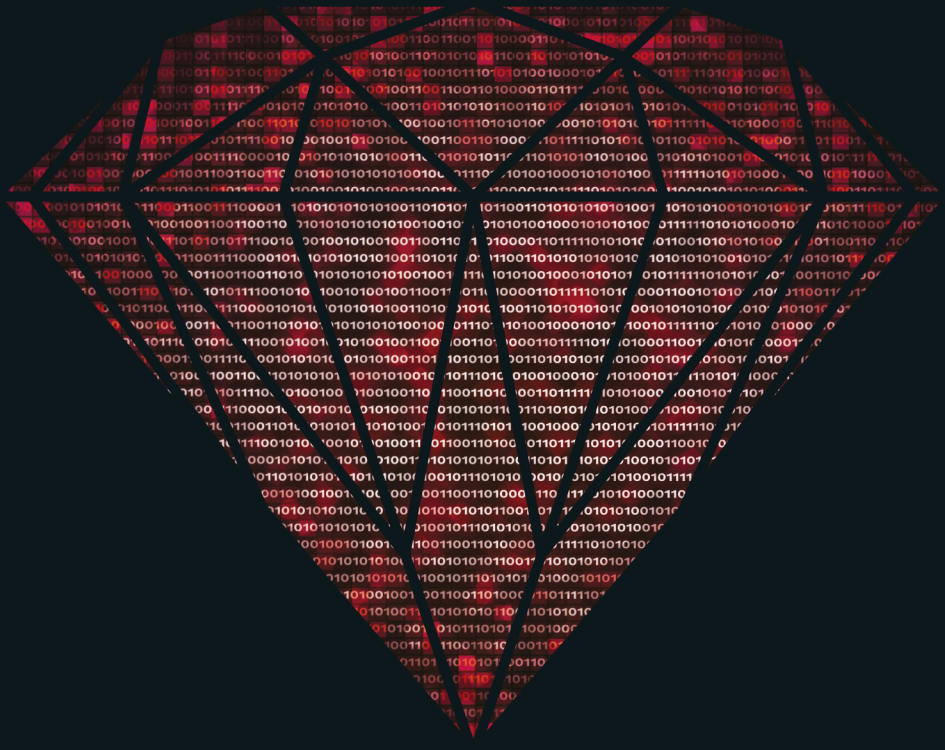
David A. Black

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: 978-83-283-1103-9

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
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

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)
	`2 - 3` (odejmowanie)
	`2 * 3` (mnożenie)
	`2 / 3` (dzielenie)
	`10.3 + 20.25`
	`103 - 202.5`
	`32.9 * 10`
	`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.
		
		Zauważ, że zamiast `.23` musisz wpisać `0.23`.

Przypisanie	`x = 1` `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`
	`s = "100"`
	`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"`
	`puts "Witaj"`
	`x = "Witaj"`
	`puts x`
	`x = "Witaj"`
	`print x`
	`x = "Witaj"`
	`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` `string = gets`	Wiersz z wprowadzonymi danymi możesz przypisać bezpośrednio do zmiennej (zmienna `string` w drugim przykładzie).

Wykonywanie warunkowe	`if x == y`
	`  puts "Tak!"`
	`else`
	`  puts "Nie!"`
	`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`
	`false`
	`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` `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  <-- 1
>> require "scanf"                                                 <-- 2
=> true
>> "David Black".scanf("%s%s")
=> ["David", "Black"]                                             <-- 3
```

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|                           <- 1
      next unless File.file?(f)                         <- 2
      print "Czy usunąć plik #{f}? "                    <- 3
      answer = $stdin.gets
      case answer
      when /^y/
        File.unlink(f)                                  <- 4
      when /^q/
        break                                           <- 5
      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.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

W

Z

Ż

Ruby to dojrzały język programowania, obecny na rynku już od dwudziestu lat, choć jego duża popularność datuje się od 2003 roku. Powodem wzmożonego zainteresowania tym językiem było opublikowanie niesamowitego szkieletu – Ruby on Rails. Wśród atutów języka Ruby należy wymienić automatyczne zarządzanie pamięcią, wyrażenia regularne wbudowane w składnię oraz tak zwany duck typing, czyli identyfikowanie typów na podstawie zachowania, a nie deklaracji. Masz ochotę nauczyć się programować w tym języku? Trafiłeś na świetną książkę!

Poznaj kluczowe pojęcia związane z językiem Ruby. W kolejnych rozdziałach znajdziesz istotne informacje na temat składni, dostępnych elementów oraz typowych konstrukcji. Ponadto zobaczysz, jak korzystać z obiektów, metod i ze zmiennych, oraz nauczysz się budować właściwą strukturę Twojego projektu. Po opanowaniu podstaw będziesz swobodnie tworzyć proste programy oraz przejdziesz do zaawansowanych tematów związanych z technikami przepływu sterowania, wbudowanymi elementami, kolekcjami oraz wyrażeniami regularnymi. Ta książka jest doskonałym podręcznikiem dla wszystkich osób chcących nauczyć się języka Ruby i wykorzystać jego potencjał w kolejnych projektach.

Dzięki tej książce:

- **poznasz składnię języka Ruby**
- **zrozumiesz filozofię stojącą za tym językiem**
- **opanujesz techniki programowania obiektowego**
- **wykorzystasz potencjał języka Ruby**

Poznaj i wykorzystaj możliwości języka Ruby!

David A. Black – jeden z najlepszych na świecie znawców języka Ruby. Główny konsultant w firmie Cyrus Innovation, założyciel organizacji Ruby Central i autor książek na temat Ruby. Aktywny prelegent oraz szkoleniowiec.

Helion

36350 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-283-1103-9

9 788328 311039

Informatyka w najlepszym wydaniu

cena: 89,00 zł