

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Programowanie w języku Ruby. Wydanie II

Autorzy: Dave Thomas, Chad Fowler, Andy Hunt

Tłumaczenie: Tomasz Bąk, Tomasz Walczak

ISBN: 83-246-0522-3

Tytuł oryginału: [Programming Ruby: The Pragmatic Programmers. Second Edition](#)

Format: B5, stron: 1072



Odkryj możliwości języka Ruby

- Opanuj zasady programowania obiektowego
- Połącz Ruby z innymi językami programowania
- Przetestuj aplikacje, wykorzystując testy jednostkowe

Ruby to obiektowy język programowania, który powstał w Japonii w 1993 roku. Od początku swojej obecności na rynku zdobywa ogromną popularność, stając się poważną konkurencją dla Perla. Dzięki technologii Ruby on Rails narzędzie to staje się coraz powszechniej wykorzystywane, szczególnie do tworzenia aplikacji internetowych. Ruby ma prostą składnię, zawiera możliwość obsługi wyjątków i wyrażeń regularnych oraz pozwala na stosowanie modułów. Ogromną zaletą tego języka jest jego zwarta konstrukcja – program napisany w Ruby jest wielokrotnie mniejszy niż realizująca podobne funkcje aplikacja utworzona w Javie lub C.

Książka „Programowanie w Ruby. Wydanie II” to jeden z najpopularniejszych na świecie podręczników opisujących ten język. Czytając ją, opanujesz wszystkie zagadnienia związane z tworzeniem aplikacji w Ruby. Poznasz zasady programowania obiektowego, korzystania z wyrażeń regularnych, obsługi wyjątków oraz pracy wielowątkowej. Dowiesz się, w jaki sposób dokumentować kod, budować aplikacje i skrypty sieciowe, stosować Ruby w systemie Windows oraz łączyć Ruby z C. W książce znajdziesz także szczegółowe omówienie klas i modułów języka Ruby.

- Instalacja i uruchamianie Ruby
- Klasy, obiekty i zmienne
- Typy danych
- Przechwytywanie wyjątków
- Operacje wejścia i wyjścia
- Wielowątkowość
- Testowanie aplikacji
- Pisanie skryptów CGI w Ruby
- Automatyzacja systemu Windows za pomocą Ruby
- Obiekty Ruby w języku C



Spis treści

Przedmowa do wydania pierwszego	15
Przedmowa do wydania drugiego	19
Przedmowa	21
Mapa	27

Część I Płaszczyzny Ruby

Rozdział 1. Pierwsze kroki	31
Instalacja Ruby	31
Uruchamianie Ruby	34
Dokumentacja Ruby — RDoc i ri	37
Rozdział 2. Ruby.new	41
Ruby to język zorientowany obiektowo	42
Wybrane podstawy Ruby	44
Tablice i tablice asocjacyjne	48
Struktury kontrolne	50
Wyrażenia regularne	51
Bloki i iteratory	53
Odczyt i zapis	56
Cała naprzód	58

Rozdział 3. Klasy, obiekty i zmienne	59
Dziedziczenie i komunikaty	61
Obiekty i atrybuty	64
Zmienne i metody klasy	68
Kontrola dostępu	73
Zmienne	76
Rozdział 4. Kontenery, bloki i iteratory	79
Kontenery	79
Bloki i iteratory	86
Kontenery są wszędzie	95
Rozdział 5. Typy standardowe	97
Liczby	97
Łańcuchy znaków	100
Przedziały	106
Wyrażenia regularne	108
Rozdział 6. Więcej na temat metod	121
Definiowanie metody	121
Wywoływanie metody	123
Rozdział 7. Wyrażenia	129
Wyrażenia operatorowe	130
Różnorodne wyrażenia	131
Przypisania	132
Wykonanie warunkowe	137
Wyrażenia case	142
Pętle	144
Zasięg zmiennych, pętle i bloki	151
Rozdział 8. Zgłaszanie i przechwytywanie wyjątków	153
Klasa Exception	154
Obsługa wyjątków	154
Zgłaszanie wyjątków	159
catch i throw	161

Rozdział 9. Moduły	163
Przestrzeń nazw	163
Miksiny	165
Iteratory i moduł Enumerable	167
Łączenie modułów	167
Dołączanie innych plików	171
Rozdział 10. Podstawy wejścia-wyjścia	173
Czym jest obiekt IO?	173
Otwieranie i zamykanie plików	174
Odczyt i zapis plików	175
Komunikacja sieciowa	180
Rozdział 11. Wątki i procesy	183
Wielowątkowość	183
Zarządzanie szeregowaniem wątków	189
Wzajemne wykluczanie	191
Uruchamianie wielu procesów	198
Rozdział 12. Testy jednostkowe	203
Szkielet Test::Unit	205
Struktura testów	209
Optymalizacja i uruchamianie testów	213
Rozdział 13. Gdy pojawiają się problemy	217
Debugger Ruby	217
Ruby interaktywny	219
Obsługa edytora	221
Ale to nie działa!	221
Ależ to jest powolne!	225

Część II Ruby w oparciu

Rozdział 14. Ruby i jego świat	233
Argumenty wiersza poleceń	233
Zakończenie programu	237
Zmienne środowiska wykonania	238

Gdzie Ruby znajduje swoje moduły?	239
Środowisko budowania	241
Rozdział 15. Interaktywna powłoka Ruby	243
Wiersz poleceń	243
Konfiguracja	248
Polecenia	253
Ograniczenia	256
rtags i xmp	256
Rozdział 16. Dokumentowanie Ruby	259
Dodawanie RDoc do kodu Ruby	261
Dodawanie RDoc do rozszerzeń C	269
Uruchamianie RDoc	270
Wyświetlanie informacji o użyciu programu	274
Rozdział 17. Zarządzanie pakietami za pomocą RubyGems	277
Instalacja RubyGems	278
Instalacja gemów aplikacji	279
Instalacja i używanie gemów bibliotek	281
Tworzenie własnych gemów	287
Rozdział 18. Ruby i sieć	299
Pisanie skryptów CGI	299
Ciasteczka	310
Zwiększanie wydajności	312
Wybór serwera WWW	313
SOAP i usługi sieciowe	315
Więcej informacji	321
Rozdział 19. Biblioteka Tk w Ruby	323
Prosta aplikacja używająca Tk	324
Kontrolki	324
Wiązanie zdarzeń	329
Kontrolka Canvas	331
Przewijanie	332
Używanie dokumentacji Perl/Tk	335

Rozdział 20. Ruby i system Windows	337
Pobieranie Ruby dla systemu Windows	337
Uruchamianie Ruby w systemie Windows	338
Win32API	339
Automatyzacja systemu Windows	340
Rozdział 21. Rozszerzenia języka Ruby	347
Pierwsze rozszerzenie	348
Obiekty Ruby w języku C	351
Rozszerzenie dla szafy grającej	359
Przydzielanie pamięci	370
System typów języka Ruby	371
Tworzenie rozszerzeń	373
Zagnieżdżanie interpretera Ruby	380
Współpraca Ruby z innymi językami	384
API języka C służące do komunikacji z Ruby	384

Część III Skrytalizowany Ruby

Rozdział 22. Język Ruby	399
Układ kodu źródłowego	399
Typy podstawowe	402
Nazwy	413
Zmienne i stałe	416
Wyrażenia	427
Definicje metod	437
Wywoływanie metod	440
Tworzenie nowych nazw	443
Definiowanie klas	444
Definiowanie modułów	446
Kontrola dostępu	449
Bloki, domknięcia i obiekty Proc	449
Wyjątki	454
Instrukcje catch i throw	456

Rozdział 23. Dynamiczne określanie typów	459
Klasy to nie typy	461
Programowanie dynamiczne	466
Protokoły standardowe i koercje	467
Od teorii do praktyki	474
Rozdział 24. Klasy i obiekty	475
Współdziałanie klas i obiektów	476
Definicje klas i modułów	483
Środowisko wykonania najwyższego poziomu	490
Dziedziczenie i widoczność	491
Zamrażanie obiektów	492
Rozdział 25. Zabezpieczenia w języku Ruby	495
Poziomy zabezpieczeń	496
Niepewne obiekty	499
Rozdział 26. Refleksja, klasa ObjectSpace i programowanie	
rozproszone	503
Analiza obiektów	504
Analiza klas	506
Dynamiczne wywoływanie metod	508
Systemowe punkty zaczepienia	511
Śledzenie przebiegu programów	514
Szeregowanie i programowanie rozproszone w języku Ruby	517
W czasie kompilacji? W czasie wykonywania programu? Zawsze!	523

Część IV Biblioteka języka Ruby

Rozdział 27. Wbudowane klasy i moduły	527
Porządek alfabetyczny	528
Array	534
Bignum	552
Binding	556
Class	557
Comparable	559

Continuation	561
Dir	563
Enumerable	570
Errno	577
Exception	578
FalseClass	582
File	583
File::Stat	599
FileTest	607
Fixnum	608
Float	612
GC	616
Hash	617
Integer	629
IO	632
Kernel	650
Marshal	676
MatchData	679
Math	683
Method	686
Module	689
NilClass	708
Numeric	710
Object	718
ObjectSpace	732
Proc	734
Process	737
Process::GID	745
Process::Status	747
Process::Sys	750
Process::UID	753
Range	755
Regexp	759
Signal	764

String	766
Struct	793
Struct::Tms	798
Symbol	799
Thread	801
ThreadGroup	811
Time	813
TrueClass	824
UnboundMethod	825
Rozdział 28. Biblioteka standardowa	827
Abbrev	830
Base64	831
Benchmark	832
BigDecimal	834
CGI	836
CGI::Session	839
Complex	840
CSV	841
Curses	843
Date/DateTime	844
DBM	846
Delegator	848
Digest	849
DL	850
dRuby	852
English	854
Enumerator	855
erb	857
Etc	860
expect	861
Fcntl	862
FileUtils	863
Find	865
Forwardable	866

ftools	867
GDBM	868
Generator	870
GetoptLong	871
GServer	873
Iconv	874
IO/Wait	876
IPAddr	877
jcode	878
Logger	879
Mail	881
mathn	883
Matrix	885
Monitor	886
Mutex	888
Mutex_m	890
Net::FTP	891
Net::HTTP	892
Net::IMAP	895
Net::POP	896
Net::SMTP	898
Net::Telnet	899
NKF	901
Observable	902
open-uri	904
Open3	906
OpenSSL	907
OpenStruct	909
OptionParser	910
ParseDate	913
Pathname	914
PP	916
PrettyPrint	918
Profile	920

Profiler_..	921
PStore ..	922
PTY ..	924
Rational ..	925
readbytes ..	926
Readline ..	927
Resolv ..	929
REXML ..	931
Rinda ..	934
RSS ..	936
Scanf ..	938
SDBM ..	940
Set ..	941
Shellwords ..	943
Singleton ..	944
SOAP ..	945
Socket ..	946
StringIO ..	948
StringScanner ..	950
Sync ..	952
Syslog ..	954
Tempfile ..	956
Test::Unit ..	957
thread ..	959
ThreadsWait ..	961
Time ..	962
Timeout ..	963
Tk ..	964
tmpdir ..	966
Tracer ..	967
TSort ..	969
un ..	971
URI ..	972
WeakRef ..	973

WEBrick	975
Win32API	976
WIN32OLE	978
XMLRPC	979
YAML	981
Zlib	982

Dodatki

Dodatek A Biblioteki do obsługi gniazd	987
BasicSocket	988
Socket	991
IPSocket	996
TCPSocket	998
SOCKSSocket	999
TCPSTerver	1000
UDPSocket	1001
UNIXSocket	1003
UNIXServer	1005
Dodatek B Biblioteka MKMF	1007
mkmf	1007
Dodatek C Pomoc techniczna	1011
Witryny internetowe	1012
Witryny z materiałami do pobrania	1013
Grupy dyskusyjne	1013
Listy korespondencyjne	1013
Dodatek D Bibliografia	1015
Dodatek E Tabele	1017
Skorowidz	1027

Rozdział 8.

Zgłaszanie i przechwytywanie wyjątków

Jak do tej pory tworzyliśmy nasz kod w mitycznym Dobrzyńcu, cudownym miejscu, gdzie nic nigdy, ale to nigdy nie sprawia problemów. Każde wywołanie biblioteki kończy się sukcesem, użytkownik nigdy nie wprowadza niepoprawnych danych, a zasobów jest pod dostatkiem i są tanie. Zaraz to zmienimy. Witamy w świecie rzeczywistym!

W świecie rzeczywistym zdarzają się błędy. Dobrzy programiści (i zwykli programiści) spodziewają się ich i starają się je obsłużyć. Nie jest to zawsze takie proste, jak może się to wydawać. Często kod, który wykryje błąd, nie ma wystarczającej znajomości jego kontekstu, aby cokolwiek z nim zrobić. Na przykład otwieranie pliku, który nie istnieje, jest czasem dopuszczalne, ale niekiedy prowadzi do błędu krytycznego. Jak ma reagować nasz moduł obsługi plików?

Tradycyjne podejście to zwracanie kodów błędów. Metoda `open` zwraca jakąś określoną wartość, aby powiedzieć, że otwarcie pliku się nie powiodło. Ta wartość jest następnie przekazywana do wyższych warstw wywołań aż do momentu, gdy któraś z nich weźmie odpowiedzialność za błąd.

Problem z tym podejściem polega na tym, że sprawdzanie wszystkich tych kodów błędów może być uciążliwe. Jeśli funkcja wywołuje metodę `open`, następnie `read`, a w końcu `close` (każda z nich może zwrócić błąd), w jaki sposób funkcja może rozróżnić ich kody błędów na podstawie wartości zwróconej do *jej* jednostki wywołującej?

Wyjątki w dużym stopniu rozwiązują ten problem. Pozwalają upakować informacje o błędzie w obiekcie. Wyjątek jest automatycznie przekazywany w górę stosu wywołań, do momentu, gdy system wykonujący nie odnajdzie kodu, który wprost deklaruje, że wie, jak obsłużyć dany typ wyjątku.

Klasa `Exception`

Paczka, która zawiera informacje o wyjątku, jest obiektem klasy `Exception` lub jednej z jej potomków. Ruby definiuje gotową, uporządkowaną hierarchię wyjątków przedstawioną na rysunku 8.1. Jak przekonamy się dalej, hierarchia ta czyni obsługę wyjątków znacznie prostszą.

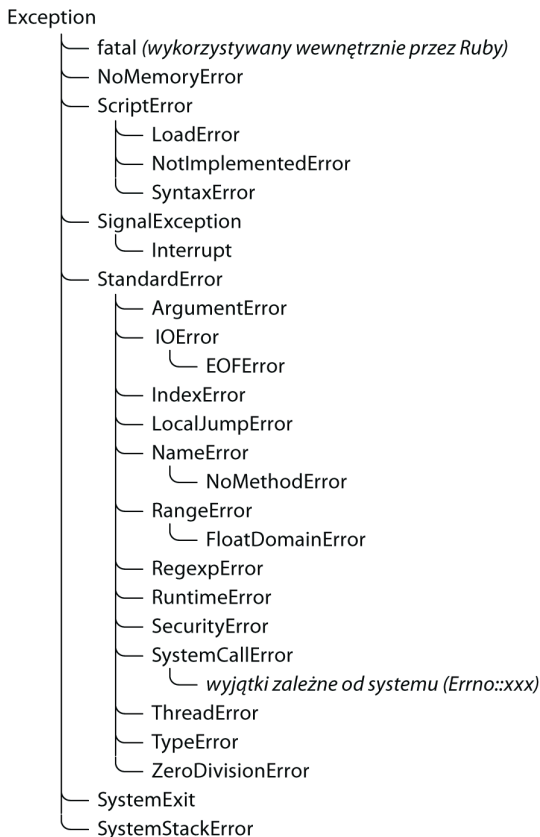
Gdy chcemy użyć wyjątku, możemy wykorzystać jedną z wbudowanych klas `Exception` lub stworzyć własną. Tworząc własną klasę, możemy dziedziczyć po klasie `StandardError` lub jednym z jej potomków. Jeśli tego nie zrobimy, nasze wyjątki nie będą domyślnie przechwytywane.

Każdy wyjątek klasy `Exception` zawiera łańcuch znaków i stos wywołań programu. Jeśli definiujemy własne wyjątki, możemy dołączyć dodatkowe informacje.

Obsługa wyjątków

Nasza szafa grająca pobiera utwory z internetu za pomocą gniazd TCP. Podstawowy kod jest prosty (zakładając, że plik i gniazdko są już utworzone).

```
op_file = File.open(opfile_name, "w")
while data = socket.read(512)
  op_file.write(data)
end
```

**Rysunek 8.1.** Hierarchia wyjątków Ruby

Co się stanie, jeśli pojawi się błąd krytyczny w połowie pobierania pliku? Z pewnością nie chcemy umieszczać niekompletnych utworów na naszej liście.

Dodajmy trochę kodu obsługującego wyjątki i zobaczmy, w jaki sposób rozwiązuje to nasz problem. Aby przechwytywać wyjątki, zawieramy kod, który może je zgłaszać, w bloku `begin-end`. Zastosujmy jedną lub więcej klauzul `rescue`, aby powiedzieć Ruby, jakie typy wyjątków chcemy obsługiwać. W tym konkretnym przypadku jesteśmy zainteresowani przechwytywaniem wyjątków `SystemCallError` (i przez to także wszystkich ich potomków), dlatego właśnie jego umieszczamy w wierszu `rescue`. W bloku obsługi błędu informujemy o błędzie, zamykamy i usuwamy powstały plik oraz ponownie zgłaszamy wyjątek.

```
op_file = File.open(opfile_name, "w")
begin
  # Wyjątki zgłaszane przez ten kod będą
  # przechwytywane przez klauzulę rescue
  while data = socket.read(512)
    op_file.write(data)
  end

  rescue SystemCallError
    $stderr.print "Błąd IO: " + $!
    op_file.close
    File.delete(opfile_name)
    raise
  end
```

Gdy zgłaszany jest wyjątek, niezależnie od jakichkolwiek następujących później przechwyceń, Ruby umieszcza odwołanie do odpowiedniego obiektu `Exception` w globalnej zmiennej `$!` (wykrzyknik doskonale oddaje nasze zaskoczenie tym, że każdy *nasz* kod może powodować błędy). W kodzie z poprzedniego listingu użyliśmy zmiennej `$!` do sformułowania naszego komunikatu o błędzie.

Po zamknięciu i usunięciu pliku wywołujemy `raise` bez parametrów, co powoduje ponowne zgłoszenie wyjątku `$!`. To użyteczna technika, ponieważ pozwala nam na pisanie kodu, który odfiltrowuje wyjątki, zgłaszając ponownie te, których nie umiemy obsłużyć do wyższych warstw. To niemal jak implementowanie hierarchii dziedziczenia dla przetwarzania błędów.

Możemy stosować wiele klauzul `rescue` w bloku `begin`, a każda z nich może precyzować wiele wyjątków do wyłapania. Na końcu każdej klauzuli `rescue` możemy podać Ruby nazwę lokalnej zmiennej, do której ma być przypisany wyjątek. Wielu uważa, że jest to bardziej czytelne niż stosowanie wszędzie `$!`.

```
begin
  eval string
  rescue SyntaxError, NameError => boom
    print "Łańcuch znaków nie jest kompilowany: " + boom
  rescue StandardError => bang
    print "Błąd podczas wykonywania skryptu: " + bang
  end
```

W jaki sposób Ruby decyduje, którą klauzulę wykonać? Okazuje się, że przetwarzanie jest podobne do tego znanego z instrukcji `case`. Dla każdej klauzuli `rescue` w bloku `begin` Ruby porównuje zwrócony wyjątek z każdym z parametrów po kolei. Jeśli zwrócony wyjątek pasuje do parametru, Ruby wykonuje ciało `rescue` i zaprzestaje dalszych sprawdzeń. Spraw-

1.8. dzenie jest dokonywane za pomocą `parametr===$!`. Dla większości wyjątków oznacza to, że sprawdzenie będzie zakończone sukcesem, jeśli wyjątek wymieniony w klauzuli `rescue` jest tego samego typu, co aktualnie zgłoszony, lub jest jego przodkiem¹. Jeśli zapisujemy klauzulę `rescue` bez listy parametrów, domyślnym parametrem jest `StandardError`.

Jeśli żadna klauzula `rescue` nie pasuje lub gdy wyjątek jest zgłaszany poza blokiem `begin-end`, Ruby poszukuje kodu przechwytyującego wyjątku wyżej w stosie, w jednostce wywołującej, a potem jednostce wywołującej tę jednostkę itd.

Mimo że parametry klauzuli `rescue` są zwykle nazwami klas `Exception`, w zasadzie mogą to być dowolne wyrażenia (także wywołania metod) zwracające klasę `Exception`.

Błędy systemowe

1.8. Błędy systemowe są zgłaszane, gdy wywołanie systemu operacyjnego zwraca kod błędu. W systemach POSIX te błędy mają nazwy podobne do `EAGAIN` czy `EPERM`. W systemach uniksowych możemy uzyskać listę tych błędów poleceniem **man errno**.

Ruby opakowuje każdy z tych błędów w oddzielny obiekt wyjątku. Każdy z nich jest podklasą `SystemCallError` i każdy jest zdefiniowany w module nazwanym `Errno`. Oznacza to, że spotkamy się z wyjątkami o nazwach klasy typu `Errno::EAGAIN`, `Errno::EIO` czy `Errno::EPERM`. Jeśli chcemy uzyskać oryginalny kod błędu, pamiętajmy, że obiekty wyjątków `Errno` posiadają specjalną stałą klasy nazwaną (jakże niespodziewanie) `Errno`, która przechowuje tę wartość.

```
Errno::EAGAIN::Errno → 35
Errno::EPERM::Errno  →  1
Errno::EIO::Errno   →  5
Errno::EWOULDBLOCK::Errno → 35
```

Zwróćmy uwagę, że `EWOULDBLOCK` i `EAGAIN` mają ten sam numer błędu. To cecha systemu operacyjnego komputera, którego użyliśmy do tworzenia tej książki — obie stałe odpowiadają temu samemu kodowi błędu. Aby sobie z tym poradzić, Ruby tak wszystko aranżuje, aby `Errno::EAGAIN` i `Errno::`

¹ To porównanie jest możliwe, ponieważ wyjątki są klasami, a klasy z kolei są rodzajem modułu. Metoda `===` dla modułów zwraca `true`, jeśli klasa operandu jest taka sama lub należy do przodków odbiorcy.

`EWOULDBLOCK` były traktowane tak samo przez klauzulę `rescue`. Jest to realizowane poprzez zdefiniowanie `SystemCallError#===` w taki sposób, że porównanie dwóch podklas `SystemCallError` powoduje porównywanie ich kodów błędów, a nie ich pozycji w hierarchii.

Sprzątanie

Czasem musimy zagwarantować, że dane przetwarzanie jest dokonywane na końcu bloku kodu, niezależnie od tego, czy zgłaszany jest wyjątek. Na przykład możemy otworzyć plik przy wejściu do bloku i musimy się upewnić, że zostanie on zamknięty wraz z zakończeniem bloku.

Służy temu klauzula `ensure`, która znajduje się za ostatnią klauzulą `rescue` i zawiera kawałek kodu, który będzie zawsze wykonany wraz z zakończeniem bloku. Nie ma znaczenia, czy blok kończy pracę normalnie czy został przerwany przez nieprzechwycony wyjątek — blok `ensure` będzie zawsze wykonany.

```
f = File.open("testfile")
begin
  # .. przetwarzanie
rescue
  # .. obsługa błędu
ensure
  f.close unless f.nil?
end
```

Klauzula `else` jest podobną, ale mniej użyteczną konstrukcją. Jeśli jest obecna, występuje po klauzuli `rescue`, ale przed `ensure`. Ciało klauzuli `else` jest wykonywane jedynie wówczas, gdy żadne wyjątki nie są zgłaszane w głównym ciele kodu.

```
f = File.open("testfile")
begin
  # .. przetwarzanie
rescue
  # .. obsługa błędu
else
  puts "Gratulacje -- nie ma błędów!"
ensure
  f.close unless f.nil?
end
```

Zagraj to jeszcze raz

Czasem możemy być w stanie naprawić przyczynę wyjątku. W takich przypadkach używamy instrukcji `retry` w klauzuli `rescue` do ponownego wywołania całego bloku `begin-end`. Może to prowadzić do powstania pętli nieskończonych, dlatego należy zachować ostrożność (i na wszelki wypadek trzymać palce na klawiszach przerywających pracę programu).

Jako przykład kodu, który ponownie wywołuje kod po wystąpieniu wyjątków, przedstawimy nieco zmieniony fragment biblioteki `net/smtp.rb` autorstwa Minero Aoki.

```
@esmtplib = true

begin
  # Najpierw spróbuj zalogować się rozszerzonym loginem. Jeśli to się powiedzie,
  # ponieważ serwer go nie obsługuje, spróbuj standardowego sposobu logowania.

  if @esmtplib then
    @command.ehlo(helodom)
  else
    @command.helo(helodom)
  end

  rescue ProtocolError
    if @esmtplib then
      @esmtplib = false
      retry
    else
      raise
    end
  end
end
```

Kod najpierw próbuje połączyć się z serwerem SMTP za pomocą polecenia EHL0, które nie jest zawsze obsługiwane. Jeśli próba połączenia zakończy się niepowodzeniem, kod ustawia wartość zmiennej `@esmtplib` na `false` i próbuje ponownie nawiązać połączenie. Jeśli ponownie wystąpi błąd, do jednostki wywołującej zgłaszany jest wyjątek.

Zgłaszanie wyjątków

Jak do tej pory byliśmy w obronie i obsługiwaliśmy wyjątki zgłaszane przez innych. Czas odwrócić sytuację i przejść do natarcia. Niektórzy mówią, że autorzy tej niniejszej publikacji zawsze są w natarciu, ale to już temat na inną książkę.

Możemy zgłaszać wyjątki w naszym kodzie za pomocą metody `Kernel.raise` (lub jej synonimu — `Kernel.fail`).

```
raise
raise "nieprawidłowe kodowanie mp3"
raise InterfaceException, "Błąd klawiatury", caller
```

Pierwsza postać ponownie zgłasza bieżący wyjątek (lub `RuntimeError`, jeśli nie ma bieżącego wyjątku). Wykorzystujemy ją w kodzie obsługi wyjątków, które muszą przechwycić wyjątek, a następnie przekazać go dalej.

Druga postać tworzy nowy wyjątek `RuntimeError` i ustawia jego komunikat na podany łańcuch znaków. Wyjątek ten jest następnie przekazywany w górę stosu wywołań.

Trzecia postać wykorzystuje pierwszy argument do utworzenia wyjątku, następnie ustawia jego komunikat na podany w drugim argumencie łańcuch znaków, a ślad stosu na trzeci argument. Zwykle pierwszy argument będzie albo klasą z hierarchii `Exception`, albo odwołaniem do egzemplarza obiektu jednej z tych klas². Ślad stosu jest zwykle zwracany przez metodę `Kernel.caller`.

Oto kilka typowych zastosowań `raise`:

```
raise

raise "Brak nazwy" if name.nil?

if i >= names.size
  raise IndexError, "#{i} >= size (#{names.size})"
end

raise ArgumentError, "Za długa nazwa", caller
```

W kodzie pokazanym na ostatnim listingu usuwamy bieżące wywołanie ze stosu programu, co jest często użyteczne w modułach bibliotek. Możemy pójść dalej — poniższy kod usuwa dwa wywołania ze stosu programu poprzez przekazanie do nowego wyjątku tylko podzbioru stosu wywołań.

```
raise ArgumentError, "Za długa nazwa", caller[1..-1]
```

² Technicznie ten argument może być obiektem, który odpowiada na komunikat `exception`, zwracając obiekt, dla którego wywołanie `object.kind_of?(Exception)` zwraca prawdę.

Rozszerzanie informacji o wyjątku

Mozemy definiować własne wyjątki przechowujące informacje, które chcemy przekazać z miejsca wystąpienia błędu. Na przykład niektóre typy błędów sieciowych mogą być chwilowe, zależnie od okoliczności. Jeśli wystąpi taki błąd, a warunki są poprawne, możemy ustawić flagę w wyjątku, która powiadomi kod przechwytyjący, że być może warto ponowić próbę.

```
class RetryException < RuntimeError
  attr :ok_to_retry
  def initialize(ok_to_retry)
    @ok_to_retry = ok_to_retry
  end
end
```

Gdzieś głęboko w kodzie pojawia się chwilowy błąd.

```
def read_data(socket)
  data = socket.read(512)
  if data.nil?
    raise RetryException.new(true), "chwilowy błąd odczytu"
  end
  # .. normalne przetwarzanie
end
```

Przechwytyjemy wyjątek wyżej w stosie wywołań.

```
begin
  stuff = read_data(socket)
  # .. przetwarzaj
rescue RetryException => detail
  retry if detail.ok_to_retry
  raise
end
```

catch i throw

Co prawda mechanizm wyjątków (`raise` i `rescue`) doskonale się sprawdza podczas przerywania wykonywania, gdy pojawiają się problemy, ale czasem byłoby dobrze mieć możliwość przerywania wielokrotnie zagnieżdżonej konstrukcji podczas prawidłowego przetwarzania. Tu właśnie przydatne są `catch` i `throw`.

```
catch (:done) do
  while line = gets
    throw :done unless fields = line.split(/\t/)
    songlist.add(Song.new(*fields))
  end
end
```

```
end
songlist.play
end
```

`catch` definiuje blok, któremu nadana jest nazwa (może to być symbol lub łańcuch znaków). Ten blok jest wykonywany normalnie aż do napotkania `throw`.

Napotykać `throw`, Ruby zwija z powrotem stos programu w poszukiwaniu wywołania bloku `catch` o pasującym symbolu. Gdy go odnajduje, zwija stos do tego punktu i kończy blok. Tak więc na poprzednim listingu `throw` przejdzie na koniec odpowiadającego jej wywołania `catch`, nie tylko kończąc w ten sposób pętlę `while`, ale także pomijając odegranie listy utworów. Gdy `throw` jest wywoływane z opcjonalnym, drugim parametrem, wartość ta jest zwracana jako wartość wyrażenia `catch`.

Kolejny listing stosuje `throw` do zakończenia interaktywnej pracy z użytkownikiem, gdy w odpowiedzi na dowolne pytanie wpisze on znak `!`.

```
def prompt_and_get(prompt)
  print prompt
  res = readline.chomp
  throw :quit_requested if res == "!"
  res
end

catch :quit_requested do
  name = prompt_and_get("Name: ")
  age = prompt_and_get("Age: ")
  sex = prompt_and_get("Sex: ")
  # ..
  # przetwarzaj informacje
end
```

Jak widać na powyższym przykładzie, `throw` nie musi występować w statycznej przestrzeni `catch`.