each_cons, 353
each_slice, 353
each_with_index, 352
encode, 281
end_with?, 273
entries, 436
Enumerator.new, 368
eval, 499, 500, 501
extend, 472, 474, 530
File.new, 430
File.open, 431
find, 343
find_all, 345
first, 348
float, 248
freeze, 91
grep, 345, 415
group_by, 347
hash, 317
include, 128, 272
include?, 329
included, 529
index, 274
initialize, 102
inject, 355
inspect, 244
instance_eval, 501
instance_exec, 502
instance_methods, 260
Integer, 248
Kernel#rand, 349
Kernel#test, 435
lambda, 494
load, 49, 50
load_and_report, 134
loop, 205
makes, 473
map, 216, 356, 357
map!, 358
match, 199
max, 350
method_added, 533
method_missing, 145, 525, 528
min, 350
Module#const_missing, 532
most_expensive, 117
my_each, 216
my_times, 216
new, 96, 119, 304
object_id, 77
open, 518
Open3.popen3, 518, 519
ord, 274
partition, 347
Person.method_missing, 149
pojedynczego obiektu, 117
proc, 485
Proc.new, 486
public_send, 80
REGEXP#MATCH, 464
require, 48, 51
respond_to, 78
respond_to?, 527
respond_to_missing?, 527
reverse_each, 352
second, 295
select, 321
send, 79
set_age, 527
singleton_class, 461
singleton_method_added, 533
singleton_methods, 541
sort_by, 364
stack, 130
start_with?, 273
store, 317
strftime, 296

String#gsub!, 465
String#scan, 411
String#split, 412
Symbol#to_proc, 490
system, 515, 516
take, 350, 367
tap, 466
times, 213
to_a, 246
to_ary, 249
to_f, 247
to_i, 247
to_proc, 489
to_str, 249
to_sym, 282
try_convert, 308
unload, 134
using, 475
with_index, 303, 377
metody
attr_*, 108, 111
chronione, 186, 537
formatujące datę i czas, 296
instancji, 97, 120
jako obiekty, 496
klasy, 120
klasy File, 428
konwersji, 262
konwersji daty i czasu, 297
najwyższego poziomu, 187
niebezpieczne, 239
o takiej samej nazwie, 140
obiektu pojedynczego, 184, 452
odczytujące, 110
odpytujące tablice, 314
pobierające argumenty, 69
prywatne, 182, 184, 537
przyjmujące role, 248
to_*, 248
ustawiające, 103, 106, 184
wbudowane, 242
z notacją wywołania, 237
moduł, 113, 127
CallerTools, 548
Comparable, 256, 275, 362
DRYRUN, 442
Enumerable, 302, 337, 338, 340, 382, 540
FileTest, 434
FileUtils, 440, 442
hermetyzujący, 129
NOWRITE, 442
open-uri, 446
Stacklike, 130, 131, 134
modyfikatory, 398, 406
modyfikatory warunkowe, 195
modyfikowanie
elementów zbioru, 331
funkcjonalności, 463
katalogów, 436, 439
klasy pojedynczych obiektów, 454
kluczy wątków, 512
łańcuchów, 268
metody regexp#match, 464
modułów, 463
podstawowych klas, 463
stałych, 123
zachowania funkcjonalności, 474

## N

nadzbiory, 334
narzędzia
plikowe, 440
standardowe, 52
narzędzie
erb, 52
gem, 52, 63
gsub, 414
irb, 33, 51
MicroTest, 551
rake, 52, 60
rdoc, 52, 59
ri, 52, 59
ruby, 52
Ruby Version Manager, 49
sub, 414
testrb, 52
nawiasy
klamrowe, 211
kwadratowe, 316
okrągłe, 303, 392, 402, 405
nazwy
metod, 37, 104
z wykrzyknikiem, 239, 242
zmiennych, 36, 102
niejawne tworzenie enumeratorów, 369

notacja
%w{...}, 304
class <<, 455
def obj.meth, 455
łańcuchów, 264
z wykrzyknikiem, 239, 242

## O

obiekt, 37, 65, 87
Class, 116
domyślny, 155
false, 252
File, 433
IO, 433
MatchData, 199, 392, 394, 397
Method, 496
nil, 250, 254
STDERR, 422
STDIN, 422
STDOUT, 422
true, 252
obiekt self, 38, 109, 156
a zasięg lokalny, 171
jako domyślny odbiorca komunikatów, 162
najwyższego poziomu, 158
określanie zmiennych instancji, 164
w definicjach klas i modułów, 159
w definicjach metod instancji, 160
w definicjach metod pojedynczego obiektu, 160
obiektowość, 66
obiekty
boolowskie, 250
Fiber, 507
IO, 421
klasy, 116
kolekcji, 337
liczbowe, 289
ogólne, 77
podstawowe, 77
Proc, 484, 487, 491
skalarne, 263
specjalne, 35
wyliczeniowe, 343, 421
ze stanem, 102
obsługa
błędów, 221
symboli wieloznacznych, 437
ochrona obiektów, 372
odbiorcy komunikatów, 115
odczytywanie plików, 424–426, 428
odpytywanie
katalogów, 439
łańcuchów, 272
obiektów
daty i czasu, 295
File, 433
IO, 433
klas, 260
tablic, 314
tablic asocjacyjnych, 322
odwołania, 87, 91
w przypisaniu do zmiennej, 89
odwracanie tablicy asocjacyjnej, 322
odwrotność przesłaniania, 370
odwrócone apostrofy, 515, 517
ograniczanie
dopasowań, 398
wywołania zwrotnego, 532
ogranicznik EOM, 267
określanie
obiektu self, 158
stałych, 173
zasięgu, 166
zmiennych instancji, 164
opcje wiersza poleceń, 53
operacja XOR, 378
operacje, 34
alternatywy wykluczającej, 378
arytmetyczne, 290
arytmetyczne daty i czasu, 297
na plikach, 419, 424
rozszerzania, 529
wejścia-wyjścia, 42, 419, 420
wyliczeniowe, 348
operator
negacji, 194
równości, 198, 345
równości przypadków, 237
operatory
arytmetyczne, 237
bitowe, 237
jednoargumentowe, 238
porównywania, 237
skrótów, 130
organizacja programu, 127
otwieranie klas, 98

## P

parametry
bloku, 218
obiektu Proc, 494
pary klucz i wartość, 317
pętla, 205
plik
cgi.rb, 46
drb.rb, 46
fileutils.rb, 46
loaddemo.rb, 49
loadee.rb, 49
Rakefile, 60
tempfile.rb, 46
pliki
.bundle, 46
.dll, 46
.so, 46
kopiowanie, 441
przenoszenie, 441
usuwanie, 441
pobieranie
danych, 237
podłańcuchów, 269
wartości z tablicy asocjacyjnej, 318
początek tablicy, 310
podzbiory, 334
polecenia systemowe, 515, 518
polecenie
cat, 518
desc, 61
gem, 62
irb, 33
require_relative, 52
porównanie
dwóch obiektów, 255
łańcuchów, 275
łańcuchów i symboli, 288
metod, 120
tablic i tablic asocjacyjnych, 302
PRAWDA, 253
problem
FizzBuzz, 381
loaddemo.rb, 49
proste dopasowywanie, 388
przechwycenia nazwane, 396
przechwytywanie
bloku kodu, 487
dziedziczenia, 531
nierozpoznanych komunikatów, 525
obiektów Method, 496
operacji rozszerzania, 529
podłańcuchów, 392
wyjątku, 225
przesłanianie metod, 97
przesłonięcia modułu enumerable, 540
przetwarzanie kolekcji, 304
przodkowie obiektów, 113
przypisanie, 105
dla stałej, 123
do zmiennej, 85, 89
zmiennej lokalnej, 197

## Q

quazi-obiekty wyliczeniowe, 359

## R

relacje między klasami, 462
rozszerzanie funkcjonalności obiektu, 472
rozszerzenie, 48
równoległe wykonywanie, 505
równość
łańcuchów, 275
obiektów, 78
różnica zbiorów, 332
Ruby, 32
rzeczywisty świat, 67
rzutowanie typów, 248

## S

scalanie kolekcji, 333
semantyka enumeratorów, 369
serwer
dat z wątkami, 508
rozmów sieciowych, 509
skalar, 263
składnia, 33
składnia przypisania, 197
składnik, 48
skracanie kodu, 43, 73

słowo kluczowe, 36
do, 212
else, 193
elsif, 193
end, 212
not, 194
rescue, 222
return, 495
super, 143, 145
unless, 194
until, 208
when, 201
while, 206
sortowanie
obiektów wyliczeniowych, 360, 362
zwięzłe, 364
specyfikatory formatu daty i czasu, 296
sposoby uzyskiwania przechwyceń, 395
sprawdzanie
kodu źródłowego, 45
obiektu self, 159
składni, 53
stałe, 120, 126, 173
stałe predefiniowane, 122
stan obiektu, 100
stany boolowskie, 250
sterowanie pętlą, 206
stosowanie
cudzysłowów, 265
pakietów, 63
suma zbiorów, 332
superklasy, 126, 152, 462
symbole, 263, 282
jako argumenty metody, 286
jako klucze tablicy asocjacyjnej, 286
niezmienność, 283
unikalność, 283
wieloznaczne, 437
szukanie pozycji w pliku, 427

## Ś

ścieżka
bezwzględna stałej, 174
wyszukiwania metod, 456
śledzenie wykonywania kodu, 544
środowisko MiniTest, 549

## T

tablice, 302
tablice asocjacyjne, 302, 315
argumenty nazwane, 324
czyszczenie, 322
łączenie, 320
odpytywanie, 322
odwracanie, 322
określanie domyślnych wartości, 319
ostatnie argumenty metody, 323
para klucz i wartość, 317
pobieranie wartości, 318
transformacje, 321
tworzenie, 315
wybieranie elementów, 321
zastępowanie, 322
techniki
delegowania, 526
przepływu sterowania, 191
testowanie
dołączenia symboli, 285
plikowych danych, 445
równości, 255
włączenia do zakresu, 329
transformacje
formatowania, 277
łańcuchów, 276
tablic, 312
tablic asocjacyjnych, 321
wielkości znaków, 276
zawartości, 278
transkodowanie, 281
tworzenie
atrybutów, 108, 110
enumeratorów, 365, 369
funkcji, 494
łańcucha metod enumeratora, 375
metod najwyższego poziomu, 187
metody klasy, 119
modułów, 128
obiektów
daty i czasu, 293, 294
klasy, 95, 115
obiektu biletu, 72
obiektu ogólnego, 67
pierwszego pliku programu, 39
programu, 39

serwera rozmów sieciowych, 509
tablicy, 304
asocjacyjnej, 315
asocjacyjnej literału, 316
wyrażeń regularnych, 387
wzorca, 389
zakresu, 327
zbiorów, 331

## U

uchwyty plików, 425
uruchamianie programu, 41
w trybie
szczegółowych informacji, 55
wierszy, 54
ustawianie
danych, 237
podłańcuchów, 269
usuwanie
obiektu do zbioru, 332
odwołania, 89
par tablic asocjacyjnych, 317
UTF-8, 280
utrzymywanie stanu klas, 180
uzyskiwanie przechwyceń, 395
użycie
doprecyzowań, 475
klasy BasicObject, 476
klasy Person, 146
klauzuli rescue, 223
metod jako obiektów, 497
metod najwyższego poziomu, 187
metody
class_eval, 503
display, 245
each enumeratora, 370
new, 304
Open3.popen3, 519
Symbol#to_proc, 490
modułu callertools, 548
modułów, 128
obiektów proc, 488
obiektu self, 161
przechwyceń, 414
słowa kluczowego super, 144
zaawansowane modułów, 133
zmiennych klasy, 177

## W

wartości
boolowskie, 74, 254
domyślne argumentów, 81
wartość
instrukcji if, 196
zwracana metody, 70
warunkowe wykonywanie
kodu, 192
pętli, 206
ważność metody each, 214
wątki, 505, 511
kończenie działania, 506
uruchamianie, 506
zatrzymywanie, 506
wbudowane
klasy, 231
metody najwyższego poziomu, 188
moduły, 231
zachowania obiektu, 76
zmienne globalne, 167
widoczność, 155
wiersz poleceń, 53
własne klasy wyjątków, 228
włączanie ostrzeżeń, 53
wstawianie, 528
wybieranie
dopasowań, 345
elementów z tablicy, 321
obiektów wyliczeniowych, 343
wyjątek, 221
ArgumentError, 222, 227
Errno::błąd, 222
IOError, 222
NameError, 222
NoMethodError, 222
RuntimeError, 222
TypeError, 222
wykonywanie
pętli, 209
poleceń systemowych, 515
programów systemowych, 516
skryptu, 54
warunkowe, 191
wykorzystanie enumeratorów, 373
wyliczanie, 374

wymaganie pliku lub rozszerzenia, 55
wymuszanie ścieżki bezwzględnej stałej, 174
wyrażenia
  arytmetyczne, 290
  regularne, 385, 386
    asercje, 403
    idiomy, 408
    konkretna liczba powtórzeń, 402
    konwersja łańcuchów, 408
    kwantyfikatory, 398
    metody, 411
    modyfikatory, 398, 406
    proste dopasowywanie, 388
    przejście do łańcucha, 410
    tworzenie wzorca, 389
    zakotwiczenia, 398, 403
  warunkowe, 197
wyrażenie <<EOM, 267
wysyłanie
  komunikatów do obiektów, 68, 79
wyszukiwanie
  metod, 135, 143
  obiektów wyliczeniowych, 343
wyświetlanie
  informacji pomocy, 56
  listy metod, 259, 261
  listy zmiennych, 543
  metod
    chronionych, 537
    nieprywatnych, 535
    pojedynczego obiektu, 541
    prywatnych, 537
  wersji języka, 55
  wzorców, 387
wywołania
  metod, 37, 116, 210
  programów systemowych, 517
  zwrotne, 523, 524
    extended, 530
    included, 530
    inherited, 532
wywołanie Class.new, 116
wyzwalanie wywołania zwrotnego, 530
wzorzec, 387
  pojedynczego obiektu, 463

## Z

zachowanie tablic asocjacyjnych, 319
zagnieżdżanie modułów i klas, 153
zakotwiczenia, 398
zakresy, 326
  testowanie włączenia, 328, 329
  wsteczne, 330
zapisywanie
  metod, 121
  programu, 39
  w plikach, 44, 429
zapytania
  boolowskie, 272, 340
  dotyczące treści, 273
zarządzanie zadaniami, 60
zasięg, 155, 156, 166
  globalny, 166
  lokalny, 169
  operacji na plikach, 430
  procesu wyszukiwania, 137
  zmiennych, 218
zastąpienia
  globalne, 414
  pojedyncze, 414
zastosowanie
  enumeratorów, 369
  łańcuchów, 264
  stałych, 121
  symboli, 282
zbiory, 330
zgłaszanie
  wyjątków, 221
    jawne, 224
    ponowne, 226
zmiany
  addytywne, 469
  z przekazaniem, 469
zmienne, 87, 511
  globalne, 166
    operacji wejścia-wyjścia, 422
    przechwycenia, 415
    $~, 398
    wbudowane, 167
    zalety i wady, 167

instancji, 36, 100
klasy, 175, 178
zalety i wady, 179
lokalne, 35, 65, 85, 92
znak
!, 194, 239
@, 36
gwiazdki, 246, 437
kropki, 327
odwróconego apostrofu, 515
plusa, 236
procentu, 105
równości, 104, 106, 242
spacji, 41
wieloznaczny kropki, 390
znaki
konstruktorów literałów, 235
literału, 389

## Ż

żądanie składnika, 50

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA