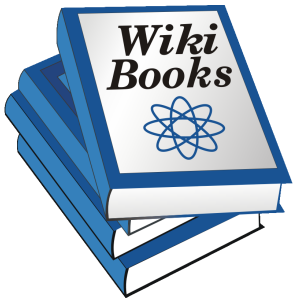
Ruby



Stworzone na Wikibooks,
bibliotece wolnych podręczników.

Wydanie I z dnia 17 lutego 2008

Spis treści

Rozdział 1

O podręczniku

Ruby to interpretowany, w pełni obiektowy język programowania stworzony przez Yukihiro Matsumoto (pseudonim *Matz*). W języku angielskim *ruby* oznacza rubin.

Od roku 2003 lawinowo zdobywa nowych zwolenników, głównie za sprawą popularnego frameworku do tworzenia aplikacji webowych o nazwie Ruby on Rails, tworzonego przez grupę programistów pod kierownictwem Davida Heinemeiera Hanssona.

W roku 2005 według statystyk sklepu Amazon dwie najpopularniejsze książki na temat **Rubiego** i Ruby on Rails były najlepiej sprzedawanymi pozycjami z kategorii *Programowanie*.

Ruby bazuje na wielu językach, takich jak Perl, Smalltalk, Python, CLU, Eiffel czy LISP. Składnia jest zorientowana liniowo i oparta na składni CLU i, w mniejszym stopniu, Perla.

Punktem wyjścia dla niniejszego podręcznika stała się książka Marka Slagella "Ruby's User Guide" udostępniona na licencji GNU Free Documentation License.

Rozdział 2

Czym jest Ruby?

Czym jest Ruby?

Ruby jest "interpretowanym językiem skryptowym do szybkiego i prostego programowania zorientowanego obiektowo" – co to znaczy?

- interpretowany język skryptowy:
  - możliwość bezpośrednich wywołań systemowych
  - potężne operacje na łańcuchach znakowych i wyrażeniach regularnych
  - natychmiastowa informacja zwrotna podczas rozwoju oprogramowania
- szybki i prosty:
  - deklaracje zmiennych są niepotrzebne
  - zmienne nie są typizowane
  - składnia jest prosta i konsekwentna
  - zarządzanie pamięcią jest automatyczne
- programowanie zorientowane obiektowo:
  - wszystko jest obiektem
  - klasy, metody, dziedziczenie, itd.
  - metody singletonowe
  - domieszkowanie dzięki modułom
  - iteratory i domknięcia
- a ponadto:
  - liczby całkowite dowolnej precyzji
  - wygodne przetwarzanie wyjątków
  - dynamiczne ładowanie
  - wsparcie dla wielowątkowości

Jeśli jakieś pojęcia wydają ci się obce, czytaj dalej i nie martw się. Mantra języka Ruby to *szybko i prosto.*

Rozdział 3

Zaczynamy

Zaczynamy

Najpierw pewnie chcesz sprawdzić, czy masz zainstalowanego Rubiego. W konsoli (znak zachęty oznaczony jest tutaj jako %, oczywiście nie wpisujemy go), wpisz:

`% ruby -v`

(`-v` powoduje wypisanie na ekranie wersji Rubiego), następnie naciśnij Enter. Jeśli Ruby jest zainstalowany powinieneś zobaczyć podobną do tej wiadomość:

```
% ruby -v
ruby 1.8.3 (2005-09-21) [i586-linux]
```

lub

`ruby 1.8.6 (2007-03-13 patchlevel 0) [i386-mswin32]`

Jeśli Ruby nie jest zainstalowany, możesz poprosić administratora o jego zainstalowanie, lub zrobić to samemu, ponieważ Ruby jest darmowym oprogramowaniem bez ograniczeń co do instalacji czy użytkowania. Wszystkie ważne informacje dotyczące instalacji, również w języku polskim, znajdziesz na stronie głównej Rubiego.

Zacznijmy zabawę z Rubim. Możesz umieścić program w Rubim bezpośrednio w linii poleceń używając opcji *-e*:

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

Konwencjonalnie, program w Rubim można umieścić w pliku.

```
% echo "puts 'witaj swiecie'" > hello.rb
% ruby hello.rb
witaj swiecie
```

Jeśli masz zamiar pisać bardziej rozbudowany kod, będziesz musiał użyć prawdziwego edytora tekstu!

Niektóre zadziwiająco złożone i przydatne rzeczy mogą zostać zrobione przy użyciu miniaturowych programów, które mogą zmieścić się w jednej linijce linii komend. Np., taki, który zamienia *foo* na *bar* we wszystkich plikach źródłowych i nagłówkowych języka C w bieżącym katalogu, dodając do plików oryginalnych rozszerzenie ".bak".

`% ruby -i.bak -pe 'sub "foo", "bar"' *.[ch]`

Ten program działa jak polecenie *cat* w UNIX'ie (ale wolniej niż *cat*):

`% ruby -pe 0 file`

Rozdział 4

Proste przykłady

Proste przykłady

Napiszmy funkcję obliczającą silnię. Matematyczna definicja silni `n` to:

```
n! = 1                (gdy n==0)
   = n * (n-1)!       (w innym przypadku)
```

W Rubim możemy ją napisać w następujący sposób[1]:

```
def silnia(n)
  if n == 0
    1
  else
    n * silnia(n-1)
  end
end
```

Warto zauważyć powtarzające się wyrażenie `end`. Ruby nazywany jest przez to "algolopodobnym" językiem programowania. Właściwie, składnia Rubiego bardziej przypomina inny język — Eiffel. Można także zauważyć brak wyrażeń `return`. Nie są one potrzebne, ponieważ funkcja w Rubim zwraca ostatnią wartość, która była w niej liczona. Używanie wyrażenia `return` jest dozwolone, lecz niepotrzebne.

Wypróbujmy naszą funkcję silni. Dodanie jednej linii kodu daje nam działający program:

```
# Program, który liczy wartość silni z danej liczby
# Zapisz go jako silnia.rb

def silnia(n)
  if n == 0
```

[1] Aby ułatwić zrozumienie, kod źródłowy we wszystkich przykładach został przetłumaczony na język polski (identyfikatory, łańcuchy znakowe, komentarze). Ponieważ Ruby w wersji stabilnej nie obsługuje jeszcze w pełni standardu Unicode, polskie znaki diaktryczne zostały zamienione na ich łacińskie odpowiedniki (*ś* na *s*, *ł* na *l*, itd...). Należy jednak wspomnieć, że powszechnie przyjęte i zalecane jest stosowanie języka angielskiego przy tworzeniu kodu niemalże w każdym języku programowania.

```
    1
  else
    n * silnia(n-1)
  end
end

puts silnia(ARGV[0].to_i)
```

`ARGV` jest tutaj tablicą zawierającą argumenty linii poleceń, a `to_i` konwertuje łańcuch na liczbę całkowitą.

```
% ruby silnia.rb 1
1
% ruby silnia.rb 5
120
```

Czy nasz program zadziała z argumentem 40? Nie policzymy tego na kalkulatorze...

```
% ruby silnia.rb 40
815915283247897734345611269596115894272000000000
```

Działa! Ruby może sobie poradzić z każdą liczbą całkowitą, która zmieści się w pamięci naszego komputera. Więc możemy policzyć nawet 400!:

```
% ruby silnia.rb 400
64034522846623895262347970319503005850702583026002959458684
44594280239716918683143627847864746326467629435057503585681
08482981628835174352289619886468029979373416541508381624264
61942352307046244325015114448670890662773914918117331955996
44070954967134529047702032243491121079759328079510154537266
72516278778900093497637657103263503315339653498683868313393
52024373788157786791506311858702618270169819740062983025308
59129834616227230455833952075961150530223608681043329725519
48526744322324386699484224042325990555161063594237696139923
31917134063858996537970147827206606320217379472010321356624
61380907794230459736069956759583609615871512991382228657857
95493616176544804532220078258184008484364155912294542753848
03558374518022675900061399560145595206127211192918105032491
00800000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
```

Nie możemy natychmiastowo sprawdzić poprawności obliczeń, ale muszą być prawidłowe. :-)

Pętla wejście/obliczenie i sposób prezentacji przykładów

Kiedy wywołasz Rubiego bez argumentów, interpreter wczyta komendy ze standardowego wejścia i wykona je po zakończeniu wprowadzania:

```
% ruby
puts "witaj swiecie"
puts "zegnaj swiecie"
```

```
^D
witaj swiecie
zegnaj swiecie
```

Znak `^D` powyżej oznacza *Ctrl+D*, wygodny sposób sygnalizowania, że wprowadzanie zostało zakończone w systemach uniksowych. W DOS/Windows spróbuj użyć F6 lub ^Z.

Ruby zawiera również program zwany *irb*, który pomaga wprowadzać kod bezpośrednio z klawiatury w interaktywnej pętli, pokazując na bieżąco rezultaty.

Oto krótka sesja z *irb*:

```
% irb
irb(main):001:0> puts "Witaj, swiecie."
Witaj, swiecie.
=> nil
irb(main):002:0> exit
```

"Witaj swiecie" jest wypisane przez `puts`. Następna linia, w tym przypadku `nil`, pokazuje cokolwiek, co zostało obliczone jako ostatnie. Ruby nie rozróżnia instrukcji i wyrażeń, więc obliczanie kawałka kodu jest równoważne z jego wykonaniem. Tutaj, `nil` oznacza, że `puts` nie zwraca żadnej (znaczącej) wartości. Zauważ, że możemy opuścić pętlę interpretera przez wpisanie `exit`.

W naszym podręczniku będziemy korzystać z programu *irb* oraz z przykładów zapisanych bezpośrednio jako kod źródłowy. Rezultat działania takiego kodu (wyjście), będziemy podawać jako komentarz, stosując oznaczenie: `#=>`. (Co to jest komentarz możesz sprawdzić tutaj.) Alternatywnie, będziemy czasem przedstawiać sesję z naszym kodem jako zapis okna terminala.

Rozdział 5

Łańcuchy znakowe

Łańcuchy znakowe

Ruby radzi sobie z łańcuchami znakowymi tak dobrze jak z danymi numerycznymi. Łańcuch może być ograniczony znakami podwójnego cudzysłowu (‘‘...’’) lub pojedynczego (apostrof) (’...’).

```
irb(main):001:0> "abc"
=> "abc"
irb(main):002:0> ’abc’
=> "abc"
```

Używanie podwójnych lub pojedynczych cudzysłowów czasami może mieć różne efekty. Łańcuch ujęty w podwójny cudzysłów pozwala stosować znaki formatujące za pomocą odwróconego ukośnika oraz obliczać zagnieżdżone wyrażenia używając #\\. Łańcuch ujęty w apostrofy nie pozwala na taką interpretację; to co widzisz — to dostajesz. Przykłady:

```
irb(main):001:0> puts "a\nb\nc"
a
b
c
=> nil
irb(main):002:0> puts ’a\nb\nc’
a\nb\nc
=> nil
irb(main):003:0> "\n"
=> "\n"
irb(main):004:0> ’\n’
=> "\\n"
irb(main):005:0> "\001"
=> "\001"
irb(main):006:0> ’\001’
=> "\\001"
irb(main):007:0> "abcd #{5*3} efg"
=> "abcd 15 efg"
```

```
irb(main):008:0> var = " abc "
=> " abc "
irb(main):009:0> "1234#{var}5678"
=> "1234 abc 5678"
```

Manipulowanie łańcuchami w Rubim jest sprytniejsze i bardziej intuicyjne niż w C. Dla przykładu, możesz łączyć ze sobą łańcuch używając `+`, a powtarzać łańcuch wiele razy za pomocą `*`:

```
irb(main):001:0> "foo" + "bar"
=> "foobar"
irb(main):002:0> "foo" * 2
=> "foofoo"
```

Konkatenacja łańcuchów w C jest bardziej kłopotliwa z powodu konieczności bezpośredniego zarządzania pamięcią:

```
char *s = malloc(strlen(s1)+strlen(s2)+1);
strcpy(s, s1);
strcat(s, s2);
/* ... */
free(s);
```

W Rubim natomiast, nie musimy w ogóle zastanawiać się nad miejscem zajmowanym w pamięci przez łańcuch. Jesteśmy wolni od jakiegokolwiek zarządzania pamięcią.

Oto kilka rzeczy, które możesz zrobić z łańcuchami.

Konkatenacja:

```
slowo = "fo" + "o"
puts slowo #=> "foo"
```

Powtórzenie:

```
slowo = slowo * 2
puts slowo #=> "foofoo"
```

Ekstrahowanie znaków (zauważ, że znaki w Rubim są liczbami całkowitymi):

```
puts slowo[0]     #=> 102
# 102 jest kodem ASCII znaku ‘f’

puts slowo[-1]    #=> 111
# 111 jest kodem ASCII znaku ‘o’
```

(Wartości ujemne oznaczają liczbę znaków od końca łańcucha.)

Ekstrahowanie podłańcuchów:

```
warzywo = "pietruszka"
puts warzywo[0,1]    #=> "p"
puts warzywo[-2,2]   #=> "ka"
puts warzywo[0..3]   #=> "piet"
puts warzywo[-5..-2] #=> "uszk"
```

Sprawdzanie równości:

```
puts "foo" == "foo" #=> true
puts "foo" == "bar" #=> false
```

Zróbmy użytek z kilku tych cech. Oto zgadywanka "co to za słowo", ale być może słowo "zgadywanka" to zbyt dużo dla tego kodu. ;-)

```
# zapisz to jako zgadnij.rb
slowa = ['fiolek', 'roza', 'bez']
sekret = slowa[rand(3)]

print "zgadniesz? "
while odp = STDIN.gets
  odp.chop!
  if odp == sekret
    puts "Wygrales!"
    break
  else
    puts "Przykro mi, przegrales."
  end
  print "zgadniesz? "
end
puts "Chodzilo o ", sekret, "."
```

Na razie nie przejmuj się za bardzo szczegółami powyższego kodu. Oto jak wygląda uruchomiona łamigłówka.

```
% ruby zgadnij.rb
zgadniesz? fiolek
Przykro mi, przegrales.
zgadniesz? bez
Przykro mi, przegrales.
zgadniesz? ^D
Chodzilo o roza.
```

(Mogło nam pójść nieco lepiej biorąc pod uwagę, że szansa na trafienie wynosi 1/3.)

Rozdział 6

Wyrażenia regularne

Wyrażenia regularne

Stwórzmy bardziej interesujący program. Tym razem sprawdzimy czy łańcuch pasuje do opisu zakodowanego jako ścisły wzorzec.

Oto pewne znaki i kombinacje znaków które mają specjalne znaczenie w tych wzorcach:

Kombinacja	Opis
`[]`	specyfikacja zakresu (np., [a-z] oznacza litery od a do z)
`\w`	litera lub cyfra; to samo co [0-9A-Za-z]
`\W`	ani litera ani cyfra
`\s`	biały znak; to samo co [\t\n\r\f]
`\S`	nie biały znak
`\d`	znak cyfra; to samo co [0-9]
`\D`	znak nie będący cyfrą
`\b`	backspace (0x08) (tylko jeśli występuje w specyfikacji zakresu)
`\b`	granica słowa (jeśli nie występuje w specyfikacji zakresu)
`\B`	granica nie słowa
`*`	treść stojąca przed tym symbolem może powtórzyć się zero lub więcej razy
`+`	treść stojąca przed tym symbolem musi powtórzyć się jeden lub więcej razy
`\m,n\`	treść stojąca przed tym symbolem musi powtórzyć się od m do n razy
`?`	treść stojąca przed tym symbolem może wystąpić najwyżej jeden raz; to samo co \0,1\
`|`	albo treść stojąca przed tym symbolem albo następne wyrażenie muszą pasować
`()`	grupowanie
`^`	początek wiersza
`$`	koniec wiersza

Wspólny termin określający wzory, które używają tych dziwnych symboli, to *wyrażenia regularne*. W Rubim tak samo jak w Perlu bierze się je raczej w ukośniki (`/`) niż w cudzysłowy. Jeżeli nigdy wcześniej nie pracowałeś z wyrażeniami regularnymi, prawdopodobnie nie wyglądają one zbyt regularnie, ale naprawdę warto poświęcić trochę czasu by się z nimi zaznajomić. Wyrażenia regularne są skuteczne i ekspresywne. Oszczędzi ci to bólów głowy (i wielu linii kodu) niezależnie od tego, czy potrzebujesz dopasowywania wzorców, wyszukiwania czy też innego manipulowania łańcuchami.

Dla przykładu, przypuśćmy, że chcemy sprawdzić czy łańcuch pasuje do tego opisu: "Zaczyna się małą literą *f*, po której zaraz następuje jedna duża litera i opcjonalnie jakieś inne znaki, aż do wystąpienia innych innych małych liter." Jeżeli jesteś doświadczonym programistą C, prawdopodobnie już napisałeś około tuzina linii kodu w głowie, prawda? Przyznaj, prawie nie możesz sobie poradzić. Ale w Rubim potrzebujesz jedynie by łańcuch został sprawdzony pod kątem występowania wyrażenia regularnego `/^f[A-Z][^a-z]*$/`.

A co z "zawiera liczbę heksadecymalną w ostrych nawiasach"? Żaden problem.

```
def ma_hex?(s)                        # "zawiera hex w ostrych nawiasach"
  (s =~ /<0(x|X)(\d|[a-f]|[A-F])+>/) != nil
end

puts ma_hex?("Ten nie ma.") #=> false

puts ma_hex?("Może ten? {0x35}") #=> false
# (zły rodzaj nawiasów)

puts ma_hex?("Albo ten? <0x38z7e>") #=> false
# fałszywa liczba hex

puts ma_hex?("Dobra, ten: <0xfc0004>.") #=> true
```

Chociaż wyrażenia regularne mogą się wydawać na początku nieco zagadkowe, z pewnością szybko osiągniesz satysfakcję z tak ekonomicznego sposobu wyrażania skomplikowanych pomysłów.

Oto mały program który pomoże ci eksperymentować z wyrażeniami regularnymi. Zapisz go jako *regx.rb* i uruchom przez wpisanie `ruby regx.rb` w linii poleceń.

```
# Wymaga terminala ANSI!

st = "\033[7m"
en = "\033[m"

puts "Aby zakonczyc wprowadz pusty tekst."

while true
  print "tekst> "; STDOUT.flush; tekst = gets.chop
  break if tekst.empty?
  print "wzor> "; STDOUT.flush; wzor = gets.chop
  break if wzor.empty?
  wyr = Regexp.new(wzor)
  puts tekst.gsub(wyr,"#{st}\\&#{en}")
end
```

Program wymaga dwukrotnego wprowadzenia danych. Raz oczekuje na łańcuch tekstowy, a raz na wyrażenie regularne. Łańcuch sprawdzany jest pod kątem występowania wyrażenia regularnego, następnie wypisywany z podświetleniem wszystkich pasujących fragmentów. Nie analizuj teraz szczegółów, analiza tego kodu wkrótce się pojawi.

```
tekst> foobar
wzor> ^fo+
foobar
~~~
```

Znaki tyldy oznaczają podświetlony tekst na wyjściu programu.

Wypróbujmy kilka innych tekstów.

```
tekst> abc012dbcd555
wzor> \d
abc012dbcd555
   ~~~    ~~~
```

Jeśli cię to zaskoczyło, sprawdź w tabeli na górze tej strony: `\d` nie ma żadnego związku ze znakiem *d*, oznacza natomiast pojedynczą cyfrę.

A co, jeśli jest więcej niż jeden sposób poprawnego dopasowania wzoru?

```
tekst> foozboozer
wzor> f.*z
foozboozer
~~~~~~~~
```

`foozbooz` jest spasowany zamiast samego `fooz`, gdyż wyrażenie regularne wybiera najdłuższy, możliwy podłańcuch.

Oto wzór do wyizolowania pola zawierającego godzinę z separatorem w postaci dwukropka.

```
tekst> Wed Feb  7 08:58:04 JST 1996
wzor> [0-9]+:[0-9]+(:[0-9]+)?
Wed Feb  7 08:58:04 JST 1996
           ~~~~~~~~
```

"`=~`" jest operatorem dopasowania w odniesieniu do wyrażeń regularnych; zwraca pozycję w łańcuchu, gdzie może być znaleziony pasujący podłańcuch lub `nil` jeżeli takowy nie występuje.

```
puts "abcdef" =~ /d/ #=> 3
puts "aaaaaa" =~ /d/ #=> nil
```

Rozdział 7

Tablice

Tablice

Możesz stworzyć tablicę przez podanie kilku jej elementów wewnątrz nawiasów kwadratowych (`[]`) oddzielonych przecinkami. W Rubim tablice mogą przyjmować obiekty różniące się typami.

```
irb(main):001:0> tab = [1, 2, "3"]
=> [1, 2, "3"]
```

Tablice mogą być konkatenowane i powtarzane tak jak łańcuchy.

```
irb(main):002:0> tab + ["pla", "bla"]
=> [1, 2, "3", "pla", "bla"]
irb(main):003:0> tab * 2
=> [1, 2, "3", 1, 2, "3"]
```

Możemy używać numerów indeksów aby odnieść się do jakiejkolwiek części tablicy.

```
irb(main):004:0> tab[0]
=> 1
irb(main):005:0> tab[0,2]
=> [1, 2]
irb(main):006:0> tab[0..1]
=> [1, 2]
irb(main):007:0> tab[-2]
=> 2
irb(main):008:0> tab[-2..-1]
=> [2, "3"]
irb(main):009:0> tab[-2,2]
=> [2, "3"]
```

(Wartości ujemne oznaczają położenie elementu od końca tablicy.)

Tablice mogą być konwertowane na łańcuchy tekstowe i odwrotnie poprzez użycie odpowiednio: `join` (*dołącz*) i `split` (*podziel*):

```
irb(main):010:0> tekst = tab.join(":")
=> "1:2:3"
irb(main):011:0> tekst.split(":")
=> ["1", "2", "3"]
```

Wreszcie, aby dodać dodać nowy element do tablicy (tablice w Rubim zachowują się jak listy) można zastosować operator <<:

```
irb(main):006:0> tab << 4
=> [1, 2, "3", 4]
irb(main):007:0> tab << "bla"
=> [1, 2, "3", 4, "bla"]
```

Tablice wielowymiarowe

W języku Ruby można także definiować tablice tablic, przez co można niejako "emulować" ich wielowymiarowość. Spójrzmy na następujący fragment kodu:

```
irb(main):012:0> t = [[1,2],[3,4]]
=> [[1, 2], [3, 4]]
irb(main):013:0> t[1][0]
=> 3
```

Jako wynik na ekranie pojawia się cyfra `3` (pierwszy element drugiej tablicy "wewnętrznej").

Tablice asocjacyjne

Tablica asocjacyjna ma elementy, które są dostępne przez klucze mogące mieć wartość dowolnego rodzaju, a nie przez kolejne numery indeksów. Taka tablica jest czasem nazywana *hash'em* lub słownikiem; w świecie Rubiego preferujemy termin *hash*. Hash (czyt. *hasz*) może być utworzony przez pary "`klucz => wartość`" umieszczone w nawiasach klamrowych (\\). Klucza używa się by odnaleźć coś w haszu, tak jak używa się indeksu by odnaleźć coś w tablicy.

```
irb(main):014:0> h = {1 => 2, "2" => "4"}
=> {1=>2, "2"=>"4"}
irb(main):015:0> h[1]
=> 2
irb(main):016:0> h["2"]
=> "4"
irb(main):017:0> h[5]
=> nil
```

Dodawanie nowego wpisu:

```
irb(main):018:0> h[5] = 10
=> 10
irb(main):019:0> h
=> {5=>10, 1=>2, "2"=>"4"}
```

Kasowanie wpisu przez podanie klucza:

```
irb(main):020:0> h.delete 1
=> 2
irb(main):021:0> h[1]
=> nil
irb(main):022:0> h
=> {5=>10, "2"=>"4"}
irb(main):023:0>
```

Rozdział 8

Powrót do prostych przykładów

Powrót do prostych przykładów

Rozbierzmy na części kod kilku poprzednich przykładowych programów.
Następujący kod pojawił się w rozdziale z prostymi przykładami.

```
def silnia(n)
  if n == 0
    1
  else
    n * silnia(n-1)
  end
end
puts silnia(ARGV[0].to_i)
```

Ponieważ jest to pierwsze objaśnienie, zbadamy każdą linię osobno.

Silnie

`def silnia(n)`

W pierwszej linii, `def` jest instrukcją służącą do definiowania funkcji (lub, bardziej precyzyjnie, metody. O tym, czym jest metoda będziemy mówić więcej w dalszym rozdziale). Tutaj, `def` wskazuje, że funkcja przyjmuje pojedynczy argument, nazwany `n`.

`if n == 0`

`if` służy do sprawdzania warunku. Kiedy warunek jest spełniony, następny fragment kodu jest obliczany. W przeciwnym razie obliczane jest cokolwiek co występuje za `else`.

`1`

Wartość `if` wynosi `1` jeżeli warunek jest spełniony.

`else`

Jeżeli warunek nie jest spełniony, obliczany jest kod znajdujący się od tego miejsca aż do `end`.

`n * silnia(n-1)`

Jeżeli warunek nie jest spełniony, wartość wyrażenia `if` wynosi `n` razy `silnia(n-1)`.

`end`

Pierwszy `end` zamyka instrukcję `if`.

`end`

Drugi `end` zamyka instrukcję `def`.

`puts silnia(ARGV[0].to_i)`

Ta linia wywołuje naszą funkcję `silnia()` używając wartości z linii poleceń oraz wypisuje wynik.

`ARGV` jest tablicą, która zawiera argumenty z linii poleceń. Elementy `ARGV` są łańcuchami znakowymi, więc aby dokonać konwersji na liczby całkowite używamy metody `to_i`. Ruby nie zamienia łańcuchów na liczby automatycznie tak jak Perl.

Co się stanie jeśli podamy naszemu programowi liczbę ujemną? Widzisz problem? Umiesz go rozwiązać?

Łańcuchy znakowe

Teraz zbadamy nasz program — łamigłówkę z rozdziału o łańcuchach znakowych. Ponieważ jest on nieco długi, ponumerujmy linie, by móc się łatwo do nich odwoływać.

```
slowa = ['fiolek', 'roza', 'bez']
sekret = slowa[rand(3)]

print "zgadnij? "
while odp = STDIN.gets
  odp.chop!
  if odp == sekret
    puts "Wygrales!"
    break
  else
    puts "Przykro mi, przegrales."
  end
  print "zgadnij? "
end
puts "Chodzilo o ", sekret, "."
```

W tym programie użyta jest nowa struktura sterująca — `while`. Kod pomiędzy `while` a jej kończącym `end` będzie wykonywany w pętli tak długo jak pewien określony warunek pozostanie prawdziwy. W tym przypadku `odp = STDIN.gets` jest zarówno aktywną instrukcją (pobierającą linię wejściową od użytkownika i zachowującą ją jako `odp`), oraz warunkiem (jeżeli nie ma żadnego wejścia, `odp`, które reprezentuje wartość całego wyrażenia `odp = STDIN.gets`, będzie miało wartość `nil`, która spowoduje przerwanie pętli `while`).

`STDIN` oznacza obiekt standardowego wejścia. Zwykle `odp = gets` robi to samo, co `odp = STDIN.gets`.

`rand(3)` w linii 2 zwraca losową liczbę w przedziale od 0 do 2. Ta losowa liczba jest użyta do wyciągnięcia jednego elementu z tablicy `slowa`.

W linii 5 czytamy jedną linię ze standardowego wejścia przez metodę `STDIN.gets`. Jeżeli wystąpi EOF (ang. *end of file* — koniec pliku) podczas pobierania linii, `gets` zwróci `nil`. Tak więc kod skojarzony z tą pętlą `while` będzie powtarzany dopóki nie zobaczy `^D` (lub ^Z czy też F6 pod DOS/Windows), co oznacza koniec wprowadzania.

`odp.chop!` w linii 6 usuwa ostatni znak z `odp`. W tym wypadku zawsze będzie to znak nowej linii, gdyż `gets` dodaje ten znak by odzwierciedlić naciśnięcie przez użytkownika klawisza Enter, co w naszym wypadku jest niepotrzebne.

W linii 15 drukujemy tajne słowo (`sekret`). Zapisaliśmy to jako wyrażenie `puts` (skrót od ang. *put string* — dosł. "połóż łańcuch") z dwoma argumentami, które są drukowane jeden po drugim. Można to zapisać równoważnie z jednym argumentem, zapisując `sekret` jako `#\sekret\` by było jasne, że jest to zmienna do przetworzenia, nie zaś literalne słowo:

```
puts "Chodzilo o #{sekret}."
```

Wielu programistów uważa, że utworzenie pojedynczego łańcucha jako argumentu metody `puts` to czytelniejszy sposób formułowania wyjścia.

Również my stosujemy `puts` do standardowego wyjścia naszego skryptu, ale ten skrypt używa również `print` zamiast `puts`, w liniach 4 i 13. `puts` i `print` nie oznaczają dokładnie tego samego. `print` wyświetla dokładnie to, co jest podane. `puts` ponadto zapewnia, że linia na wyjściu posiada znak końca linii. Używanie `print` w liniach 4 i 13 ustawia kursor dalej, poza uprzednio wydrukowanym tekstem zamiast przenosić go na początek następnego wiersza. Tworzy to rozpoznawalny znak zachęty do wprowadzania danych przez użytkownika.

Poniższe cztery wywołania wyjścia są równoważne:

```
# nowa linia jest automatycznie dodawana przez puts, jeżeli znak nowej linii jeszcze nie wystąpił
puts  "Zona Darwina, Esmerelda, zginela w ataku pingwinow."

# znak nowej linii musi być jawnie dodany do polecenia print:
print "Zona Darwina, Esmerelda, zginela w ataku pingwinow.\n"

# możesz dodawać wyjście stosując +:
print "Zona Darwina, Esmerelda, zginela w ataku pingwinow." + "\n"

# lub możesz dodawać podając więcej niż jeden łańcuch:
print "Zona Darwina, Esmerelda, zginela w ataku pingwinow.", "\n"
```

Jeden słaby punkt: czasami okno tekstowe z powodu prędkości działania posiada buforowane wyjście. Poszczególne znaki są buforowane i wyświetlane dopiero gdy pojawi się znak przejścia do nowej linii. Więc, jeżeli skrypt naszej zgadywanki nie pokazuje zachęty dla użytkownika dopóki użytkownik nie poda odpowiedzi, niemal na pewno winne jest buforowanie. Aby upewnić się, że tak się nie stanie możesz wyświetlić (ang. *flush* — dosł. "wylać") wyjście jak tylko zostanie wydrukowana zachęta dla użytkownika. `flush` mówi standardowemu urządzeniu wyjściowemu (obiekt nazwany `STDOUT`), "nie czekaj — wyświetl to co masz w tej chwili".

```
print "zgadnij? "; STDOUT.flush print "zgadnij? "; STDOUT.flush
```

Będziemy z tym bezpieczniejsi również w następnym skrypcie.

Wyrażenia regularne

W końcu zbadamy program z rozdziału o wyrażeniach regularnych.

```
st = "\033[7m"
en = "\033[m"

puts "Aby zakonczyc wprowadz pusty tekst."
```

```
while true
  print "tekst> "; STDOUT.flush; tekst=gets.chop
  break if tekst.empty?
  print "wzor> "; STDOUT.flush; wzor=gets.chop
  break if wzor.empty?
  wyr = Regexp.new(wzor)
  puts tekst.gsub(wyr, "#{st}\\&#{en}")
end
```

W linii 6 w warunku dla pętli `while` zakodowana jest "na sztywno" wartość `true`, co w efekcie daje nam nieskończoną pętlę. Aby więc przerwać wykonywanie pętli umieściliśmy instrukcje `break` w liniach 8 i 10. Te dwie instrukcje są również przykładem *modyfikatorów* `if`. *Modyfikator* `if` wykonuje wyrażenie po swojej lewej stronie wtedy i tylko wtedy, gdy określony warunek jest prawdziwy. Konstrukcja ta jest niezwykła, gdyż działa logicznie od prawej do lewej strony, ale jest dostępna, ponieważ wielu ludziom przypomina podobne wzorce obecne w mowie potocznej. Dodatkowo jest ona zwięzła — nie potrzebuje wyrażenia `end` by wskazać interpreterowi ile kodu następującego po `if` ma być traktowane jako warunek. *Modyfikator* `if` jest wygodnym sposobem używanym w sytuacjach, gdzie wyrażenie i warunek są wystarczająco krótkie by zmieścić się razem w jednej linii skryptu.

Rozważmy zmiany w interfejsie użytkownika w stosunku do poprzedniego skryptu — łamigłówki. Bieżący interfejs pozwala użytkownikowi zakończyć program poprzez wciśnięcie klawisza Enter przy pustej linii. Sprawdzamy czy każda linia z wejścia jest pustym łańcuchem, a nie czy w ogóle istnieje.

W liniach 7 i 9 mamy "nie-destruktywny" `chop`. Pozbywamy się tak niechcianego znaku końca linii, który zawsze otrzymujemy od `gets`. Jak dodamy wykrzyknik, będziemy mieli "destruktywny" `chop`. Jaka to różnica? W Rubim istnieje konwencja dołączania znaków `!` lub `?` do końca nazw pewnych metod. Wykrzyknik (`!`) oznacza coś potencjalnie destruktywnego, coś co może zmienić wartość przylegającego wyrażenia. `chop!` zmienia łańcuch bezpośrednio, `chop` daje ci obciętą kopię bez psucia oryginału. Oto ilustracja tej różnicy.

```
irb(main):001:0> s1 = "teksty"
=> "teksty"
irb(main):002:0> s1.chop!        # To zmienia s1.
=> "tekst"
irb(main):003:0> s2 = s1.chop    # To tworzy zmienioną kopię pod s2,
=> "teks"
irb(main):004:0> s1              # ... bez wpływu na s1.
=> "tekst"
```

Czasem będziesz też widział w użyciu `chomp` i `chomp!`. Te dwa są bardziej selektywne: końcówka łańcucha jest obcinana tylko wtedy, gdy jest znakiem końca linii. Dla przykładu, `''XYZ''.chomp!` nie zrobi nic. Jeżeli potrzebujesz jakiegoś triku by zapamiętać różnice, pomyśl o osobie lub zwierzęciu, które smakuje coś nim zdecyduje się to ugryźć (ang. *chomp* — "jeść niechlujnie"), a toporze rąbiącym jak popadnie (ang. *chop* — "odrąbanie").

Pozostałe konwencje nazywania metod pojawiają się w liniach 8 i 10. Znak zapytania (`?`) oznacza metodę predykatową (orzekającą o czymś), która zwraca albo prawdę (`true`) albo fałsz (`false`).

Linia 11 tworzy obiekt będący wyrażeniem regularnym z łańcucha podanego przez użytkownika. Cała właściwa praca wykonywana jest wreszcie w linii 12, która używa `gsub` do globalnego podstawienia każdego dopasowania naszego wyrażenia z samym sobą, ale otoczonego przez znaczniki ANSI. Ta sama linia wyświetla również wyniki.

Moglibyśmy podzielić linię 12 na osobne linie, tak jak tutaj:

```
podswietlony = tekst.gsub(wyr,"#{st}\\&#{en}")
puts podswietlony
```

lub w "destruktywnym" stylu:

```
tekst.gsub!(wyr,"#{st}\\&#{en}")
puts tekst
```

Spójrz ponownie na ostatnią część linii 12. `st` i `en` były zdefiniowane w liniach 1-2 jako sekwencje ANSI które odpowiednio zmieniają i przywracają kolor tekstu. W linii 12 są one zawarte w `#\\` by w ten sposób były właściwie interpretowane (i nie wystąpiły zamiast nich nazwy zmiennych). Pomiędzy nimi widzimy `\\&`. Jest to trochę podstępne. Ponieważ podstawiany łańcuch zawarty jest w cudzysłowach (podwójnych), para odwróconych ukośników będzie zinterpretowana jako jeden ukośnik, co `gsub` zobaczy właściwie jako `\&`. To spowoduje powstanie specjalnego kodu, który z kolei odwoła się do czegokolwiek, co jako pierwsze będzie pasować do wzorca. Tak więc nowy łańcuch, gdy będzie wyświetlony będzie wyglądał tak jak pierwszy z wyjątkiem tego, że te fragmenty które pasują do wzorca będą wyświetlone w odwróconych kolorach.

Rozdział 9

Struktury sterujące

Struktury sterujące

Ten rozdział odkrywa nieco więcej na temat struktur sterujących Rubiego.

case

Instrukcji `case` używamy do sprawdzenia sekwencji warunków. Na pierwszy rzut oka jest ona podobna do instrukcji `switch` z języka C lub Java ale, jak za chwilę zobaczymy, znacznie potężniejsza.

```
def okresl(i)
  case i
  when 1, 2..5
    puts "1..5"
  when 6..10
    puts "6..10"
  end
end

okresl(8) #=> 6..10
```

`2..5` jest wyrażeniem oznaczającym przedział zamknięty od 2 do 5. Następujące wyrażenie sprawdza czy wartość `i` należy do tego przedziału:

```
(2..5) === i
```

`case` wewnętrznie używa operatora relacji `===` by sprawdzić kilkanaście warunków za jednym razem. W odniesieniu do obiektowej natury Rubiego, `===` jest interpretowany odpowiednio dla obiektu który pojawia się jako warunek w instrukcji `when`. Na przykład, następujący kod sprawdza równość łańcuchów znakowych w pierwszym wyrażeniu `when` oraz zgodność wyrażeń regularnych w drugim `when`.

```
def okresl(s)
  case s
  when 'aaa', 'bbb'
    puts "aaa lub bbb"
  when /def/
```

```
    puts "zawiera def"
  end
end

okresl("abcdef") #=> zawiera /def/
```

while

Ruby dostarcza wygodnych sposobów do tworzenia pętli, chociaż jak odkryjesz w następnym rozdziale, wiedza o tym jak używać iteratorów często zaoszczędzi ci bezpośredniego pisania własnych pętli.

`while` jest powtarzanym `if`. Używaliśmy tej instrukcji w nasze słownej zgadywance i w programach sprawdzających wyrażenia regularne (zobacz poprzedni rozdział). Tutaj instrukcja ta przyjmuje formę `while warunek ... end` otaczającą blok kodu który będzie powtarzany dopóty, dopóki warunek jest prawdziwy. Ale `while` i `if` mogą być łatwo zaaplikowane również do pojedynczych wyrażeń:

```
irb(main):001:0> i = 0
=> 0
irb(main):002:0> puts "To jest zero." if i == 0
To jest zero.
=> nil
irb(main):003:0> puts "To jest liczba ujemna" if i < 0
=> nil
irb(main):004:0> puts i += 1 while i < 3
1
2
3
=> nil
```

Czasami będziesz chciał zanegować sprawdzany warunek. `unless` jest zanegowanym `if`, natomiast `until` zanegowanym `while`. Jeśli chcesz, poeksperymentuj z tymi instrukcjami.

Są cztery sposoby do przerywania wykonywania pętli z jej wnętrza. Pierwszy, `break` oznacza, tak jak w C, zupełną ucieczkę z pętli. Drugi, `next`, przeskakuje na początek kolejnej iteracji (podobnie jak znane z C `continue`). Trzeci, to specyficzne dla Rubiego `redo`, które oznacza ponowne wykonanie bieżącej iteracji. Następujący kod w języku C ilustruje znaczenia instrukcji `break`, `next`, i `redo`:

```
while (warunek) {
etykieta_redo:
   goto etykieta_next;        /* w Rubim: "next" */
   goto etykieta_break;       /* w Rubim: "break" */
   goto etykieta_redo;        /* w Rubim: "redo" */
   ...
   ...
etykieta_next:
}
etykieta_break:
...
```

Czwarty sposób by wyjść z pętli będąc w jej wnętrzu to `return`. Obliczenie instrukcji `return` spowoduje wyjście nie tylko z pętli ale również z metody która tę pętlę zawiera. Jeżeli podany został argument, będzie on zwrócony jako rezultat wywołania metody. W przeciwnym wypadku zwracane jest `nil`.

for

Programiści C mogą się zastanawiać jak zrobić pętlę "`for`". Petla `for` Rubiego może służyć w ten sam sposób, choć jest nieco bardziej elastyczna. Pętla poniżej iteruje każdy element w kolekcji (tablicy, tablicy asocjacyjnej, sekwencji numerycznej, itd.), ale nie zmusza programisty do myślenia o indeksach:

```
for element in kolekcja
  # ... "element" wskazuje na element w kolekcji
end
```

Kolekcją może być przedział wartości (to właśnie większość ludzi ma na myśli, gdy mówi o pętli `for`):

```
for num in (4..6)
  print num
end
#=> 456
```

W tym przykładzie przeiterujemy kilka elementów tablicy:

```
for elem in [100, -9.6, "zalewa"]
  puts "#{elem}\t(#{elem.class})"
end
#=> 100     (Fixnum)
#   -9.6    (Float)
#   zalewa  (String)
```

Ale tak naprawdę `for` jest po prostu innym sposobem zapisania instrukcji `each`, która jest naszym pierwszym przykładem iteratora. Poniższe formy są równoważne:

Jeżeli przywykłeś do C lub Javy, możesz preferować tą.

```
for element in kolekcja
  ...
end
```

Natomiast programista Smalltalka może preferować taką.

```
kolekcja.each {|element|
  ...
}
```

Iteratory często mogą być używane zamiast konwencjonalnych pętli. Jak już nabierzesz wprawy w ich użyciu, stają się zazwyczaj łatwiejsze w od pętli. Ale zanim dowiemy się więcej o iteratorach, poznajmy jedną najciekawszych konstrukcji języka Ruby: domknięcia.

Rozdział 10

Domknięcia i obiekty procedurowe

Domknięcia i obiekty procedurowe

Domknięcia

Ruby jest językiem korzystającym w dużym stopniu z domknięć. Domknięcie jest blokiem kodu przekazywanym do metody. Samo w sobie nie jest obiektem. Domknięcie zawierające niewiele instrukcji, które można zapisać w jednej linii zapisujemy pomiędzy nawiasami klamrowymi ({}), zaraz za wywołaniem metody:

```
3.times { print "Bla" } #=> BlaBlaBla
```

Domknięcia dłuższe zapisujemy w bloku `do ... end`

```
i = 0
3.times do
  print i
  i += 2
end
#=> 024
```

Obsługa bloku przekazanego do funkcji odbywa się poprzez słowo kluczowe `yield`, które przekazuje sterowanie do bloku. Spójrzmy na przykład metody `powtorz`.

```
def powtorz(ilosc)
  while ilosc > 0
    yield   # tu przekazujemy sterowanie do domkniecia
    ilosc -= 1
  end
end

powtorz(3) { print "Bla" } #=> BlaBlaBla
```

Po zakończeniu wykonywania przekazanego bloku sterowanie wraca z powrotem do metody. Dzięki słowu kluczowemu `yield` możemy również przekazywać do bloku obiekty:

```
def powtorz(ilosc)
  while ilosc > 0
    yield ilosc
    ilosc -= 1
  end
end
```

Aby użyć wartości przekazanej do bloku stosujemy identyfikator ujęty w znaki |:
`powtorz(3) { |n| print "#{n}.Bla " } #=> 3. Bla 2. Bla 1.Bla`
Co jednak, gdy używamy `yield`, a do metody nie przekazaliśmy żadnego bloku? Aby uchronić się przed wystąpieniem wyjątku używamy metody `block_given?`, która zwraca `true`, gdy blok został przekazany.

```
def powtorz(ilosc)
  if block_given?
    while ilosc > 0
      yield ilosc
      ilosc -= 1
    end
  else
    puts "Brak bloku"
  end
end

powtorz(3) # nie przekazujemy bloku
#=> Brak bloku
```

Obiekty procedurowe

Bloki można zamienić w łatwy sposób na obiekty (są to obiekty klasy `Proc`. O tym, czym dokładnie są obiekty i klasy dowiesz się w rozdziale o klasach) Można użyć w tym celu słów kluczowych `lambda` lub `proc`, z czego **zalecane** jest to pierwsze. Poniższy kod utworzy dwa obiekty procedurowe:

```
hej = lambda { print "Hej" }

witaj = proc do
  puts "Witaj!"
end
```

Aby wykonać dany blok zawarty w obiekcie procedurowym (wywołać go) należy użyć metody `call`:

```
hej.call #=> Hej
witaj.call #=> Witaj!
```

W wywołaniu `call` możemy również przekazać parametry do bloku:

```
drukuj = lambda { |tekst| print tekst }
drukuj.call("Hop hop!") #=> Hop hop!
```

Obiekty procedurowe mogą być, jak każde inne obiekty, przekazywane jako parametry. Możemy zdefiniować alternatywną metodę `powtorz` która bedzie wykorzystywać lambdę przekazaną jako parametr. Rozważmy poniższy przykład:

```
def powtorz(ile, co)
  while ile > 0
    co.call(ile) # wywołujemy blok "co"
    ile -= 1
  end
end

l = lambda do |x|
    print x
end

powtorz(3, l) #=> 321
powtorz(3, lambda { print "bla" }) #=> blablabla
```

Jak widzimy w ostatniej linii, obiekty lambda mogą być anonimowe (nie nadajemy im żadnej nazwy). O obiektach anonimowych dowiemy się wkrótce więcej. Natomiast w rozdziale o zmiennych lokalnych zobaczymy, że obiekty procedurowe i domknięcia zachowują kontekst (stan zmiennych lokalnych) w jakim zostały wywołane.

Różnice między lambdą a `Proc.new`

Obiekty procedurowe można również tworzyć używając konstrukcji `Proc.new`. Bardziej szczegółowo omówimy tę konstrukcję w rozdziale dotyczącym klas. Tutaj jedynie przedstawimy pewne różnice pomiędzy lambdami a obiektami utworzonymi za pomocą `Proc.new`.

Surowe obiekty `Proc` (ang. *raw procs*), czyli utworzone poprzez `Proc.new`, posiadają jedną niedogodność: użycie instrukcji `return` powoduje nie tyle wyjście z domknięcia obiektu procedurowego, co wyjście z całego bloku, w którym domknięcie było wywołane. Może to powodować niespodziewane wyniki działania naszych programów, dlatego zaleca się używanie lambd, a nie surowych obiektów `Proc`.

```
def proc1
  p = Proc.new { return -1 }
  p.call
  puts "Nikt mnie nie widzi :-("
end

def proc2
  p = lambda { return -1 }
  puts "Blok zwraca #{p.call}"
end
```

Wywołany `proc1` zwraca jedynie wartość, nie wypisze żadnego tekstu. Odmiennie działa `proc2` — tutaj `return` powoduje, że sama lambda zwraca wartość, do której można się odwołać w dalszej części bloku, w którym utworzono lambdę.

Rozdział 11

Iteratory

Iteratory

Iteratory nie są oryginalnym pojęciem Rubiego. Występują one powszechnie w językach programowania zorientowanych obiektowo. Używane są również w Lispie, choć nie są tam nazywane iteratorami. W tym rozdziale szczegółowo przyjrzymy się wszechobecnym iteratorom Rubiego.

Czasownik "iterować" oznacza wykonywać tę samą czynność wiele razy, tak więc iterator jest czymś co wykonuje tę samą rzecz wiele razy (przykładem może być metoda `powtórz` z rozdziału o domknięciach).

Podczas pisania kodu potrzebujemy pętli w wielu różnych sytuacjach. W C, kodujemy je używając `for` lub `while`. Na przykład:

```
char *str;
for (str = "abcdefg"; *str != '\0'; str++) {
  /* tutaj przetwarzamy znak */
}
```

Składnia pętli `for (...)` z języka C dostarcza pewnej abstrakcji, która pomaga w utworzeniu pętli, ale sprawdzenie czy `*str` nie wskazuje na znak pusty znak wymaga od programisty znajomości szczegółów o wewnętrznej strukturze łańcucha znakowego. Między innymi dlatego, C jest odbierany jako język niskiego poziomu. Języki wyższego poziomu odznaczają się bardziej elastycznym wsparciem iteracji. Rozważ następujący skrypt *sh* powłoki systemowej:

```
#!/bin/sh

for i in *.[ch]; do
  # ... tutaj byłby kod do wykonania dla każdego pliku
done
```

Wszystkie pliki źródłowe i nagłówkowe języka C w bieżącym katalogu są przetwarzane i powłoka systemowa bierze na siebie detale dotyczące wskazywania i podstawiania po kolei wszystkich nazw plików, jedna po drugiej. To chyba działa na wyższym poziomie niż C, nie sądzisz?

Trzeba zauważyć jeszcze jedno: często język dostarcza iteratorów dla typów wbudowanych, ale budzi rozczarowanie gdy okazuje się, że musimy wracać z powrotem do

pętli nisko poziomowych by iterować nasze własne typy danych. W programowaniu zorientowanym obiektowo (OOP — ang. *Object-Oriented Programming*), użytkownicy zazwyczaj definiują dużo własnych typów danych, więc to może być całkiem poważny problem.

Każdy język wspierający OOP zawiera jakieś udogodnienia dotyczące iterowania. Niektóre języki dostarczają w tym celu specjalnych klas, natomiast Ruby pozwala na definiowanie iteratorów bezpośrednio, używając w tym celu znanych już nam domknięć.

Typ `String` Rubiego posiada kilka użytecznych iteratorów:

```
ruby> "abc".each_byte { |c| printf "<%c>", c }
#=> <a><b><c>
```

`each_byte` to iterator wskazujący każdy znak w łańcuchu. Każdy znak jest podstawiany do zmiennej lokalnej `c`. To samo można przełożyć na coś bardziej przypominającego kod C...

```
s = "abc"
i = 0
while i < s.length
  printf "<%c>", s[i]
  i+=1
end
#=> <a><b><c>
```

... jednakże iterator `each_byte` jest koncepcyjnie prostszy, i wydaje się, że działałby nadal nawet gdyby klasa `String` uległa w przyszłości radykalnym modyfikacjom. Dużą zaletą iteratorów jest to, że zachowują one swoje poprawne działanie na przekór takim radykalnym zmianom. Jest to charakterystyczna cecha dobrego kodu w ogólności.

Innym iteratorem klasy `String` jest `each_line`.

```
"a\nb\nc\n".each_line { |l| print l }
#=> a
#   b
#   c
```

Zadania które wymagałyby dużego wysiłku w C (wyszukiwanie ograniczników linii, generowanie podłańcuchów, itd.) z użyciem iteratorów można wykonać bardzo łatwo.

Instrukcja `for` pojawiająca się w rozdziale o instrukcjach sterujących dokonywała iteracji przez użycie iteratora `each`. Iterator `each` klasy `String` działa w ten sam sposób jak `each_line`, więc przepiszmy powyższy przykład z `for`:

```
for l in "a\nb\nc\n"
  print l
end
#=> a
#   b
#   c
```

Możemy używać struktury sterującej `retry` w połączeniu z iterowaną pętlą. Spowoduje ona rozpoczęcie iterowania pętli od początku.

```
c = 0
for i in 0..4
  print i
  if i == 2 and c == 0
    c = 1
    print "\n"
    retry
  end
end
#=> 012
#   01234
```

Zamienienie `retry` na `redo` w powyższym przykładzie spowoduje, że tylko bieżąca iteracja będzie wykonana ponownie, z następującym wynikiem:

```
012
234
```

`yield` jak już wiemy, jest wyrażeniem, które przenosi sterowanie do bloku kodu który został przekazany do iteratora. Używając instrukcji `yield` i `retry` można zdefiniować iterator który będzie działał mniej więcej jak standardowa pętla `while`.

```
def WHILE(warunek)
  return if not warunek
  yield
  retry
end

i=0
WHILE(i < 3) { print i; i+=1 } #=> 012
```

Jak więc widzimy, iteratory w Rubim są metodami obsługującymi przekazane do nich domknięcia. Owszem, istnieją pewne ograniczenia, ale możesz pisać własne oryginalne iteratory. Szczególnie, gdy definiujemy nowy typ danych, wygodnie jest zdefiniować odpowiednie iteratory które będą na nim operować. W tym kontekście powyższe przykłady nie są szczególnie użyteczne. Poznajmy zatem bardziej praktyczne iteratory.

Przegląd iteratorów

`all?`

Przekazuje do bloku każdy element kolekcji. Zwraca `true`, jeśli blok nigdy nie zwróci `false` (lub `nil`).

```
[1, 2, 5].all? { |element| element <= 5 }                    #=> true
[1, 2, 5].all? { |element| element <= 4 }                    #=> false
%w{RAM CPU GPU DDR}.all? { |element| element.length == 3 }   #=> true

[1, 2, 5].all? do |element|
  puts "Sprawdzam #{element}; #{element} < 5: #{element < 5}"
  element < 5
end                                                          #=> false
```

Wyjście:

```
Sprawdzam 1; 1 < 5: true
Sprawdzam 2; 2 < 5: true
Sprawdzam 5; 5 < 5: false
```

`any?`

Zwraca `true`, jeśli przekazany do bloku element kiedykolwiek zwróci `true`.

```
[1, 2, 5].any? { |element| element > 5 }    # => false
[1, 2, 5].any? { |element| element == 2}    # => true
```

`collect(map)`

Przekazuje do bloku każdy element kolekcji, następnie tworzy nową — z elementów zwracanych przez blok.

```
%w{kot tulipan parowka}.collect { |element| element.upcase }
# => ["KOT", "TULIPAN", "PAROWKA"]
[1, 2, 3].collect { |element| element + 1}
#=> [2, 3, 4]
```

`collect!(map!)`

Działa jak `collect`, z tą jednak różnicą, że operacji kolekcja dokonuje na sobie, w każdej iteracji zmieniając swoją zawartość.

```
a = [1, 2, 3]                              #=> [1, 2, 3]
a.collect! { |element| element + 1 }       #=> [2, 3, 4]
a                                          #=> [2, 3, 4]
```

`delete_if`

Usuwa z kolekcji elementy, dla których blok zwraca `true`

```
[1, 2, 3, 4, 5, 6].delete_if { |i| i%2 == 0 }   # => [1, 3, 5]
```

`detect(find)`

Zwraca pierwszy element, dla którego blok zwróci `true`

```
(36..100).detect { |i| i%7 == 0 }     # => 42
```

`downto`

Wykonuje blok, podając w kolejności malejącej liczby od siebie samej do podanej jako parametr.

```
9.downto(0) { |i| print i } #=> 9876543210
```

`each`

Przekazuje do bloku każdy z elementów kolekcji

```
['pies', 'kot', 'ryba'].each { |word| print word + " " }
(0..9).each { |i| print i }
#=> pies kot ryba 0123456789
```

`each_index`

Działa jak each, ale przekazuje sam indeks każdego elementu.

```
[3, 6, -5].each_index { |i| print i.to_s + " " }
#=> 0 1 2
```

`each_with_index`

Przekazuje jednocześnie element i jego indeks do bloku.

```
["jeden", 2, "trzy"].each_with_index do |element, index|
  puts "Indeksowi #{index} przyporzadkowalem #{element}"
end

#=> Indeksowi 0 przyporzadkowalem jeden
#   Indeksowi 1 przyporzadkowalem 2
#   Indeksowi 2 przyporzadkowalem trzy
```

`find_all`

Zwraca wszystkie elementy kolekcji, dla których blok zwróci `true`.

```
(0..30).find_all { |i| i%9 == 0 }     #=> [0, 9, 18, 27]
```

`grep`

Zwraca elementy spełniające dopasowanie podane jako parametr. Jeśli podano blok, przekazuje do niego tylko te elementy i zwraca tablicę zbudowaną z wartości zwracanych przez blok.

```
# Zwraca wyrazy zawierajace litere 'r'
%w{ruby python perl php}.grep(/r/) do |w|
  print "#{w.upcase} "
  w.capitalize
end                          #=> ["Ruby", "Perl"]

#=> RUBY PERL
```

`inject`

Przekazuje do bloku każdy element kolekcji. Posiada dodatkowo pamięć, która początkowo jest równa pierwszemu elementowi (lub wartości podanej jako parametr). Po zakończeniu każdej iteracji pamięć jest aktualizowana do wartości zwracanej przez blok.

```
# Zwraca największą liczbę z tablicy
a = [-5, 2, 10, 17, -50]
a.inject a.first do |mem, element|
  mem > element ? mem : element
end                                  #=> 17

# Silnia
```

```
(1..5).inject do |mem, element|
  mem *= element
end                                  #=> 120
```

`partition`

Zwraca dwie tablice: jedną z elementami, dla których blok zwraca `true` i drugą — z resztą.

```
(1..6).partition { |i| i%2 == 0 }   #=> [[2, 4, 6], [1, 3, 5]]
```

`reject`

Odrzuca z kolekcji wszystkie elementy, dla których blok zwróci `true`.

```
(1..10).reject { |i| i >= 3 and i <= 7 }    #=> [1, 2, 8, 9, 10]
```

`reject!`

Wyrzuca z siebie elementy, dla których blok zwraca `true`.

```
a = (1..10).to_a                      # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
a.reject! { |i| i >= 3 and i <= 7 }  # => [1, 2, 8, 9, 10]
a                                     # => [1, 2, 8, 9, 10]
```

`reverse_each`

Działa jak `each` tyle, że podaje elementy w odwrotnej kolejności.

```
(0..9).to_a.reverse_each { |i| print i }
#=> 9876543210
```

`step`

Przekazuje do bloku wartości od, do — z określonym krokiem.

```
# (1)
0.step(100, 10) { |i| puts i}
# (2)
(0..100).step(10) { |i| puts i }
```

W obu przypadkach wyjście będzie wyglądało tak:

```
0
10
20
30
40
50
60
70
80
90
100
```

`times`

Wykonuje dany blok określoną ilość razy.

```
5.times { puts "Hej!" }
5.times { |i| print "#{i} "}

#=> Hej!
#   Hej!
#   Hej!
#   Hej!
#   Hej!
#   0 1 2 3 4
```

`upto`

Iteruje blok, przekazując liczby od, do.

```
1.upto(3) { |i| print i }
#=> 123
```

Rozdział 12

Myślenie zorientowane obiektowo

Myślenie zorientowane obiektowo

Zorientowany obiektowo to chwytliwe określenie. Powiedzenie o czymkolwiek, że jest "zorientowane obiektowo" brzmi naprawdę mądrze. Ruby określa się jako język skryptowy zorientowany obiektowo, ale co to naprawdę znaczy "zorientowany obiektowo"?

Istnieje wiele różnorodnych odpowiedzi na to pytanie, które można by prawdopodobnie sprowadzić do tego samego. Zamiast jednak zbyt szybko podsumowywać to zagadnienie, spróbujmy pomyśleć przez chwilę o tradycyjnym paradygmacie programowania.

Tradycyjnie, problem programowania rozwiązywany jest przez podejście, w którym obecne są różne typy reprezentacji danych oraz procedury które na tych danych operują. W modelu tym dane są obojętne, pasywne i bezradne; oczekują na łaskę dużego proceduralnej bryły, która jest aktywna, logiczna i wszechmocna.

Problem w tym podejściu jest taki, że programy są pisane przez programistów, którzy są tylko ludźmi i potrafią pamiętać i kontrolować w danym czasie tylko pewną skończoną ilość detali. Jak projekt staje się większy jego proceduralny rdzeń rośnie do punktu, w którym trudno jest już pamiętać jak cała rzecz działa. Drobne pomyłki w sposobie myślenia i usterki typograficzne stają się przyczyną dobrze zamaskowanych błędów. Złożone i niezamierzone interakcje zaczynają wynurzać się z proceduralnego rdzenia i zarządzanie tym zaczyna przypominać noszenie w kółko wściekłej kałamarnicy i walkę z jej mackami. Są pewne wytyczne dotyczące programowania, które mogą pomóc zminimalizować i zlokalizować błędy w tym tradycyjnym paradygmacie, ale jest lepsze rozwiązanie które pociąga za sobą fundamentalną zmianę sposobu pracy.

Programowanie zorientowane obiektowo pozwala nam przekazywać większość przyziemnych i monotonnych czynności logicznych do samych danych; zmienia to nasze pojmowanie danych z pasywnych na aktywne. Innymi słowy:

- Przestajemy traktować każdy kawałek danych jak skrzynkę z otwartym wiekiem, do której możemy wkładać lub wyjmować przedmioty.
- Zaczynamy traktować każdy kawałek danych jak pracującą maszynę, z zamkniętą pokrywą i dobrze oznakowanymi przełącznikami oraz potencjometrami.

To co nazwaliśmy wyżej "maszyną" może być w środku bardzo proste lub bardzo złożone. Nie możemy tego określić patrząc z zewnątrz, jak i nie grzebiemy w jej wnętrzu (za wyjątkiem sytuacji, gdy jest absolutnie pewnie, że coś jest nie tak z jej projektem), więc jedyne czego się od nas wymaga by wpływać na dane to przekręcanie przełączników i odczytywanie potencjometrów. Jak już maszyna jest zbudowana nie chcemy sobie dalej zaprzątać głowy tym, jak ona działa.

Możesz sądzić że po prostu robimy sobie więcej roboty, ale tak naprawdę to dobra robota w celu chronienia wielu rzeczy przed błędami.

Zacznijmy od przykładu, który jest zbyt prosty by mieć wartość praktyczną, ale powinien zilustrować przynajmniej jedną cześć tej koncepcji. Twój samochód ma tripmeter[1]. Jego celem jest rejestrowanie odległości którą przebył pojazd od momentu naciśnięcia przycisku. Jak moglibyśmy wymodelować to w języku programowania? W C, tripmeter byłby po prostu zmienną numeryczną, możliwe że typu `float`. Program mógłby manipulować tą zmienną zwiększając jej wartość przyrostowo małymi krokami, z okazjonalnym resetowaniem jej wartości do zera, jeśli zaszłaby taka potrzeba. A co w tym złego? Z nieokreślonej liczby niespodziewanych powodów błąd w programie mógłby przypisać błędną wartość do zmiennej. Każdy, kto programował w C wie, co to znaczy spędzać godziny lub dni próbując ustalić gdzie tkwi taki błąd. Jego przyczyna, jak się już ją odkryje, wydaje się absurdalnie głupia. (Moment znajdowania błędu jest przeważnie rozpoznawalny przez odgłos głośnego klepnięcia w czoło.)

Ten sam problem można zaatakować z zupełnie innej strony, w podejściu zorientowanym obiektowo. Pierwszym pytaniem, które zadaje programista, gdy projektuje tripmeter nie jest "jaki znany mi typ danych odpowiada najbliżej tej rzeczy?" ale "jak właściwie ta rzecz ma działać?" Różnica jest zasadnicza. Potrzeba poświęcić odrobinę czasu ustalając po co dokładnie jest drogomierz i w jaki sposób zewnętrzny świat zamierza się z nim kontaktować. Decydujemy się zbudować małą maszynkę z metodami regulacji które pozwolą nam zwiększać wartość, czytać ją, kasować, i nic poza tym.

Nie dostarczamy żadnego sposobu na przypisanie do tripmetera arbitralnych wartości. Dlaczego? Ponieważ wszyscy wiemy, że drogomierze nie działają w ten sposób. Jest tylko kilka rzeczy, które powinieneś robić z tripmeterem, i to wszystko na co pozwalamy. Zatem, jeśli coś innego w programie błędnie spróbuje umieścić jakąś inną wartość (powiedzmy, docelową temperaturę ze systemu sterowania klimatyzacją w samochodzie) w tripmeterze, dostaniemy natychmiastową wskazówkę co poszło nie tak. Będziemy poinformowani, gdy program zostanie uruchomiony (lub podczas kompilacji, zależnie od natury języka programowania), że nie mamy prawa przypisywać arbitralnych wartości do obiektów `Tripmeter`. Wiadomość może nie będzie dokładnie tak jasna, ale będzie ona sensownie zbliżona. Nie uchroni nas to przed błędem, prawda? Ale za to szybko wskaże nam, gdzie mniej więcej leży przyczyna błędu. To tylko jeden z kilkunastu sposobów, w jaki OOP może nam zaoszczędzić dużo zmarnowanego czasu.

Zazwyczaj robimy jeszcze jeden krok w abstrakcji, ponieważ okazuje się, że równie łatwo jak naszą maszynę można zbudować całą fabrykę która tworzy takie maszyny. Prawdopodobnie nie budowalibyśmy bezpośrednio pojedynczego tripmetera, raczej zbudowalibyśmy dowolną ilość tripmeterów z pojedynczego wzorca. Ten wzorzec (lub jak wolisz, fabryka tripmeterów) odpowiada temu co nazywamy klasą. Indywidualny tripmeter wygenerowany z tego wzorca (lub zbudowany przez fabrykę) odpowiada obiektowi. Większość języków obiektowych, wymaga by klasa była zdefiniowana nim

[1] Urządzenie elektroniczne, które rejestruje czas pracy pojazdu, liczbę przejechanych kilometrów, zużycie paliwa oraz inne parametry — przyp. tłum.

będziemy mogli uzyskać nowy rodzaj obiektu, ale nie Ruby.

Warto tu zanotować, że użycie języka zorientowanego obiektowo nie wymusza odpowiedniego zorientowanego obiektowo projektu. W rzeczy samej, pisanie kodu, który jest niejasny, niechlujny, źle zaczęty, pełny błędów i chwiejący się na wszystkie strony, możliwe jest w każdym języku. To co Ruby robi dla ciebie (szczególnie w przeciwieństwie do C++) to to, że praktyka programowania obiektowego jest na tyle naturalna, że nawet gdy pracujesz w małej skali nie czujesz potrzeby by uciec się do brzydkiego kodu by zaoszczędzić sobie wysiłku. Będziemy omawiać sposoby, w których Ruby osiąga ten wspaniały cel, w miarę postępu tego podręcznika. Następnym tematem będą "przełączniki i potencjometry" (metody obiektów) a stamtąd przeniesiemy się do "fabryk" (klas). Jesteś wciąż z nami?

Rozdział 13

Metody

Metody

Czym jest metoda?

W programowaniu obiektowym nie myślimy o operowaniu na danych bezpośrednio spoza obiektu. Obiekty mają raczej pewne rozumienie tego jak należy operować na sobie samych (gdy ładnie poprosimy by to robiły). Można powiedzieć, że przekazujemy pewne wiadomości do obiektu i te wiadomości zazwyczaj powodują jakiegoś rodzaju akcję lub uzyskują znaczącą odpowiedź. Powinno to dziać się bez naszej zaangażowania w to, jak obiekt naprawdę działa od wewnątrz. Zadania, o wykonanie których mamy prawo prosić obiekt (lub równoważnie — wiadomości które on zrozumie) są właśnie owymi metodami obiektu.

W Rubim wywołujemy metody obiektu posługując się zapisem z kropką (tak jak w C++ lub Javie). Nazwa obiektu do którego mówimy znajduje się na lewo od kropki.

```
"abcdef".length #=> 6
```

Rozumując intuicyjnie, ten łańcuch pytany jest o swoją długość. Technicznie natomiast, wywołujemy metodę `length` na rzecz obiektu ‘‘`abcdef`’’.

Pozostałe obiekty mogą mieć nieco inną interpretację długości lub nawet nie mieć żadnej. Decyzje dotyczące tego, jak odpowiedzieć na wiadomość podejmowane są w locie, podczas wykonywania programu, i podejmowane działanie może być zmienione w zależności o tego, na co wskazuje zmienna.

```
a = "abc"
puts a.length #=> 3
a = ["abcde", "fghij"]
a.length #=> 2
```

To, co rozumiemy przez długość może się różnić w zależności od rodzaju obiektu do którego mówimy. Za pierwszym razem w powyższym przykładzie pytamy `a` o jej długość i `a` wskazuje na prosty łańcuch znakowy, więc jest tylko jedna sensowną odpowiedź. Za drugim razem `a` odnosi się do tablicy i możemy rozsądnie myśleć o jej długości jako o 2, 5 lub 10 — oczywiście w tym przypadku jest to 2. Różne obiekty mogą mieć różnego rodzaju długości.

```
a[0].length #=> 5
a[0].length + a[1].length #=> 10
```

Rzeczą godną uwagi jest to, że tablica wie o sobie to coś, co oznacza, że jest ona tablicą. W Rubim kawałki danych przechowują tę wiedzę. Zatem żądania, które wobec nich kierujemy mogą być automatycznie spełnione na wiele różnych sposobów. Zdejmuje to z programisty brzemię pamiętania olbrzymiej liczby wielu specyficznych nazw funkcji, ponieważ relatywnie mała liczba nazw metod (będących w zgodzie koncepcjami wyrażalnymi w języku naturalnym) może być zastosowana do różnych typów danych. W rezultacie programista otrzymuje to, czego się spodziewał. Ta cecha języków programowania obiektowego nazywana jest *polimorfizmem*.

Kiedy obiekt otrzymuje komunikat, którego nie rozumie, "podnoszony" jest błąd:

```
a = 5
a.length
ERR: (eval):1: undefined method ‘length’ for 5(Fixnum)
```

Tak więc należy wiedzieć, które metody są akceptowane przez obiekt, chociaż nie trzeba analizować jak są one przetwarzane.

Jeżeli przekazujemy do metody jakieś argumenty, zazwyczaj otaczamy je nawiasami okrągłymi:

`obiekt.metoda(arg1, arg2)`

Można je pominąć, jeśli nie stanie się to przyczyną dwuznaczności[1].

`obiekt.metoda arg1, arg2`

Jest pewna specjalna zmienna w Rubim — `self`. Odnosi się ona tylko do obiektu na rzecz którego wywołujemy metodę. Dzieje się to tak często, że dla wygody "`self.`" może być opuszczone w metodach odwołujących się z danego obiektu do samego obiektu:

`self.nazwa_metody(argumenty...)`

oznacza to samo co

`nazwa_metody(argumenty...)`

To co tradycyjnie nazwalibyśmy wywołaniem funkcji jest po prostu skróconą formą zapisu wywołań metod przez `self`. To właśnie czyni z Rubiego czysto obiektowy język programowania. Ponadto metody funkcyjne nadal zachowują się całkiem podobnie do funkcji w innych językach programowania. Jest to pewne ułatwienie dla tych, którym łatwiej jest traktować wywołania metod jak wywołania funkcji. Jeśli chcemy, możemy, np. w celach edukacyjnych, traktować funkcje tak jakby nie były one naprawdę metodami obiektów.

W rozdziale dotyczącym zmiennych klasowych zobaczymy zastosowanie słowa kluczowego `self` przy definiowaniu metod należących do całej klasy, czyli metod *klasowych*.

* czyli zmienna lista argumentów

Czasami, analizując różne przykłady kodu w Rubim możemy natknąć się na taką definicję metody (albo wywołanie), w której ostatni parametr poprzedzony jest znakiem `*` lub `&`. Dla początkujących może to wyglądać enigmatycznie, a programistów C/C++/C# mogą dodatkowo mylić skojarzenia ze wskaźnikami i referencjami. Obydwa znaki mają jednak zupełnie inne znaczenie, a ponieważ nie ma nic gorszego od kodu, którego nie rozumiemy, wyjaśnijmy znaczenie obu tych symboli.

[1] Zaleca się jednak pomijanie nawiasów tylko w wywołaniach najprostszych i najbardziej oczywistych metod, jak np. `puts`.

Gwiazdka (*) oznacza *zmienną* listę argumentów. Jeżeli * pojawia się w nagłówku definiowanej metody, poprzedzając ostatni parametr, oznacza to, że począwszy od tego tego argumentu do metody można przekazać dowolną ich ilość. Wszystkie te argumenty są widoczne w metodzie jako tablica.

```
def metoda(*args)
  wynik = ""
  args.each {|arg| wynik += "#{arg}, "}
  wynik[0...-2] # ucinamy 2 ostanie znaki: ", "
end

puts metoda("a", "b", 3) #=> a, b, 3
```

Gwiazdkę * można też stosować w wywołaniu metody, przed ostatnim argumentem — tablicą. Powoduje ona wtedy konwersję z tablicy na poszczególne argumenty:

```
def inna_metoda(a, b, c)
  "#{a}, #{b}, #{c}"
end

puts inna_metoda(*["a", "b", 3]) #=> a, b, 3
puts inna_metoda("a", *["b", 3]) #=> a, b, 3
```

& czyli przekazywanie bloku

Poznaliśmy już domknięcia i sposoby przekazywania ich do metody. Domknięcie możemy przekazać, definiując je bezpośrednio za nazwą metody. Natomiast obiekt procedurowy możemy przekazywać jako parametr. Wiemy też, że sterowanie do domknięcia przekazujemy przez `yield`, natomiast procedurę obiektu procedurowego wywołujemy przez metodę `call`. Co jednak, gdy chcielibyśmy użyć bloku przekazanego jako domknięcie tak jakby był obiektem (stosując `call` zamiast `yield`)? Albo gdybyśmy chcieli utworzony już obiekt procedurowy przekazać tak jakby był blokiem?

Rozważmy naszą metodę `powtorz` z rozdziału o domknięciach:

```
def powtorz(ilosc)
  while ilosc > 0
    yield ilosc
    ilosc -= 1
  end
end
```

Aby przekazać do tej metody blok, który mamy w postaci np. lambdy, należy użyć symbolu `&` i przekazać nasz blok jako ostatni (niby *fikcyjny*) argument. Fikcyjny, bo nie jest on jawnie zdefiniowany w nagłówku metody.

```
l = lambda { |x| print x }
powtorz(3, &l) #=> 321
```

Efekt jest taki sam jakbyśmy przekazali blok tradycyjnie:
`powtorz(3) { |x| print x } #=> 321`

Symbolu `&` możemy też używać przed ostatnim parametrem w definicji metody. Dzięki temu, możemy uzyskać niejako odwrotne działanie: odwoływać się do bloku jak do obiektu procedurowego:

```
def powtorz(ilosc, &blok)
  while ilosc > 0
    blok.call(ilosc) # to samo co yield ilosc
    ilosc -= 1
  end
end

powtorz(3) { |x| print x } #=> 321

l = lambda { |x| print x }
powtorz(3, &l) #=> 321
```

Jak widzimy, jawne przekazanie obiektu `l` jako bloku również jest poprawne.

Wiele wartości w instrukcji `return`

Na koniec powróćmy jeszcze do niuansów stosowania instrukcji `return`. Domyślnie metoda zwraca ostatnie obliczone wyrażenie, choć można oczywiście zastosować słowo kluczowe `return`, by metoda zwróciła konkretną wartość. Jednak w Rubim, w odróżnieniu od C/C++, Javy czy C# instrukcja `return` może zwracać więcej niż jedną wartość. Rozważmy taki przykład:

```
def metoda
   return "a", 0, "b"
end

tab = metoda
t1, t2 = metoda

puts tab.class   # => Array
puts t1.class    # => String
puts t2.class    # => Fixnum
```

Jak widzimy, jeżeli użyjemy jednej zmiennej metoda zwróci nam tablicę, w której będą wszystkie wartości wyrażeń przekazanych do instrukcji `return`. Jeżeli po lewej stronie przypisania wyniku metody umieścimy więcej niż jedną zmienną (`t1`, `t2`, itd.) będą do nich podstawione kolejne wartości zwracane przez `return`. Jeżeli zmiennych po lewej stronie będzie mniej niż wartości zwracanych przez metodę, "nadmiarowe" *wartości* zostaną zignorowane (jak ma to miejsce wyżej). Jeżeli natomiast będzie ich więcej, "nadmiarowe" *zmienne* dostaną wartości `nil`:

```
a1, a2, a3, a4 = metoda
puts a4.nil? #=> true
```

Rozdział 14

Klasy

Klasy

Świat rzeczywisty wypełniony jest obiektami, które możemy poklasyfikować. Dla przykładu, bardzo małe dziecko mówi "hau-hau" gdy widzi psa, niezależnie od jego rasy. To zupełnie naturalne, że postrzegamy świat w oparciu takie kategorie.

W terminologii programowania obiektowego kategoria obiektów takich jak "pies" nazywana jest klasą, natomiast pewien specyficzny obiekt należący do klasy nazywany jest instancją tej klasy.

Zazwyczaj, by utworzyć jakiś obiekt w Rubim lub w każdym innym języku obiektowym, najpierw należy zdefiniować pewne cechy charakterystyczne klasy, a następnie utworzyć jej instancję. By zilustrować ten przykład, zdefiniujmy prostą klasę `Pies`.

```
class Pies
  def szczekaj
    puts "Hau hau"
  end
end
```

W Rubim, definicja *klasy* jest regionem kodu pomiędzy słowami kluczowymi `class` i `end`. Słowo kluczowe `def` wewnątrz tego regionu rozpoczyna definicję *metody* danej klasy, która, jak omówiliśmy w poprzednim rozdziale, odpowiada pewnemu specyficznemu zachowaniu obiektów tej klasy.

Teraz, jak już zdefiniowaliśmy klasę `Pies` możemy użyć jej do utworzenia psa:

`burek = Pies.new`

Utworzyliśmy nową instancję klasy `Pies` i nazwaliśmy ją `burek`. Metoda `new` jakiejkolwiek klasy tworzy nową instancję. Ponieważ `burek` jest psem (`Pies`) zgodnie z definicją naszej klasy, ma on wszelkie własności, które (jak zdecydowaliśmy) pies powinien posiadać. Ale ponieważ nasza idea "pieskości" była bardzo prosta, jest tylko jedna sztuczka, o którą możemy poprosić naszego psa `burek`.

`burek.szczekaj #=> Hau hau`

Tworzenie nowej instancji jest czasem zwane *instancjonowaniem* klasy. Potrzebujemy mieć jakiegoś psa nim będziemy mogli doświadczyć przyjemności rozmowy z nim. Nie możemy po prostu poprosić klasy `Pies` by dla nas szczekała.

```
Pies.szczekaj
ERR: (eval):1: undefined method ‘szczekaj’ for Pies:class
```

Ma to tyle sensu, co *próba zjedzenia koncepcji kanapki.*

Obiekty anonimowe

Z drugiej strony, jeśli chcemy usłyszeć dźwięk psa bez większego zaangażowania emocjonalnego, możemy utworzyć (instancjonować) efemerycznego, tymczasowego psa, i namówić go do wydania małego odgłosu nim zniknie.

```
(Pies.new).szczekaj   # lub bardziej powszechnie: Pies.new.szczekaj
#=> Hau hau
```

"Zaraz", powiesz, "o co chodzi z tym biednym kolegą, co zniknie za chwilę?" To prawda. Jeżeli nie dbasz o to, by dać mu imię (tak jak daliśmy imię psu `burek`), automatyczne odśmiecanie Rubiego zdecyduje, że jest to niechciany, bezpański pies, i bezlitośnie się go pozbędzie. Tak naprawdę to nic strasznego, ponieważ, jak wiesz, możemy utworzyć wszystkie psy, które tylko chcemy.

Rozdział 15

Dziedziczenie

Dziedziczenie

Klasyfikacja obiektów w codziennym życiu jest ze swojej natury hierarchiczna. Wiemy, że *wszystkie koty są ssakami*, a *wszystkie ssaki są zwierzętami*. Mniejsze klasy *dziedziczą* cechy charakterystyczne po większych, do których należą. Jeżeli wszystkie ssaki oddychają, to również wszystkie koty oddychają.

Możemy wyrazić tę koncepcję w Rubim:

```
class Ssak
  def oddychaj
    puts "wdech i wydech"
  end
end

class Kot < Ssak
  def daj_glos
    puts "Miau"
  end
end
```

Chociaż nie określamy jak `Kot` powinien oddychać, to każdy kot będzie dziedziczył to zachowanie z klasy `Ssak` ponieważ `Kot` został zdefiniowany jako podklasa klasy `Ssak`. (W terminologii obiektowej, mniejsza klasa jest *podklasą*, natomiast większa klasa jest *nadklasą*.) Odtąd, z punktu widzenia programisty, koty uzyskują zdolność oddychania "za darmo". Jak dodamy metodę `daj_glos` nasze koty będą mogły zarówno oddychać oraz mówić.

```
mruczek = Kot.new
mruczek.oddychaj #=> wdech i wydech
mruczek.daj_glos #=> Miau
```

Na pewno wystąpią również takie sytuacje, że niektóre własności nadklasy nie powinny być dziedziczone przez jakąś konkretną podklasę. Chociaż ptaki generalnie potrafią latać, pingwiny są podklasą ptaków nielotnych (nielotów).

```
class Ptak
```

```
  def czysc
    puts "Czyszcze piorka."
  end

  def lataj
    puts "Latam."
  end
end

class Pingwin < Ptak
  def lataj
    fail "Przykro mi. Raczej poplywam."
  end
end
```

Zamiast kompletnie definiować każdą cechę każdej nowej klasy, potrzebujemy jedynie dołączyć, a raczej przedefiniować różnice pomiędzy każdą podklasą a jej nadklasą. Takie użycie dziedziczenia jest czasem nazywane *programowaniem różnicowym*. Jest to jedna z zalet programowania zorientowanego obiektowo.

Rozdział 16

Przedefiniowywanie metod

Przedefiniowywanie metod

W podklasie możemy zmienić zachowanie instancji poprzez przedefiniowanie metod z nadklasy.

```
class Czlowiek
  def przedstaw_sie
    puts "Jestem osoba."
  end
  def koszt_biletu(wiek)
    if wiek < 12
      puts "Oplata ulgowa.";
    else
      puts "Oplata normalna.";
    end
  end
end

Czlowiek.new.przedstaw_sie   #=> Jestem osoba.

class Student1 < Czlowiek
  def przedstaw_sie
    puts "Jestem studentem."
  end
end

Student1.new.przedstaw_sie        #=> Jestem studentem.
```

Przypuszczalnie moglibyśmy tylko ulepszyć metodę `przedstaw_sie` z nadklasy zamiast całkowicie ją podmieniać. Do tego celu możemy użyć słowa kluczowego `super`.

```
class Student2 < Czlowiek
  def przedstaw_sie
    super
    puts "Jestem rowniez studentem."
```

```
  end
end

Student2.new.przedstaw_sie
#=> Jestem osoba.
#   Jestem rowniez studentem.
```

`super` pozwala nam przekazywać argumenty do oryginalnej metody. Czasem mówi się, że ludzi dzielimy na dwa rodzaje...

```
class Nieuczciwy < Czlowiek
  def koszt_biletu(wiek)
    super(11)   # chcemy skromna oplate
  end
end

Nieuczciwy.new.koszt_biletu(25) #=> Oplata ulgowa.

class Uczciwy < Czlowiek
  def koszt_biletu(wiek)
    super(wiek) # przekazujemy argument ktory dostalismy
  end
end

Uczciwy.new.koszt_biletu(25)    #=> Oplata normalna.
```

Rozdział 17

Kontrola dostępu

Kontrola dostępu

Wcześniej powiedzieliśmy, że Ruby nie posiada funkcji, tylko metody. Jednakże jest nieco więcej różnych rodzajów metod. W tym rozdziale przedstawimy sposoby kontroli dostępu.

Rozważmy, co się stanie, gdy zdefiniujemy metodę na samym szczycie hierarchii, nie wewnątrz jakiejkolwiek klasy? Możemy myśleć o takiej metodzie analogicznie jak o funkcji w bardziej tradycyjnym języku, takim jak C.

```
def kwadrat(n)
  n * n
end

kwadrat(5) #=> 25
```

Wydaje się, że nasza nowa metoda nie należy do żadnej klasy, ale w rzeczywistości Ruby dodał ją do klasy `Object` która jest nadklasą każdej innej klasy. W rezultacie każdy obiekt powinien mieć możliwość używania tej metody. Jest to prawdą, ale z małym kruczkiem: jest to *prywatna* metoda każdej klasy. Wprawdzie będziemy jeszcze dokładnie rozważać co to znaczy, ale zauważmy, że jedną z konsekwencji tego faktu jest to, że metoda ta może być wywołana tylko w funkcyjnym stylu, jak poniżej:

```
class Klasa
  def czwarta_potega_z(x)
    kwadrat(x) * kwadrat(x)
  end
end

Klasa.new.czwarta_potega_z(10) #=> 10000
```

Wyraźnie nie możemy wywołać metody na rzecz obiektu:

```
"ryba".kwadrat(5)
ERR: (eval):1: private method ‘kwadrat’ called for "ryba":String
```

To raczej zręcznie chroni czysto obiektową naturę Rubiego (funkcje są wciąż metodami obiektów, ale odbiorcą domyślnie jest `self`) dostarczając funkcji które mogą być zapisane podobnie jak w bardziej tradycyjnym języku.

Powszechną dyscypliną umysłową w programowaniu obiektowym, którą zasugerowaliśmy we wcześniejszym rozdziale, jest problem rozdzielenia *specyfikacji* i *implementacji*, czyli *jakie* zadania wymagamy by obiekt wypełniał i *jak* je właściwie wypełnia. Wewnętrzne prace obiektu powinny być zazwyczaj ukryte przed jego użytkownikami. Powinni oni dbać o to, co wchodzi i wychodzi do/z obiektu oraz ufać, że obiekt wie co robi wewnętrznie z danymi. Z tego powodu często pomocne jest, gdy klasa posiada metody niewidoczne z zewnątrz, ale używane wewnętrznie, które mogą być poprawione przez programistę kiedy tylko zajdzie taka potrzeba, bez zmieniania sposobu, w jaki użytkownicy widzą obiekty danej klasy. W trywialnym przykładzie poniżej, pomyśl o metodzie `wylicz` jako o niewidocznych pracach klasy.

```
class Test
  def dwa_razy(a)
    puts "#{a} razy dwa to #{wylicz(a)}"
  end

  def wylicz(b)
    b*2
  end

  private :wylicz  # to ukryje wylicz przed uzytkownikami
end

test = Test.new
<source>

<source lang="ruby">
test.wylicz(6)
ERR: (eval):1: private method ‘wylicz’ called for #<Test:0x4017181c>

test.dwa_razy(6)
#=> 6 razy dwa to 12
```

Moglibyśmy oczekiwać, że `test.wylicz(6)` zwróci `12`, ale zamiast tego nauczyliśmy się, że metoda `wylicz` jest niedostępna, gdy odgrywamy rolę użytkownika obiektu `Test`. Tylko inne metody klasy `Test`, takie jak `dwa_razy` mogą korzystać z `wylicz`. Od nas wymagane jest posługiwanie się publicznym interfejsem, który składa się z metody `dwa_razy`. Programista, który jest pod kontrolą tej klasy może swobodnie modyfikować `wylicz` (tutaj, być może zmieniając `b*2` na `b+b` i argumentując to przypuszczalnie wzrostem wydajności) bez wpływania na to jak użytkownik współdziała z obiektami klasy `Test`. Ten przykład jest oczywiście zbyt prosty by był użyteczny; korzyści z metod kontroli dostępu staną się bardziej widoczne tylko wtedy, gdy zaczniemy tworzyć bardziej skomplikowane i interesujące klasy.

Modyfikatory dostępu

W Rubim mamy dostępne trzy modyfikatory dostępu.

Modyfikator	Tłumaczenie	Działanie
`private`	prywatny	Metoda dostępna tylko dla obiektu danej klasy. Każdy obiekt danej klasy może wywoływać metody prywatne tylko na rzecz samego siebie. W innych językach programowania (np. w Javie) obiekty tej samej klasy mogą wykonywać swoje metody prywatne.
`protected`	chroniony	Metoda dostępna dla wszystkich obiektów danej klasy i klas potomnych.
`public`	publiczny	Metoda dostępna dla wszystkich obiektów.

W Rubim modyfikatory dostępu dotyczą tylko metod. Jeżeli nie stosujemy żadnych modyfikatorów dostępu, domyślnym modyfikatorem jest *public*, tak więc wszystkie metody (poza `initialize`) są domyślnie publiczne. Dostęp do metod można określać na dwa sposoby:

1. Stosując słowa kluczowe przed definicjami metod

2. Stosując symbole o nazwach metod

Modyfikatory przed definicjami metod

```
class Test
  public # każda metoda (poza initialize, która jest prywatna) jest domyślnie publiczna

  def pub_met1
  end

  def pub_met2
  end

  private # metody prywatne

  def priv_met1
  end

  protected # metody chronione

  def prot_met1
  end

  def prot_met2
  end
end
```

Jak widzimy, modyfikatory dostępu (`private`, `protected`, `public`) występują tu przed definicjami metod.

Modyfikatory z symbolami

Identyczny efekt (z tym, że dostęp będzie nadawany dynamicznie) można uzyskać stosując modyfikatory oraz nazwy metod jako *symbole*. Co to są symbole powiemy dokładnie w rozdziale o symbolach.

```
class Test
  def pub_met1
  end

  def pub_met2
  end

  def priv_met1
  end

  def prot_met1
  end

  def prot_met2
  end

  public :pub_met1, :pub_met2
  private :priv_met1
  protected :prot_met1, :prot_met2
end
```

Nazwy metod po modyfikatorach poprzedzone są dwukropkami (`:`) — tak właśnie oznaczamy symbole. Ale czym są owe symbole? Dowiedzmy się!

Rozdział 18

Symbole

Symbole

Rozdział 19

Metody singletonowe

Metody singletonowe

Zachowanie instancji zdeterminowane jest przez jej klasę, ale mogą być sytuacje, w których wiemy, że konkretna instancja powinna mieć specjalne zachowanie. W większości języków musimy niestety definiować nową klasę, która byłaby wtedy instancjonowana tylko raz. W Rubim możemy dać każdemu obiektowi jego własne metody.

```
class SingletonTest
  def rozmiar
    25
  end
end

test1 = SingletonTest.new
test2 = SingletonTest.new

def test2.rozmiar
  10
end

test1.rozmiar #=> 25
test2.rozmiar #=> 10
```

W tym przykładzie obiekty `test1` i `test2` należą do tej samej klasy, ale `test2` otrzymał przedefiniowaną metodę `rozmiar`, więc obiekty zachowują się odmiennie. Metoda dana pojedynczemu obiektowi nazywana jest *metodą singletonową*. Nazwa ta może kojarzyć się z wzorcem projektowym singletonu, jednak wymaga nieco więcej wyjaśnienia. Każdy obiekt w Rubim posiada swoją klasę prototypową, która jest również obiektem i którą można modyfikować (indywidualnie dla każdego obiektu). W tym sensie każda para obiekt-klasa prototypowa jest singletonem, czyli każdy obiekt jest tylko jedną jedyną instancją swojej klasy prototypowej. Aby otworzyć definicję klasy prototypowej używamy <<. Moglibyśmy równoważnie zapisać powyższy przykład w ten sposób:

```
class << test2
```

```
  def rozmiar
    10
  end
end
```

Metody singletonowe są często używane w elementach graficznego interfejsu użytkownika (ang. *GUI*), gdzie, w zależności od naciśniętych przycisków muszą być podejmowane odpowiednie akcje.

Metody singletonowe nie występują jedynie w Rubim. Pojawiają się również w CLOS, Dylanie, itd. Ponadto, niektóre języki, dla przykładu Self i NewtonScript, posiadają tylko metody singletonowe. Są one czasem nazywane *językami prototypowymi*.

Rozdział 20

Moduły

Moduły

Moduły w Rubim są podobne do klas, ale:

- Moduł nie może mieć instancji.
- Moduł nie może mieć podklas.
- Moduł jest definiowany przez słowa kluczowe `module` i `end`.

Właściwie to... klasa `Module` modułu jest nadklasą klasy `Class` klasy[1]. Rozumiesz? Nie? Idźmy dalej.

Istnieją dwa typowe zastosowania modułów. Jedno to zebranie powiązanych metod i stałych w jednym centralnym miejscu. Moduł `Math` z standardowej biblioteki Rubiego odgrywa taką rolę:

```
irb(main):001:0> Math.sqrt(2)
=> 1.4142135623731
irb(main):002:0> Math::PI
=> 3.14159265358979
irb(main):003:0>
```

Operator `::` mówi interpreterowi Rubiego, który moduł powinien on sprawdzić by pobrać wartość zmiennej (możliwe, że jakiś moduł oprócz `Math` może użyć `PI` do oznaczenia czegoś innego). Jeżeli chcemy odnieść się do metod lub stałych modułu bezpośrednio, bez używania `::`, możemy "zawrzeć" ten moduł używając słowa kluczowego `include`:

```
irb(main):003:0> include Math
=> Object
irb(main):004:0> sqrt(2)
=> 1.4142135623731
irb(main):005:0> PI
=> 3.14159265358979
irb(main):006:0>
```

[1] To ma sens :). W Rubim obiektami są nawet klasy lub moduły i posiadają one swoje klasy (`Module` i `Class`).

Domieszkowanie klas

Inne użycie modułu nazywane jest *domieszkowaniem* klas (ang. *mixin*). Niektóre języki obiektowe, włączając C++ lub Eiffel, pozwalają na wielokrotne dziedziczenie, to znaczy, dziedziczenie po więcej niż jednej nadklasie. Przykładem wielokrotnego dziedziczenia z codziennej rzeczywistości jest budzik. Możesz zaliczyć budziki do klasy zegarków jak i do klasy przedmiotów z brzęczykami.

Ruby celowo nie implementuje prawdziwego wielokrotnego dziedziczenia, ale domieszkowanie klas jest dobrą alternatywą. Pamiętaj, że moduły nie mogą posiadać instancji ani podklas. Ale jeśli włączymy (`include`) moduł w definicję klasy, jego metody będą efektywnie dodane czy też "wmieszane" w klasę.

Domieszkowanie klas może być rozważane jako odpowiedź na pytanie o wszelkie partykularne własności, które chcemy mieć. Na przykład, jeżeli klasa ma działającą metodę `each`, zmieszanie jej ze standardowym modułem `Enumerable` da nam dodatkowo metody `sort` oraz `find`. Dzieje się tak, ponieważ metody z modułu `Enumerable` używają właśnie metody `each`.

Takie użycie modułów dostarcza podstawowej funkcjonalności wielokrotnego dziedziczenia, pozwalając jednocześnie, by relacje pomiędzy klasami były nadal reprezentowane za pomocą prostych struktur drzewiastych. W ten sposób upraszcza się znacząco implementacja języka (podobny punkt widzenia został przyjęty przez projektantów Javy). Domieszkowanie wydaje się poza tym dużo wygodniejsze niż wielokrotne dziedziczenie i dużo efektywniejsze niż np. stosowane w C# i Javie — interfejsy.

Przykładowy moduł

Zrealizujmy nasz przykład z zegarkami.

```
module Brzeczyk
  def dzwon
    puts "BZZZZ!BZZZZ!BZZZZ!"
  end
end

class Czasomierz
  def podaj_czas
    puts Time.now
  end
end

class Budzik < Czasomierz
  include Brzeczyk
end

b = Budzik.new
b.podaj_czas #=> Sun Aug 05 17:24:08 +0200 2007
b.dzwon      #=> BZZZZ!BZZZZ!BZZZZ!
```

Jak widzimy, `Budzik` dziedziczy po klasie `Czasomierz` oraz zawiera metody (akurat tylko jedną) z modułu `Brzeczyk`. Dzieje się tak dzięki wspomnianej już metodzie `include`. A co, jeśli chcielibyśmy dodać brzęczyk do jednego tylko obiektu danej klasy?

W tym celu możemy użyć metody `extend(module)` która dodaje metody modułu do konkretnego obiektu.

```
zegarek = Czasomierz.new
zegarek.extend(Brzeczyk)
zegarek.dzwon #=> BZZZZ!BZZZZ!BZZZZ!
```

Metoda `extend` może być również używana tak jak `include`, jednak jej działanie jest nieco inne. Otóż wszystkie metody niestatyczne z modułów przekazanych do `extend` zostaną włączone do klasy (lub modułu) jako metody klasowe (*statyczne*). O metodach klasowych powiemy jeszcze przy okazji omawiania zmiennych klasowych.

```
class Buczek
  extend Brzeczyk
end

Buczek.dzwon #=> BZZZZ!BZZZZ!BZZZZ!
```

Rozdział 21

Zmienne

Zmienne

Ruby ma cztery rodzaje zmiennych, jeden rodzaj stałych i dokładnie dwa rodzaje pseudo-zmiennych. Zmienne i stałe nie mają typu. Owszem, niestypizowane zmienne mają kilka wad, jednak ilość korzyści znacznie je przewyższa. Zmienne niestypizowane dobrze pasują do szybkiej i prostej filozofii Rubiego.

Zmienne muszą być zadeklarowane w większości języków w celu określenia ich typu, zdolności do modyfikacji (np. czy są stałymi), oraz zasięgu. Odkąd jednak typ nie jest problemem a cała reszta jest widoczna z nazwy zmiennej, co za chwilę zobaczysz, nie potrzebujemy deklaracji zmiennych w Rubim.

Na pierwszy rzut oka można określić kategorię zmiennej, ponieważ pierwszy znak lub znaki nazwy zmiennej (identyfikatora) charakteryzują ją:

Znak(i)	**Rodzaj zmiennej**
`$`	zmienna globalna
`@@`	zmienna klasowa
`@`	zmienna instancji
`[a-z]` lub _	zmienna lokalna
`[A-Z]`	stała

Jedynymi wyjątkami od podanych zasad są pseudo-zmienne Rubiego: `self` — która zawsze wskazuje bieżąco wykonywany obiekt oraz `nil` — która jest nieznaczącą wartością przypisywaną niezainicjalizowanym zmiennym. Obie są nazwane tak, jakby były lokalnymi zmiennymi, ale `self` jest zmienną globalną zarządzaną przez interpreter, natomiast `nil` jest tak naprawdę stałą. Jako, że istnieją tylko te dwa wyjątki, nie powinny one zbytnio gmatwać powyższych konwencji.

Do `self` lub `nil` nie możesz przypisać żadnych wartości. `main`, jako wartość `self` wskazuje tu na bieżący główny obiekt:

```
irb(main):001:0> self
=> main
irb(main):002:0> nil
=> nil
```

Rozdział 22

Zmienne globalne

Zmienne globalne

Zmienna globalna posiada nazwę zaczynającą się znakiem `$`. Można się do niej odwołać z dowolnego punktu programu. Przed inicjalizacją zmienna globalna ma specjalną wartość: `nil`.

```
irb(main):001:0> $a
=> nil
irb(main):002:0> $a = 5
=> 5
irb(main):003:0> $a
=> 5
```

Zmienne globalne powinny być używane oszczędnie. Są one niebezpieczne, ponieważ mogą być zapisane z dowolnego miejsca programu. Nadmierne użycie zmiennych globalnych może sprawić, że izolowanie błędów będzie trudne, a ponadto może wskazywać, że projekt programu nie został gruntownie przemyślany. Ilekroć uznasz za niezbędne użycie zmiennej globalnej, upewnij się, że dasz jej opisową nazwę która nie będzie skłaniać do użycia jej potem niezgodnie z twoimi intencjami. Na przykład, nazwanie zmiennej globalnej "`$a`" będzie prawdopodobnie złym pomysłem.

Jedną z przyjemnych cech zmiennych globalnych jest to, że mogą być śledzone. Możesz określić procedurę, która będzie wywoływana za każdym razem, gdy wartość zmiennej ulegnie zmianie.

```
irb(main):004:0> trace_var :$a, lambda { puts "$a wynosi teraz #{$a}" }
=> nil
irb(main):005:0> $a = 0
$a wynosi teraz 0
=> 0
```

Jeśli zmienna globalna została przeznaczona do pracy jako przełącznik wywołujący procedurę, gdy tylko zmieni się jej wartość, nazywamy ją czasem *aktywną zmienną*. Takie użycie zmiennych może być całkiem użyteczne do np. aktualizacji wyświetlania GUI.

Nazwa	Opis
`$!`	informacja o ostatnim błędzie
`$@`	położenie błędu
`$_`	ostatni łańcuch przeczytany przez `gets`
`$.`	numer linii przeczytany ostatnio przez interpreter
`$&`	ostatni łańcuch dopasowany przez wyrażenie regularne
`$`	ostatnie dopasowanie do wyrażanie regularnego, jako tablica podwyrażeń
`$n`	`n`-te podwyrażenie w ostatnim `match` (to samo co `$ [n]`)
`$=`	flaga niewrażliwości na duże/małe litery
`$/`	separator danych wejściowych
`$\`	separator danych wyjściowych
`$0`	nazwa bieżącego skryptu Rubiego
`$*`	argumenty z linii poleceń
`$$`	identyfikator procesu interpretera
`$?`	status wyjściowy ostatniego wykonanego procesu potomka

Istnieje zbiór specjalnych zmiennych, który nazwy składają się ze znaku dolara (`$`) oraz jakiegoś pojedynczego znaku. Dla przykładu, `$$`. zawiera identyfikator procesu interpretera Rubiego i jest tylko do odczytu. Oto główne zmienne systemowe:

W powyższym `$_` i `$` mają zasięg **lokalny**. Choć ich nazwy sugerują, że powinny być globalne, to z wygody jak i pewnych historycznych zaszłości są używane w ten sposób.

Są jeszcze dwie specjalne zmienne systemowe:

Nazwa	Opis
`__FILE__`	zwraca nazwę bieżącego pliku (nie skryptu, z którego został uruchomiony program)
`__LINE__`	zwraca numer bieżącej linii

Rozdział 23

Zmienne klasowe

Zmienne klasowe

Zmienne *klasowe* są współdzielone przez wszystkie instancje danej klasy. W Rubim, do oznaczania zmiennych klasowych używa się prefiksu `@@`. W poniższym przykładzie zmienna `@@populacja` odzwierciedla całkowitą ilość instancji klasy `Zwierze` utworzonych w czasie działania programu.

```
class Zwierze
  @@ilosc = 0

  def initialize
    @@ilosc += 1
  end

  def self.populacja # metoda klasy
    @@ilosc
  end

  def populacja      # metoda instancji
    @@ilosc
  end
end

3.times do
  Zwierze.new
end

puts Zwierze.populacja # => 3
pies = Zwierze.new
puts pies.populacja    # => 4
```

`@@ilosc` jest zmienną klasową. W konstruktorze klasy `Zwierze` zwiększamy ją o jeden, czyli *inkrementujemy* (zmniejszanie o jeden nazywamy *dekrementacją*). Zapis `@@ilosc += 1` jest równoważny zapisowi `@@ilosc = @@ilosc + 1`. Jak widzimy, każde kolejne utworzenie instancji klasy `Zwierze` powoduje zwiększenie o jeden zmiennej `@@ilosc`.

Metoda zdefiniowana jako `self.populacja` jest *metodą klasową*. Inną metodą tego rodzaju jest np. `new`. Metoda klasowa może być wywołana jedynie na rzecz klasy, nie na rzecz obiektu. W naszym przykładzie równoważną metodą, ale nie klasową, jest metoda `populacja`. Obie zwracają wartość naszej zmiennej klasowej.

Rozdział 24

Zmienne instancji

Zmienne instancji

Zmienna instancji ma nazwę zaczynającą się znakiem `@`, a jej zasięg zasięg ograniczony jest tylko do obiektu wskazywanego przez `self`. Dwa różne obiekty, nawet jeśli należą do tej samej klasy, mogą mieć różne wartości swoich zmiennych instancji. Na zewnątrz obiektu zmienne instancji nie mogą być zmienione, ani nawet odczytane (zmienne instancji w Rubim nigdy nie są publiczne), chyba że za pośrednictwem metod wyraźnie dostarczonych przez programistę. Tak jak zmienne globalne, zmienne instancji mają wartość `nil` dopóki nie zostaną zainicjalizowane.

Zmienne instancji nie muszą być zadeklarowane. Wskazuje to na elastyczną strukturę obiektu. W rzeczywistości, każda zmienna instancji jest dynamicznie dołączana do obiektu w czasie pierwszego przypisania jej wartości.

```
irb(main):001:0> class TestInst
irb(main):002:1> def ustaw_a(n)
irb(main):003:2> @a = n
irb(main):004:2> end
irb(main):005:1> def ustaw_b(n)
irb(main):006:2> @b = n
irb(main):007:2> end
irb(main):008:1> end
=> nil
irb(main):009:0> i = TestInst.new
=> #<TestInst:0x2e46770>
irb(main):010:0> i.ustaw_a(2)
=> 2
irb(main):011:0> i
=> #<TestInst:0x2e46770 @a=2>
irb(main):012:0> i.ustaw_b(4)
=> 4
irb(main):013:0> i
=> #<TestInst:0x2e46770 @b=4, @a=2>
```

Zauważ, że powyższe `i` nie zwraca wartości `@b` dopóki nie zostanie wywołana metoda `ustaw_b`.

Wiedzę o zmiennych instancji możemy wykorzystać do udoskonalenia naszej klasy `Zwierze` z rozdziału o zmiennych klasowych:

```
class Zwierze
  @@ilosc = 0

  def initialize(nazwa, wiek)
    @nazwa = nazwa
    @wiek = wiek
    @@ilosc += 1
  end

  def self.populacja
    puts "Populacja zwierzat liczy sztuk: #{@@ilosc}"
  end

  def podaj_nazwe
    @nazwa
  end

  def podaj_wiek
    @wiek
  end

  def to_s
    "Jestem #{@wiek}-letni #{podaj_nazwe}"
  end
end

pies = Zwierze.new("pies", 3)
kot = Zwierze.new("kot", 5)

puts pies         #=> Jestem 3-letni pies
puts kot          #=> Jestem 5-letni kot
Zwierze.populacja #=> Populacja zwierzat liczy sztuk: 2
```

Rozdział 25

Zmienne lokalne

Zmienne lokalne

Zmienna lokalna posiada nazwę zaczynającą się małą literą lub znakiem podkreślenia (_). Zmienne lokalne nie posiadają, w przeciwieństwie do zmiennych globalnych i zmiennych instancji, wartości `nil` przed inicjalizacją:

```
irb(main):001:0> $a
=> nil
irb(main):002:0> @a
=> nil
irb(main):003:0> a
NameError: undefined local variable or method ‘a’ for main:Object
        from (irb):3
```

Pierwsze przypisanie do zmiennej lokalnej odgrywa rolę jakby deklaracji. Jeżeli odwołasz się do niezainicjalizowanej zmiennej lokalnej, interpreter Rubiego nie będzie pewny, czy odwołujesz się do prawdziwej zmiennej. Możesz, dla przykładu, zrobić błąd w nazwie metody. Dlatego zobaczyłeś raczej ogólną informację o błędzie.

Zazwyczaj zasięgiem zmiennej lokalnej jest jedno z poniższych:

- `proc { ... }`
- `lambda { ... }`
- `loop { ... }`
- `def ... end`
- `class ... end`
- `module ... end`
- cały skrypt (chyba że zastosowano jedno z powyższych)

Użyty w następnym przykładzie `defined?` jest operatorem, który sprawdza czy identyfikator został zdefiniowany. Zwraca on opis identyfikatora, jeśli jest on zdefiniowany lub, w przeciwnym razie, `nil`. Jak widzisz, zasięg zmiennej `b` jest ograniczony lokalnie do pętli. Gdy pętla zostaje przerwana zmienna `b` jest niezdefiniowana.

```
a = 44
puts defined?(a) #=> local-variable

loop do
  b=45
  break
end
puts defined?(b) #=> nil
```

Obiekty procedurowe, które żyją w pewnym zakresie widoczności współdzielą tylko te zmienne lokalne, które również należą do tego zakresu. Tutaj, zmienna lokalna `a` jest współdzielona przez `main` oraz obiekty procedurowe `l1` i `l2`:

```
a = nil
l1 = lambda {|n| a=n}
l2 = lambda {a}
l1.call(5)
puts a       #=> 5
puts l2.call #=> 5
```

Zauważ, że nie można pominąć `a = nil` na początku. To przypisanie zapewnia, że zasięg zmiennej `a` obejmie `l1` i `l2`. Inaczej `l1` i `l2` utworzyłyby swoje własne zmienne lokalne `a`, i rezultatem wywołania `l2` byłby błąd "undefined local variable or method" (niezdefiniowana zmienna lokalna lub metoda). Moglibyśmy użyć `a = 0`, ale użycie `nil` jest pewną uprzejmością wobec przyszłych czytelników naszego kodu. Pokazuje naprawdę jasno, że tylko ustanawiamy zakres, ponieważ wartość przypisana do zmiennej nie niesie żadnego specjalnego znaczenia (`nil`).

Domknięcia a kontekst

W rozdziale o domknięciach i obiektach procedurowych wspomnieliśmy, że domknięcia i obiekty procedurowe zachowują kontekst używanych zmiennych lokalnych. Jest to bardzo potężna zaleta: współdzielone zmienne lokalne pozostają poprawne nawet wtedy, gdy przekazane są poza pierwotny zakres.

```
def pudelko
  zawartosc = nil
  wez = lambda { zawartosc }
  wloz = lambda { |n| zawartosc = n }
  return wez, wloz
end

odczyt, zapis = pudelko

puts odczyt.call   #=> nil
zapis.call(2)
puts  odczyt.call  #=> 2
```

Ruby jest szczególnie sprytny jeśli chodzi o zakres. W naszym przykładzie ewidentnie widać, że zmienna `zawartosc` jest współdzielona pomiędzy `odczyt` i `zapis`.

Możemy również wytworzyć wiele par `odczyt-zapis` używając metody `pudelko` zdefiniowanej powyżej. Każda para współdzieli zmienną `zawartosc`, ale pary nie kolidują ze sobą nawzajem. Dopóki istnieją obiekty procedurowe, zachowane są ich konteksty wywołania wraz z odpowiadającymi im zmiennymi lokalnymi!

```
odczyt_1, zapis_1 = pudelko
odczyt_2, zapis_2 = pudelko

zapis_1.call(99)
puts odczyt_1.call  #=> 99

# w tym pudelku jeszcze nic nie ma
puts odczyt_2.call  #=> nil
```

Powyższy kod można by wręcz uważać za lekko perwersyjny zorientowany obiektowo framework. Metoda `pudelko` odgrywa rolę klasy podczas gdy `wez` i `wloz` służą jako metody, natomiast `zawartosc` jest samotną zmienną instancji. Oczywiście stosowanie właściwego szkieletu klas Rubiego prowadzi do znacznie bardziej czytelnego kodu ;).

Rozdział 26

Stałe klasowe

Stałe klasowe

Nazwa stałej zaczyna się od dużej litery. Stałej nie powinno przypisywać się wartości więcej niż jeden raz. W bieżącej implementacji Rubiego ponowne przypisanie wartości do stałej generuje ostrzeżenie, ale nie błąd:

```
irb(main):001:0> zmienna = 30
=> 30
irb(main):002:0> zmienna = 31
=> 31
irb(main):003:0> Stala = 32
=> 32
irb(main):004:0> Stala = 33
(irb):4: warning: already initialized constant Stala
=> 33
```

Stałe mogą być definiowane wewnątrz klas, ale w przeciwieństwie do zmiennych instancji lub zmiennych klasowych, są one dostępne na zewnątrz klasy.

```
class Stale
  S1=101
  S2=102
  S3=103
  def pokaz
    puts "#{S1} #{S2} #{S3}"
  end
end

#puts S1        #=> .../main.rb:9: uninitialized constant S1 (NameError)

puts Stale::S1  #=> 101
Stale.new.pokaz #=> 101 102 103
```

Stałe mogą być również definiowane w modułach.

```
module StaleModul
```

```
  S1=101
  S2=102
  S3=103
  def pokaz_stale
    puts "#{S1} #{S2} #{S3}"
  end
end

#put S1             #=> .../main.rb:10: uninitialized constant S1 (NameError)

include StaleModul
puts S1             #=> 101
pokaz_stale         #=> 101 102 103
S1=99               # nie najlepszy pomysl
puts S1             #=> 99
StaleModul::S1      #=> 101
StaleModul::S1=99   # ...to nie bylo mozliwe we wczesniejszych wersjach
#=> /main.rb:10: uninitialized constant S1 (NameError)

#pelna swoboda by strzelic sobie w stope
puts StaleModul::S1  #=> 99
```

Rozdział 27

Przetwarzanie wyjątków: rescue

Przetwarzanie wyjątków: rescue

Wykonujący się program może napotkać na niespodziewane problemy. Plik, które chce odczytać może nie istnieć, dysk może być pełny, gdy trzeba zapisać dane, a użytkownik wprowadza niepoprawny rodzaj danych wejściowych.

```
irb(main):001:0> plik = open('jakis_plik')
Errno::ENOENT: No such file or directory - jakis_plik
        from (irb):1:in `initialize'
        from (irb):1:in `open'
        from (irb):1
```

Solidny program powinien radzić sobie z takimi sytuacjami sensownie i wdzięcznie. Sprostanie temu wymaganiu może być irytującym zadaniem. Od programistów języka C oczekuje się sprawdzania wyniku każdego wywołania systemowego które potencjalnie mogło się nie powieść oraz natychmiastowego zdecydowania, co należy zrobić:

```
FILE *plik = fopen("jakis_plik", "r");
if (plik == NULL) {
  fprintf( stderr, "Plik nie istnieje.\n" );
  exit(1);
}
bajty_przeczytane = fread( buf, 1, bajty_zadane, file );
if (bajty_przeczytane != bajty_zadane) {
  /* tutaj wiecej kodu obslugi bledow... */
}
...
```

Jest to bardzo męcząca praktyka, którą programiści mają w zwyczaju traktować niedbale i pomijać, czego rezultatem jest to, że program źle sobie radzi z wyjątkami. Z drugiej strony, porządne wykonanie tej pracy czyni programy trudnymi do czytania, ponieważ duża ilość kodu obsługi wyjątków przesłania właściwą logikę programu.

W Rubim, tak jak w wielu współczesnych językach programowania, możemy radzić sobie z wyjątkami poszczególnych bloków kodu oddzielnie, co jednak skutecznie acz

nie nadmiernie obciąża programistę lub każdego, kto będzie potem czytał kod. Blok kodu oznaczony słowem `begin` wykonuje się dopóki nie napotka na wyjątek, który powoduje przekierowanie kontroli do bloku zarządzania błędami, rozpoczynającego się od `rescue`. Jeżeli nie wystąpi żaden wyjątek, kod z bloku `rescue` nie jest używany. Następująca metoda zwraca pierwszą linię z pliku tekstowego lub `nil` jeżeli napotka wyjątek:

```
def pierwsza_linia(nazwa_pliku)
  begin
    plik = open(nazwa_pliku)
    info = plik.gets
    plik.close
    info  # Ostatnia obliczona rzecz jest zwracana
  rescue
    nil   # Nie mozesz przeczytac pliku? więc nie zwracaj łancucha
  end
end
```

Będą występować sytuacje, gdy będziemy chcieli kreatywnie pracować nad problemem. Tutaj, jeśli plik, który żądamy jest niedostępny, możemy spróbować użyć standardowego wejścia:

```
begin
  plik = open("jakis_plik")
rescue
  plik = STDIN
end

begin
  # ... przetwarzaj wejscie ...
rescue
  # ... tutaj obsluguj wyjatki.
end
```

Słowo kluczowe `retry` może być używane w bloku `rescue`, by wystartować blok `begin` od początku. Pozwala to nam przepisać poprzedni przykład nieco zwięźlej:

```
nazwap = "jakis_plik"
begin
  plik = open(nazwap)
  # ... przetwarzaj wejscie ...
rescue
  nazwap = "STDIN"
  retry
end
```

Jednakże, mamy tutaj pewną wadę. Nieistniejący plik sprawi, że pętla ta będzie powtarzana w nieskończoność. Musisz zwracać uwagę na tego rodzaju pułapki podczas przetwarzania wyjątków.

Każda biblioteka Rubiego podnosi wyjątek jeśli wystąpi jakiś błąd. Ty również możesz podnosić wyjątki jawnie w kodzie. By podnieść wyjątek użyj słowa kluczowego

`raise`. Przyjmuje ono jeden argument, którym powinien być łańcuch znakowy opisujący wyjątek. Argument jest wprawdzie opcjonalny, jednak nie powinien być pomijany. Będzie on mógł być później dostępny za pomocą specjalnej zmiennej globalnej `$!`.

```
begin
  raise "test2"
rescue
  puts "Wystapil blad: #{$!}"
end
#=> Wystapil blad: test2
```

Zmienna `$!` zwraca konkretny obiekt który jest podklasą klasy `Exception`. Klasa `Exception` jest nadklasą (niekoniecznie wprost) wszystkich wyjątków. Cechę tę można efektywnie wykorzystać podczas definiowania różnych bloków obsługujących poszczególne typy wyjątków.

```
begin
  plik = open("jakis_plik")
rescue SystemCallError
  puts "Blad WE/WY: #{$!}"
rescue Exception
  puts "Blad: #{$!}"
end
#=> Blad WE/WY: No such file or directory - jakis_plik
```

Operacja otwarcia pliku generuje wyjątek będący podklasą `SystemCallError` więc zostanie wykonany blok obsługujący ten wyjątek. Interpreter po kolei sprawdza wszystkie bloki `rescue` i wykonuje pierwszy pasujący. Z tego też powodu nie należy umieszczać `rescue Exception` jako pierwszego, gdyż `Exception` jako nadklasa wszystkich wyjątków będzie tu zawsze pasować i blok obsługujący `Exception` będzie zawsze wykonywany.

Jeżeli podmienimy `plik = open(‘‘jakis_plik’’)` na np. `raise ‘‘jakis blad’’` wykonany zostanie blok `rescue` obsługujący `Exception`:

```
begin
  raise "jakis blad"
rescue SystemCallError
  puts "Blad WE/WY: #{$!}"
rescue Exception
  puts "Blad: #{$!}"
end
#=> Blad: jakis blad
```

Zamiast zmiennej `$!` można używać zmiennych nazwanych stosując operator `=>` i składnię przypominającą definiowanie wpisu tablicy asocjacyjnej.

```
begin
  # ... jakis kod ...
rescue SystemCallError => e
  puts "Blad we/wy: #{e}"
rescue Exception => e
  puts "Blad: #{e}"
end
```

Rozdział 28

Przetwarzanie wyjątków: ensure

Przetwarzanie wyjątków: ensure

Może się tak zdarzyć, że potrzebne jest dodatkowe sprzątanie, gdy metoda kończy swoją pracę. Być może otwarty plik powinien być zamknięty, bufor opróżniony, itp. Jeżeli byłby zawsze tylko jeden punkt wyjścia dla każdej metody, moglibyśmy z pełną ufnością umieścić kod czyszczący w jednym miejscu i wiedzielibyśmy, że zostanie on wykonany. Jednakże, metoda może zwracać wartość z różnych miejsc, lub nasze zamierzone czyszczenie może być niespodziewane ominięte z powodu wyjątku.

```
begin
  plik = open("/tmp/jakis_plik", "w")
  # ... zapis do pliku ...
  plik.close
end
```

W powyższym przykładzie, jeżeli wyjątek wystąpiłby w sekcji kodu, w której dokonujemy zapisu do pliku, plik mógłby pozostać otwarty. A my nie chcemy uciekać się tego rodzaju redundancji:

```
begin
  plik = open("/tmp/jakis_plik", "w")
  # ... zapis do pliku ...
  plik.close
rescue
  plik.close
  fail # ponownie podnosi przechwycony wyjatek
end
```

Ten kod nie dość, że niezdarny, będzie jeszcze bardziej skomplikowany, ponieważ trzeba będzie obsługiwać każdy `return` i `break`.

Z tego powodu dodamy nowe słowo kluczowe do naszego schematu "`begin...rescue...end`", którym jest `ensure`. Blok `ensure` wykonuje się niezależnie od pomyślnego lub niepomyślnego zakończenia bloku `begin`.

```
begin
  plik = open("/tmp/jakis_plik", "w")
  # ... zapis do pliku ...
rescue
  # ... obsluga wyjatkow ...
ensure
  plik.close   # ... zawsze wykonywane.
end
```

Możliwe jest używanie `ensure` bez `rescue` i vice versa, ale jeśli używane są razem w tym samym bloku `begin...end`, `rescure` musi poprzedzać `ensure`.

Rozdział 29

Akcesory

Akcesory

Czym jest akcesor?

Krótko omówiliśmy zmienne instancji we wcześniejszym rozdziale, ale jeszcze za wiele z nimi nie robiliśmy. Zmienne instancji obiektu są jego atrybutami, to te rzeczy, które odróżniają obiekt od innych obiektów tej samej klasy. Za ważną czynność zapisywania i odczytywania atrybutów odpowiedzialne są metody nazywane *akcesorami atrybutów*. Jak za chwilę zobaczymy, nie musimy pisać akcesorów bezpośrednio. Wcześniej jednak poznajmy wszystkie etapy ich tworzenia.

Wyróżniamy dwa rodzaje akcesorów: *piszące* (ang. *writer*[1]) i *czytające* (ang. *reader*[2]).

```
class Owoc
  def zapisz_rodzaj(r)    # akcesor piszacy
    @rodzaj = r
  end

  def czytaj_rodzaj       # akcesor czytajacy
    @rodzaj
  end
end

o1 = Owoc.new

# uzycie akcesora piszacego:
o1.zapisz_rodzaj("brzoskwinia")

# uzycie akcesora czytajacego:
o1.czytaj_rodzaj #=> "brzoskwinia"
```

Gdybyśmy wpisali powyższy kod do *irba*, moglibyśmy zastosować następujący sposób inspekcji:

[1] W innych językach programowania powszechnie funkcjonuje nazwa *setter*.
[2] W innych językach programowania — *getter*.

```
irb(main):011:0> o1
=> #<Owoc:0x2e47044 @rodzaj="brzoskwinia">
```

Proste prawda? Możemy przechowywać i odczytywać informację o tym, na jaki owoc patrzymy. Ale nasze nazwy metod są nieco rozwlekłe. W następującym przykładzie są już nieco bardziej zwięzłe i konwencjonalne:

```
class Owoc
  def rodzaj=(r)
    @rodzaj = r
  end

  def rodzaj
    @rodzaj
  end
end

o2 = Owoc.new
o2.rodzaj = "banan"
o2.rodzaj #=> "banan"
```

Metoda `inspect`

Jest tu potrzebna krótka dygresja. Z pewnością zauważyłeś, że jeżeli próbujemy spojrzeć na obiekt w *irbie* bezpośrednio, pokazuje się nam coś zagadkowego w rodzaju `#<jakisObiekt:0x83678>`. To jest po prostu domyślne zachowanie i oczywiście możemy je zmienić. Wszystko, co musimy zrobić to dodać metodę o nazwie `inspect`. Powinna ona zwracać łańcuch, który opisuje obiekt w jakiś sensowny sposób, włączając stany części lub wszystkich jego zmiennych instancji.

```
class Owoc
  def inspect
    "owoc rodzaju: #{@rodzaj}"
  end
end
```

Podobną metodą jest metoda `to_s` (ang. *convert to string* — zamień na łańcuch), która jest używana gdy drukujemy obiekt. Ogólnie rzecz biorąc, możesz traktować `inspect` jako narzędzie, gdy piszesz i debugujesz programy, natomiast `to_s` jako sposób na formowanie wyjścia programu. *irb* używa `inspect` ilekroć wyświetla wyniki. Możesz użyć metody `p` by łatwo debugować wyjście z programów.

```
# Te dwie linie są równoważne:
p obiekt
puts obiekt.inspect
```

Łatwy sposób tworzenia akcesorów

Ponieważ wiele zmiennych instancji potrzebuje akcesorów, Ruby dostarcza wygodne skróty dla standardowych form.

Skrót	Efekt
`attr_reader :v`	`def v; @v; end`
`attr_writer :v`	`def v=(value); @v=value; end`
`attr_accessor :v`	`attr_reader :v; attr_writer :v`
`attr_accessor :v, :w`	`attr_accessor :v; attr_accessor :w`

Korzystając z tabeli uporządkujmy naszą klasę i dodajmy informację na temat świeżości. Najpierw automatycznie wygenerujemy akcesory: czytające i piszące, a następnie dodamy nową informację do metody `inspect`:

```
class Owoc
  attr_accessor :stan
  attr_accessor :gatunek

  def inspect
    "#{@stan} owoc rodzaju: #{@gatunek}"
  end
end

o = Owoc.new
o.gatunek = "banan"
o.stan = "dojrzaly"
p o #=> dojrzaly owoc rodzaju: banan
```

Więcej zabawy z owocem

Jeżeli nikt nie zjadł naszego dojrzałego owocu, może należałoby pozwolić, by czas zebrał swoje żniwo (poniższy kod możemy dopisać bezpośrednio do powyższego):

```
class Owoc
  def uplywa_czas
    @stan = "gnijacy"
  end
end

p o #=> dojrzaly owoc rodzaju: banan
o.uplywa_czas
p o #=> gnijacy owoc rodzaju: banan
```

W następnym rozdziale zobaczymy, jak zapewnić, by już w momencie utworzenia `Owoc` miał zdefiniowany rodzaj i stan.

Rozdział 30

Inicjalizacja obiektów

Inicjalizacja obiektów

Nasza klasa `Owoc` z poprzedniego rozdziału ma dwie zmienne instancji, jedną by opisywać rodzaj owocu, a drugą by opisywać jego kondycję. Ruby dostarcza eleganckiego sposóbu na zapewnienie, że zmienne instancji będą zawsze zainicjalizowane.

Metoda `initialize`

Ilekroć Ruby tworzy nowy obiekt, szuka metody nazwanej `initialize` i wykonuje ją. Tak więc najprostszą rzeczą, którą możemy zrobić jest użycie metody `initialize`, by przypisać domyślne wartości do wszystkich zmiennych instancji.

```
class Owoc
  def initialize
    @rodzaj = "jablko"
    @stan = "dojrzaly"
  end
end

o = Owoc.new
p o #=> "dojrzaly owoc rodzaju: jablko"
```

Zmiana założeń w wymagania

Będą takie przypadki, że domyślna wartość nie będzie miała wielkiego sensu. Czy jest w ogóle coś takiego jak domyślny rodzaj owocu? Bardziej pożądanym może być wymaganie, by każdy kawałek owocu posiadał swój własny rodzaj określony podczas tworzenia. By zrobić to, możemy dodać formalny argument do metody `initialize`. Z powodów, w które nie będziemy się teraz zagłębiać, argumenty które przekazujesz do metody `new` są dostarczane do `initialize`.

```
class Owoc
  def initialize(r)
    @rodzaj = r
    @stan = "dojrzaly"
```

```
  end
end

o = Owoc.new("mango")
p o #=> "dojrzaly owoc rodzaju: mango"
o2 = Owoc.new #=> ERR: (eval):1:in `initialize': wrong # of arguments(0 for 1)
```

Elastyczna inicjalizacja

Jak wyżej widzimy, gdy tylko argument dołączony jest do metody `initialize`, nie można go opuścić bez generowania błędu. Jeśli chcemy nieco bardziej dbać o użytkownika, możemy argumentom nadać od razu wartości domyślne.

```
class Owoc
  def initialize(r = "jablko")
    @rodzaj = r
    @stan = "dojrzaly"
  end
end

o = Owoc.new("mango")
p o  #=> "dojrzaly owoc rodzaju: mango"

o2 = Owoc.new
p o2 #=> "dojrzaly owoc rodzaju: jablko"
```

Wartości domyślnych możesz używać dla każdej metody, nie tylko `initialize`. Lista argumentów musi być tak ustawiona, by argumenty z domyślnymi wartościami były podawane jako ostanie.

Rozdział 31

Komentarze i organizacja kodu

Komentarze i organizacja kodu

Ten rozdział zawiera kilka praktycznych porad, przydatnych w dalszym poznawaniu Rubiego.

Separatory instrukcji

Niektóre języki wymagają pewnego rodzaju znaków przestankowych, często średników (;), by zakończyć każdą instrukcję w programie. Zamiast tego Ruby podąża za konwencją używaną w powłokach systemowych takich jak *sh* i *csh*. Wiele instrukcji w jednej linii musi być odseparowanych średnikami, ale nie są one wymagane na końcu linii. Znak końca wiersza traktowany jest jako przecinek. Jeżeli linia kończy się znakiem odwróconego ukośnika (\), znak końca linii jest ignorowany, co pozwala na utworzenie jednej linii logicznej podzielonej na kilka części.

Komentarze

Po co pisać komentarze? Chociaż dobrze napisany kod ma tendencję do dokumentowania samego siebie, skrobanie po marginesach jest często pomocne. Błędem jest sądzić, że inni będą w stanie spojrzeć na nasz kod i natychmiast zobaczyć go w ten sposób co ty. Poza tym, praktycznie rzecz biorąc, ty też jesteś inną osobą co parę dni... Któż z nas nie wrócił kiedyś do kodu programu, by go poprawić lub rozszerzyć i nie powiedział: *wiem, że to napisałem, ale co to u licha znaczy?*

Niektórzy doświadczeni programiści wskażą, całkiem słusznie, że sprzeczne lub przestarzałe komentarze mogą być gorsze niż żadne. Z pewnością komentarze nie powinny być substytutem czytelnego kodu. Jeżeli twój kod jest nieczytelny, prawdopodobnie zawiera również błędy. Możesz odkryć, że potrzebujesz więcej komentować, gdy uczysz się Rubiego, a mniej, jak staniesz się lepszy w wyrażaniu pomysłów w prostym, eleganckim, czytelnym kodzie.

Ruby podąża za ogólną konwencją skryptów, która używa symbolu # by oznaczać początek komentarza. Wszystko co występuje za znakiem # do końca linii jest ignorowane przez interpreter.

Również, by ułatwić wprowadzanie dużych bloków komentarzy, interpreter Rubiego ignoruje wszystko pomiędzy linią zaczynającą się od "`=begin`" a drugą linią zaczynającą się od "`=end`".

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

=begin
************************************************************************
  To jest komentarz blokowy. Cos co piszesz dla innych czytelnikow,
  i rowniez dla siebie. Interpreter zignoruje ten tekst. Nie trzeba
  w nim uzywac '#' na poczatku kazdej linii.
************************************************************************
=end
```

Organizacja kodu

Niezwykle wysoki poziom dynamizmu Rubiego oznacza, że klasy, moduły oraz metody istnieją tylko, gdy definiujący je kod zostanie uruchomiony. Jeżeli nawykłeś do programowania w bardziej statycznym języku, może to czasami prowadzić do niespodzianek.

```
# Ponizszy kod zwraca blad: "undefined method":

puts nastepca(3)

def nastepca(x)
  x + 1
end
```

Chociaż interpreter sprawdza, czy w skrypcie nie występują błędy składniowe przed wykonaniem go, kod "`def nastepca ... end`" musi być zinterpretowany w celu utworzenia metody `nastepca`. Tak więc kolejność kodu w skrypcie może mieć istotne znaczenie.

Nie zmusza to, jak mogło by się wydawać, do organizacji ściśle według stylu "od dołu do góry". Kiedy interpreter napotka definicję metody, może ona bezpiecznie zawierać niezdefiniowane referencje, dopóki będziesz pewny, że będą one zdefiniowane nim metoda zostanie wywołana.

```
# Konwersja stopni ze skali Fahrenheita do Celsiusza, podzielona
# na dwie czesci.

def f_na_c(f)
  skala(f - 32.0)  # To jest referencja do niezdefiniowanej jeszcze metody, ale to nie s
end

def skala(x)
  x * 5.0 / 9.0
end

printf "%.1f jest komfortową temperaturą.\n", f_na_c(72.3)
```

To podejście może wydawać się to mniej wygodne niż to, które możesz znać z Perla czy Javy, ale jest mniej restrykcyjne niż próbowanie pisania w C bez prototypów (co zawsze zmuszałoby cię do częściowego zarządzania kolejnością, co wskazuje na co). Umieszczanie kodu na najwyższym poziomie zagnieżdżenia, na końcu pliku źródłowego zawsze działa. I sprawia mniej kłopotu, niż można sądzić. Sensownym i bezbolesnym sposobem, by wymusić określone zachowanie, jest zdefiniowanie funkcji `main` na górze pliku i wywołanie jej z dołu.

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

def main
  # tutaj logika nawyższego poziomu
end

# ... dodatkowy kod ...

main # ... start main.
```

Pewna pomoc płynie również z faktu, że Ruby dostarcza narzędzi do dzielenia skomplikowanych programów w czytelne, nadające się do powtórnego użycia, logicznie powiązane klocki. Widzieliśmy już użycie słowa `include` by uzyskać dostęp do modułów. Również mechanizmy `load` i `require` są bardzo użyteczne. Instrukcja `load` działa tak, jakby plik do którego się ona odnosi został skopiowany i wklejony (coś jak dyrektywa preprocesora `#include` w C). Instrukcja `require` jest nieco bardziej zaawansowana, powodując, że kod będzie załadowany tylko raz i tylko wtedy, gdy będzie to konieczne.

Dodatek A

Informacje o pliku i historia

Historia

Ta książka została stworzona na polskojęzycznej wersji projektu Wikibooks przez autorów wymienionych poniżej w sekcji Autorzy. Dla wygody czytelników stworzony został niniejszy plik PDF. Najnowsza wersja podręcznika jest dostępna pod adresem `http://pl.wikibooks.org/wiki/Ruby`.

Informacje o pliku PDF i historia

PDF został utworzony przez Derbetha dnia 17 lutego 2008 na podstawie wersji z 17 lutego 2008 podręcznika na Wikibooks. Wykorzystany został poprawiony program Wiki2LaTeX autorstwa użytkownika angielskich Wikibooks, Hagindaza. Wynikowy kod po ręcznych poprawkach został przekształcony w książkę za pomocą systemu składu LaTeX.

Najnowsza wersja tego PDF-u jest postępna pod adresem `http://pl.wikibooks.org/wiki/Image:Ruby.pdf`.

Autorzy

Ajsmen91, Derbeth, Fservant, Kj, Koko, Maniel, Piotr, Sblive, Szymon wro i anonimowi autorzy.

Grafiki

Autorzy i licencje grafik:

- grafika na okładce: autor Yukihiro Matsumoto, Ruby Visual Identity Team, źródło Wikimedia Commons, licencja Creative Commons Attribution ShareAlike 2.5
- logo Wikibooks: zastrzeżony znak towarowy, © & ™All rights reserved, Wikimedia Foundation, Inc.

Dodatek B

Dalsze wykorzystanie tej książki

Wstęp

Ideą Wikibooks jest swobodne dzielenie się wiedzą, dlatego też uregulowania Wikibooks mają na celu jak najmniejsze ograniczanie możliwości osób korzystających z serwisu i zapewnienie, że treści dostępne tutaj będą mogły być łatwo kopiowane i wykorzystywane dalej. Prosimy wszystkich odwiedzających, aby poświęcili chwilę na zaznajomienie się z poniższymi zasadami, by uniknąć w przyszłości nieporozumień.

Status prawny

Cała zawartość Wikibooks (o ile w danym miejscu nie jest zaznaczone inaczej; szczegóły niżej) jest udostępniona na następujących warunkach:

Udziela się zezwolenia na kopiowanie, rozpowszechnianie i/lub modyfikację treści artykułów polskich Wikibooks zgodnie z zasadami Licencji GNU Wolnej Dokumentacji (GNU Free Documentation License) w wersji 1.2 lub dowolnej późniejszej opublikowanej przez Free Software Foundation; bez Sekcji Niezmiennych, Tekstu na Przedniej Okładce i bez Tekstu na Tylnej Okładce. Kopia tekstu licencji znajduje się na stronie GNU Free Documentation License.

Zasadniczo oznacza to, że artykuły pozostaną na zawsze dostępne na zasadach open source i mogą być używane przez każdego z obwarowaniami wyszczególnionymi poniżej, które zapewniają, że artykuły te pozostaną wolne.

Grafiki i pliku multimedialne wykorzystane na Wikibooks mogą być udostępnione na warunkach innych niż GNU FDL. Aby sprawdzić warunki korzystania z grafiki, należy przejść do jej strony opisu, klikając na grafice.

Wykorzystywanie materiałów z Wikibooks

Jeśli użytkownik polskich Wikibooks chce wykorzystać materiały w niej zawarte, musi to zrobić zgodnie z GNU FDL. Warunki te to w skrócie i uproszczeniu:

Publikacja w Internecie

1. dobrze jest wymienić Wikibooks jako źródło (jest to nieobowiązkowe, lecz jest dobrym zwyczajem)
2. należy podać listę autorów lub wyraźnie opisany i funkcjonujący link do oryginalnej treści na Wikibooks lub historii edycji (wypełnia to obowiązek podania autorów oryginalnego dzieła)
3. trzeba jasno napisać, że treść publikowanego dzieła jest objęta licencją GNU FDL
4. należy podać link to tekstu licencji (najlepiej zachowanej na własnym serwerze)
5. postać tekstu nie może ograniczać możliwości jego kopiowania

Druk

1. dobrze jest wymienić Wikibooks jako źródło (jest to nieobowiązkowe, lecz jest dobrym zwyczajem)
2. należy wymienić 5 autorów z listy w rozdziale *Autorzy* danego podręcznika (w przypadku braku podobnego rozdziału — lista autorów jest dostępna pod odnośnikiem "historia" na górze strony). Gdy podręcznik ma mniej niż 5 autorów, należy wymienić wszystkich.
3. trzeba jasno napisać na przynajmniej jednej stronie, że treść publikowanego dzieła jest objęta licencją GNU FDL
4. *pełny* tekst licencji, w oryginale i bez zmian, musi być zawarty w książce
5. jeśli zostanie wykonanych więcej niz 100 kopii książki konieczne jest:
   (a) dostarczenie płyt CD, DVD, dysków lub innych nośników danych z treścia książki w formie możliwą do komputerowego przetwarzania; lub:
   (b) dostarczenie linku do strony z czytelną dla komputera formą książki (link musi być aktywny przynajmniej przez rok od publikacji; można użyć linku do spisu treści danego podręcznika na Wikibooks)

Nie ma wymogu pytania o zgodę na wykorzystanie tekstu jakichkolwiek osób z Wikibooks. Autorzy nie mogą zabronić nikomu wykorzystywania ich tekstów zgodnie z licencją GNU FDL. Można korzystać z książek jako całości albo z ich fragmentów. Materiały bazujące na treści z Wikibooks mogą być bez przeszkód sprzedawane; zyskami nie trzeba dzielić się z autorami oryginalnego dzieła.

Jeżeli w wykorzystanych materiałach z polskich Wikibooks są treści objęte innymi niż GNU FDL licencjami, użytkownik musi spełnić wymagania zawarte w tych licencjach (dotyczy to szczególnie grafik, które mogą mieć różne licencje).

Dodatek C

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and

is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or

"**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under

this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.