

» Idź do

- Spis treści
- Przykładowy rozdział

» Katalog książek

- Katalog online
- Zamów drukowany katalog

» Twój koszyk

- Dodaj do koszyka

» Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

» Czytelnia

- Fragmenty książek online

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

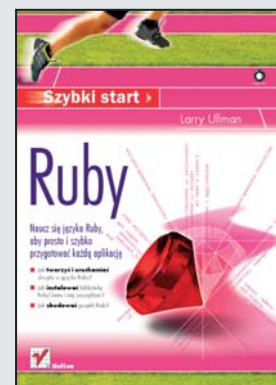
Ruby. Szybki start

Autor: Larry Ullman

ISBN: 978-83-246-2258-0

Tytuł oryginału: [Ruby: Visual QuickStart Guide](#)

Format: 170x230, stron: 448



Naucz się Języka Ruby, aby prosto i szybko przygotować każdą aplikację

- Jak tworzyć i uruchamiać skrypty w języku Ruby?
- Jak instalować bibliotekę RubyGems i zarządzać nią?
- Jak zbudować projekt Rails?

Ruby to dynamiczny i niezwykle elastyczny język programowania. Dzięki prostemu kodowi jest on także przystępny i łatwy w nauce. Pozwala na zmianę elementów języka podczas pracy programu. Co więcej – na najwyższym stopniu zawansowania aplikacje napisane w tym języku mogą wykorzystywać refleksję, czyli zdolność do samoanalizy. Biblioteka RubyGems zawiera niestandardowy, ale bardzo rozbudowany kod, a poza tym udostępnia dodatkowe narzędzia, co znacznie przyspiesza prace nad tworzeniem dowolnej aplikacji.

Książka „Ruby. Szybki start” zawiera wszystkie potrzebne wiadomości, podane tak, abyś szybko i sprawnie nauczył się tego języka – bez obciążania pamięci zbędnymi szczegółami czy niezrozumiałym technicznym żargonem. Zamieszczone tu instrukcje, z dodatkowymi objaśnieniami graficznymi, krok po kroku pokazują na przykład kod, jaki należy wpisać z klawiatury. Z podręcznikiem w ręku nauczysz się m.in. używać wątków, konfigurować bazę danych, instalować bibliotekę RubyGems i zarządzać nią. Reasumując – z tą książką możesz od razu zacząć pracę i korzystać z możliwości języka Ruby do realizacji wielu zadań programistycznych.

- Dokumentacja języka Ruby
- Uruchamianie skryptów i pobieranie danych
- Tablice, zakresy i hasze
- Struktury sterujące
- Tworzenie metod
- Klasy i dziedziczenie
- Moduły
- Wyrażenia regularne
- Debugowanie i obsługa błędów
- Katalogi i pliki
- Bazy danych
- Ruby on Rails
- Programowanie dynamiczne

Szybki start w świat języka Ruby!

Spis treści

	Wstęp	9
Rozdział 1.	Zaczynamy	17
	Instalacja w systemie Windows	18
	Instalacja w systemie Mac OS X	20
	Testowanie instalacji	23
	Dokumentacja języka Ruby	25
	Interaktywna powłoka języka Ruby	28
	Konfiguracja powłoki irb	31
Rozdział 2.	Proste skrypty	35
	Tworzenie najprostszego skryptu	36
	Uruchamianie skryptów w Windows	38
	Uruchamianie skryptów z wiersza poleceń	40
	Skrypty wykonywalne	42
	Wyświetlanie tekstu	44
	Pobieranie danych	46
	Komentarze	48
Rozdział 3.	Typy podstawowe	51
	Liczby	52
	Obliczenia arytmetyczne	54
	Metody operujące na liczbach	56
	Łańcuchy znaków	59
	Interpolacja i zastępowanie	61
	Podstawowe metody operujące na łańcuchach znaków	64
	Łańcuchy znakowe wielowierszowe	66
	Stałe	69
	Data i czas	71
Rozdział 4.	Tablice, zakresy i hasze	75
	Tworzenie tablic	76
	Podstawowe metody klasy Array	79
	Dodawanie elementów	83

	Usuwanie elementów	86
	Tablice i łańcuchy znaków	88
	Używanie zakresów	90
	Tworzenie haszów	93
	Standardowe metody klasy Hash	95
Rozdział 5.	Struktury sterujące	97
	Operatory	98
	Podstawowe instrukcje warunkowe	101
	Instrukcje warunkowe złożone	105
	Operator warunkowy	108
	Instrukcja case	112
	Pętle	116
	Iteratory liczbowe	120
	Iteratory kolekcji	123
Rozdział 6.	Tworzenie metod	127
	Proste metody	128
	Zwracanie wartości	131
	Pobieranie argumentów	135
	Domyślne wartości argumentów	138
	Używanie self	140
	Argumenty o zmiennej długości	143
	Metody i bloki	147
Rozdział 7.	Klasy	151
	Proste klasy	152
	Zmienne instancji	156
	Akcesory	159
	Konstruktory	162
	Definiowanie operatorów	166
	Metody specjalne	175
	Walidacja i duck typing	180
Rozdział 8.	Dziedziczenie i cała reszta	185
	Proste dziedziczenie	186
	Nadpisywanie metod	191
	Metody łączone	194
	Kontrola dostępu	198
	Zmienne klasy	204
	Metody klasy	207

Rozdział 9.	Moduły	213
	Moduły jako przestrzenie nazw	214
	Moduły jako klasy mieszane	218
	Dołączanie plików	222
	Standardowa biblioteka języka Ruby	227
Rozdział 10.	Wyrażenia regularne	229
	Przeprowadzanie dopasowań	230
	Definiowanie prostych wzorców	233
	Używanie kotwic	236
	Używanie kwantyfikatorów	239
	Używanie klas znaków	242
	Używanie modyfikatorów	246
	Wyszukiwanie dopasowań	248
	Przeprowadzanie podstawień	252
Rozdział 11.	Debuggowanie i obsługa błędów	257
	Używanie debuggera Ruby	258
	Obsługa wyjątków	264
	Obsługa wyjątku w zależności od jego typu	268
	Zgłaszanie wyjątków	271
	Testowanie jednostkowe (Unit Testing)	276
Rozdział 12.	RubyGems	283
	Instalacja RubyGems	284
	Instalacja i zarządzanie bibliotekami	287
	Korzystanie z pakietów	291
	Pakiet creditcard	294
	Pakiet Highline	296
	Pakiet RedCloth	302
Rozdział 13.	Katalogi i pliki	305
	Podstawy	306
	Dostęp do zawartości katalogu	309
	Właściwości katalogów i plików	312
	Uprawnienia	315
	Tworzenie, przenoszenie, kopiowanie i usuwanie	319
	Odczytywanie danych z plików	322
	Zapisywanie danych w plikach	325
	FasterCSV	329

Rozdział 14.	Bazy danych	333
	Zaczynamy	334
	Wykonywanie prostych zapytań	337
	Wstawianie rekordów	339
	Pobieranie rekordów	344
	Wykonywanie transakcji	348
Rozdział 15.	Sieć	353
	Tworzenie serwera gniazd	354
	Używanie wątków	357
	Tworzenie klienta gniazd	362
	Połączenia HTTP	365
	Obsługa źródeł RSS	369
Rozdział 16.	Ruby on Rails	373
	Elementarz Rails	374
	Zaczynamy	376
	Konfiguracja bazy danych	381
	Tworzenie bazy danych	384
	Wypróbowywanie serwisu	389
	Dostosowywanie modeli	391
	Dostosowywanie widoków	395
	Dostosowywanie kontrolerów	403
Rozdział 17.	Programowanie dynamiczne	409
	Integracja z systemem operacyjnym	410
	Skazone dane	414
	Poziomy bezpieczeństwa	418
	Elementy proc i lambda	422
	Skorowidz	429

Klasy są fundamentem programowania zorientowanego obiektowo (OOP). **Klasa** jest wzorcem, **obiekt** jest **instancją** klasy. Ponieważ w języku Ruby wszystko jest obiektem, nawet prosty przykład *Witaj, Świecie!* korzysta z obiektów. Mimo że Ruby dostarcza mnóstwo wbudowanych funkcji poprzez wbudowane klasy, w końcu będziesz chciał tworzyć własne.

Rozdział ten przedstawia podstawy tworzenia i używania klas w języku Ruby. Znajdziesz w nim wiadomości o składni klas, zmiennych klas oraz jak pisać kilka różnych rodzajów metod, które są wykorzystywane w klasach. Wprowadzone zostaną również dwa pojęcia, które odróżniają Ruby od wielu innych języków zorientowanych obiektowo. Pierwszym jest łatwość, z jaką możesz zmieniać istniejące klasy. Drugim jest *duck typing*. Zostanie ono opisane na końcu rozdziału.

Rozdział 8., „Dziedziczenie i cała reszta”, jest z niniejszym powiązany. Główna uwaga skupia się w nim na definiowaniu klas, które wywodzą się z innych klas.

Proste klasy

Programowanie proceduralne podchodzi do aplikacji jako do **kroków**, które należy wykonać. Przeciwnie do niego programowanie obiektowe skupia się na **danych** przetwarzanych w aplikacji. Stosując takie podejście, klasa musi reprezentować dane aplikacji w sensie danych, które są przechowywane, jak i zadań do wykonania. Informacja może być reprezentowana przez zmienne (nazywane **atributami** lub **właściwościami** klasy) oraz funkcje (w klasach nazywanymi **metodami**). Jako teoretyczny przykład weźmy osobę: posiada ona atrybuty — imię, wiek itp. — oraz metody — je, śpi itd.

Podstawową definicję klasy rozpoczyna słowo kluczowe `class`, po czym następuje nazwa klasy. Następnie jest jej kod (ciało klasy), definicję klasy kończy słowo kluczowe `end`.

```
class NazwaKlasy
  # kod klasy
end
```

Nazwa klasy może zawierać litery, cyfry i znaki podkreślenia, zazwyczaj jednak zawiera tylko litery. Nazwy klas traktowane są jako stałe, więc muszą rozpoczynać się od wielkiej litery. Przyjmuje się, że używają wielkich liter do oddzielania wyrazów w nazwach klas: *NazwaKlasy*, nie *nazwaKlasy* czy *Nazwa_klasy*.

Metody wewnątrz klasy definiuje się tak samo, jak to zostało opisane w rozdziale 6., „Tworzenie metod”. Definicje metod zazwyczaj są wcięte o dwie spacje, lecz składnia tego nie wymaga.

Na przykład tak bardzo kochasz przykład *Witaj, Świecie!*, że uważasz, iż zasługuje na własną klasę:

```
class HelloWorld
  def say_hello
    puts "Witaj, Świecie!"
  end
end
```

Po utworzeniu klasy możesz stworzyć zmienną jej typu, używając *NazwaKlasy.new*. Wiersz ten korzysta z wywołania metody *new* danej klasy, każda klasa posiada taką metodę automatycznie, bez konieczności definiowania jej przez Ciebie.

```
hw = HelloWorld.new
```

Klasa Struct

Jeśli potrzebujesz przechować wiele informacji w strukturze podobnej do klasy, ale nie musisz definiować żadnych metod, możesz użyć klasy `Struct`. Aby ją zdefiniować, wpisz `Struct.new`, jako parametry podając nazwę struktury oraz zmiennych (jako symbole), jakie powinna zawierać:

```
Book = Struct.new('Book', :title, :author, :isbn)
```

Teraz możesz utworzyć zmienną tego typu:

```
rubyvqs = Book.new('Ruby: VQS', 'Larry Ullman', '978-0-321-55385-0')
```

Od tego momentu możesz używać składni z kropką do dostępu do danych:

```
puts rubyvqs.title # Ruby: VQS
```

Alternatywnie możesz traktować tę zmienną jak hasz:

```
puts rubyvqs[:title] # Ruby: VQS
```

```

C:\ Wiersz polecenia
>> class HelloWorld
>>   def say_hello
>>     puts "Witaj, Świecie!"
>>   end
>> end
=> nil
>> hw = HelloWorld.new
=> #<HelloWorld:0x2939b88>
>> hw.say_hello
Witaj, Świecie!
=> nil
>>

```

Rysunek 7.1. Po utworzeniu nowej klasy Ruby zwraca nil. Kiedy utworzony zostanie nowy obiekt klasy, Ruby zwraca nazwę typu oraz referencję do obiektu (0x2939b88). Metody klasy mogą być wywoływane na jej obiektach

Użyj składni z kropką do wywołania metod klasy (rysunek 7.1):

```
hw.say_hello
```

Ruby posiada wiele metod, które mogą być użyte przez każdą klasę. Podobnie jak metoda new, te inne metody są automatycznie **dziedziczone** przez klasy, które utworzyłeś (rozdział 8., „Dziedziczenie i cała reszta”, zajmuje się szczegółowo dziedziczeniem). Na przykład metoda class zwraca nazwę klasy, do której należy dany obiekt:

```
hw.class # HelloWorld
```

Aby wyświetlić listę metod, jakie posiada klasa, użyj `NazwaKlasy.methods`. Aby wyświetlić listę metod, jakie posiada dany obiekt, wpisz `nazwa_obiektu.methods` (rysunek 7.2) lub `NazwaKlasy.instance_methods`.

W bieżącym rozdziale utworzysz wiele różnych klas — niektóre w celu lepszego zrozumienia zasad programowania obiektowego, inne będą bardziej użyteczne. Dla uproszczenia początkowe przykłady będą napisane przy użyciu powłoki *irb*, późniejsze przykłady będą skryptami w osobnych plikach. Możesz napisać skrypty dla każdego prezentowanego przykładu, jeśli oczywiście chcesz (tak jak opisano w rozdziale 2., „Proste skrypty”).

```

C:\ Wiersz polecenia
>> HelloWorld.methods
=> ["inspect", "instance_method", "private_class_method", "const_missing", "clone", "public_methods", "public_instance_methods", "display", "instance_variable_defined?", "method_defined?", "superclass", "equal?", "freeze", "included_modules", "const_get", "methods", "respond_to?", "module_eval", "class_variables", "autoload?", "dup", "protected_instance_methods", "instance_variables", "public_method_defined?", "___id___", "method", "eql?", "const_set", "id", "singleton_methods", "send", "class_eval", "taint", "frozen?", "instance_variable_get", "include?", "private_instance_methods", "___send___", "instance_of?", "private_method_defined?", "to_a", "name", "type", "new", "<=", "protected_methods", "instance_eval", "object_id", "<=>", "=="", "instance_variable_set", "kind_of?", "extend", "protected_method_defined?", "const_defined?", ">=", "ancestors", "to_s", "<=", "public_class_method", "allocate", "hash", "class", "instance_methods", "tainted?", "!="", "private_methods", "class_variable_defined?", "autoload", "nil?", "untaint", "constants", "is_a?"]
>> hw.methods
=> ["say_hello", "instance_variables", "___id___", "to_s", "send", "display", "dup", "private_methods", "!="", "is_a?", "class", "tainted?", "singleton_methods", "eql?", "method", "untaint", "instance_of?", "id", "instance_variable_get", "inspect", "instance_eval", "extend", "nil?", "object_id", "___send___", "frozen?", "taint", "instance_variable_defined?", "public_methods", "hash", "to_a", "clone", "protected_methods", "respond_to?", "freeze", "kind_of?", "=="", "instance_variable_set", "type", "=="", "equal?", "methods"]
>>

```

Rysunek 7.2. Pierwsza część wyświetla 75 metod, które mogą być użyte z klasą HelloWorld (w której zdefiniowano tylko jedną metodę). Druga część wyświetla 42 metody, które mogą być wywołane na obiekcie HelloWorld

Aby utworzyć klasę:**1. Rozpocznij definiowanie klasy:**

```
class Dog
```

Klasa nazywa się Dog (pies) i posłuży do uproszczonego reprezentowania, eee, psa w kodzie języka Ruby.

2. Dodaj kilka metod:

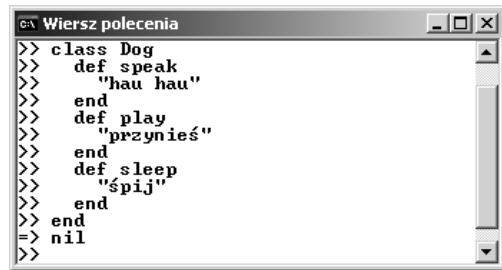
```
  def speak
    "hau"
  end
  def play
    "przynieś"
  end
  def sleep
    "śpij"
  end
```

Te trzy proste metody reprezentują czynności, które może wykonywać pies. Każda z nich zwraca pojedynczy wyraz jako ciąg znaków. Więcej o definiowaniu metod można przeczytać w rozdziale 6., „Tworzenie metod”.

3. Zakończ definiowanie klasy (rysunek 7.3):

```
end
```

Gdybym pisał ten kod jako skrypt Ruby, dołączyłbym tutaj komentarz, aby zaznaczyć, że ten end kończy definicję klasy (w przeciwieństwie do końca definicji metody).



```

C:\> Wiersz polecenia
>> class Dog
>>   def speak
>>     "hau hau"
>>   end
>>   def play
>>     "przynieś"
>>   end
>>   def sleep
>>     "śpij"
>>   end
>> end
=> nil

```

Rysunek 7.3. Definicja klasy Dog

4. Użyj klasy (rysunek 7.4):

```
trixie = Dog.new
trixie.speak
trixie.play
trixie.sleep
```

Pierwszy wiersz tworzy nowy obiekt typu Dog. Obiekt jest zmienną o nazwie `trixie`. Jego metody mogą być wywołane za pomocą składni z kropką. Ponieważ każda z metod zwraca ciąg znaków, wynikiem wywołania każdej z nich jest po prostu ciąg znaków. Jako alternatywę możesz wypisać wynik wywołania każdej metody:

```
puts trixie.speak
```

Wskazówki

- Metody definiowane na zewnątrz klasy, jak te z rozdziału 6, „Tworzenie metod”, automatycznie są definiowane jako metody klasy `Object`. Ponieważ każda klasa automatycznie dziedziczy po klasie `Object`, metody te są dziedziczone przez każdą klasę.
- Metody zdefiniowane wewnątrz klasy, takie jak `speak`, `play` czy `sleep`, nazywane są **metodami instancji** (ang. *instance methods*).



```
scoreo@tinki: ~
Plik Edycja Widok Terminal Karty Pomog
irb(main):020:0> trixie = Dog.new
=> #<Dog:0x8da395c>
irb(main):021:0> trixie.speak
=> "hau hau"
irb(main):022:0> trixie.play
=> "przynieś"
irb(main):023:0> trixie.sleep
=> "śpi"
irb(main):024:0>
```

Rysunek 7.4. Utworzenie obiektu klasy `Dog` i wywołanie kilku jego metod

Zmienne instancji

Zmienne definiowane poza klasą są zmiennymi **lokalnymi**, tak jak te definiowane i używane w poprzednich rozdziałach. Zmienne definiowane wewnątrz klasy to **zmienne instancji** (ang. *instance variables*). Należą one do klasy, dlatego należą też do utworzonych obiektów danej klasy. Zmienne instancji podlegają tym samym regułom nazewnictwa co inne zmienne, wyjątkiem jest to, że ich nazwa zaczyna się od znaku @.

W klasie Dog zmienne instancji mogą przechowywać imię, wiek, rasę, płeć itd. poszczególnego psa. W klasach często tworzone są metody służące do **przypisywania** wartości danej zmiennej instancji (ang. *setter method*) lub **pobierania** jej zawartości (ang. *getter method*). W tym celu można w klasie Dog utworzyć takie dwie metody:

```
def set_name(n)
  @name = n
end
def get_name
  @name
end
```

Taka wersja klasy będzie używana w następujący sposób (rysunek 7.5):

```
trixie = Dog.new
trixie.set_name('trixie')
puts "Mój pies wabi się #{trixie.get_name}."
```

Oczywiście każda metoda klasy może korzystać ze zmiennej instancji:

```
def sleep
  "#@name śpi"
end
```

W metodzie sleep pokazano nowy przykład **interpolacji** (wstawienie wewnątrz ciągu znaków wartości zwróconej w wyniku polecenia). Aby wstawić zwykłe zmienne, musisz użyć `#{nazwa_zmiennej}`, ale by wstawić wartość zmiennej instancji, możesz użyć `#@nazwa_zmiennej`.



```

c:\ Wiersz polecenia
>> class Dog
>>   def speak
>>     "hau hau"
>>   end
>>   def play
>>     "przynieś"
>>   end
>>   def sleep
>>     "śpij"
>>   end
>>   def set_name(n)
>>     @name = n
>>   end
>>   def get_name
>>     @name
>>   end
>> end
=> nil
>> trixie = Dog.new
=> #<Dog:0x291d3ac>
>> trixie.set_name('trixie')
=> "trixie"
>> puts "Mój pies wabi się #{trixie.get_name}."
Mój pies wabi się trixie.
=> nil
>>
```

Rysunek 7.5. Jedna metoda setter i jedna metoda getter zostały dodane do klasy Dog, by mieć dostęp do zmiennej @name

```

C:\> Wiersz polecenia - irb
>> class Circle
>>   def set_radius(r)
>>     @radius = r
>>   end
>>   def get_radius
>>     @radius
>>   end
>>

```

Rysunek 7.6. Początek definicji klasy *Circle* z metodami setter i getter dla zmiennej *@radius*

```

C:\> Wiersz polecenia - irb
>>   def area
>>     Math::PI * @radius**2
>>   end
>>   def perimeter
>>     2 * Math::PI * @radius
>>   end
>> end
=> nil
>>

```

Rysunek 7.7. Koniec definicji klasy *Circle*, jedna z metod zwraca pole koła, druga jego obwód

Aby używać zmiennych instancji:

1. Rozpocznij definiowanie nowej klasy:

```
class Circle
```

Klasa *Circle* będzie reprezentować koła.

2. Zdefiniuj metodę setter:

```
def set_radius(r)
  @radius = r
end
```

Metoda ta przypisuje zmiennej *@radius* wartość przekazaną podczas wywoływania tejże metody.

3. Zdefiniuj metodę getter (rysunek 7.6):

```
def get_radius
  @radius
end
```

Metoda ta zwraca długość promienia koła.

Pamiętaj, że wynik ostatniego wykonanego polecenia w metodzie będzie przez nią zwrócony (patrz rozdział 6., „Tworzenie metod”). Użycie nazwy zmiennej jako ostatniego polecenia w metodzie spowoduje zwrócenie wartości tej zmiennej.

4. Zdefiniuj jeszcze dwie metody:

```
def area
  Math::PI * @radius**2
end
def perimeter
  2 * Math::PI * @radius
end
```

Metody te zwracają obliczone pole koła (ang. *area*) oraz obwód koła (ang. *perimeter*). Do obliczeń wykorzystują zmienną *@radius* oraz pochodzącą z modułu *Math* stałą *PI*.

Wykorzystany do obliczania pola operator potęgowania (****) posiada wyższy priorytet niż operator mnożenia, jeśli jednak chcesz, by wszystko było bardziej jednoznaczne, możesz napisać:

```
Math::PI * (@radius**2)
```

5. Zakończ definiowanie klasy (rysunek 7.7):

```
end
```

6. Utwórz nowe koło oraz ustaw długość promienia: Wskazówki

```
c = Circle.new
c.set_radius(4.59)
```

W pierwszym wierszu tworzony jest nowy obiekt klasy `Circle`. W drugim wierszu wywołujemy metodę `set_radius`, która zmiennej `@radius` przypisuje wartość 4.59.

7. Wyświetl właściwości koła (rysunek 7.8):

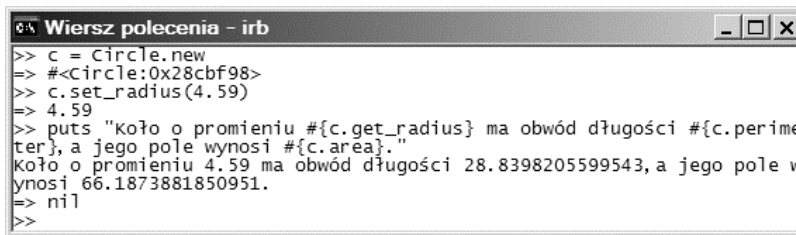
```
puts "Koło o promieniu #{c.get_radius} ma
↳obwód długości #{c.perimeter}, a jego pole
↳wynosi #{c.area}."
```

Trzy metody obiektu są wywoływane, żeby wyświetlić odpowiednie wartości.

- **Zmienne instancji** są dostępne jedynie dla **metod instancji**. Oznacza to na przykład, że nie możesz odwoływać się do zmiennej `@radius` spoza klasy `Circle`.

- W języku Ruby przed użyciem zmiennych nie musisz ich deklarować ani inicjalizować. Dotyczy to także zmiennych instancji. W rzeczy samej **nie powinieneś** próbować deklarować ani inicjalizować zmiennych instancji inaczej niż za pomocą metod. Nie rób czegoś takiego:

```
class Person
  @name = null # Żle!
  def set_name(n)
    @name = n
  end
end
```



```
Wiersz polecenia - irb
>> c = Circle.new
=> #<Circle:0x28cbf98>
>> c.set_radius(4.59)
=> 4.59
>> puts "koło o promieniu #{c.get_radius} ma obwód długości #{c.perime
koło o promieniu 4.59 ma obwód długości 28.8398205599543, a jego pole w
ynosi 66.1873881850951.
=> nil
>>
```

Rysunek 7.8. Przykładowe użycie klasy `Circle`

Akcesory

Jak już wspomniałem kilka razy w tej książce, język Ruby został stworzony po to, aby wszystko było dla programisty łatwe (albo przynajmniej łatwiejsze niż dotychczas). Skoro obecność w klasach metod setter i getter jest powszechna, język Ruby posiada skrót ułatwiający ich tworzenie. Podczas definicji klasy przed definiowaniem jakichkolwiek metod możesz użyć słów kluczowych `attr_reader` oraz `attr_accessor`, aby wskazać zmienne, dla których powinny istnieć metody setter i getter.

```
class NazwaKlasy
  attr_accessor :var1, :var2, :var3
  # inne metody
end
```

Jak widać w powyższym przykładzie, używasz słowa kluczowego `attr_accessor` lub `attr_reader`, a następnie podajesz listę symboli reprezentujących zmienne instancji oddzielone przecinkami. Tak więc definicja klasy `Dog` może zaczynać się w ten sposób:

```
class Dog
  attr_accessor :name, :breed, :gender
```

Zauważ, że składnia jest następująca `:name`, `:breed` i `:gender`, a nie `@name`, `@breed`, i `@gender`.

Jeśli użyjesz `attr_accessor`, właściwe metody setter i getter zostaną utworzone dla każdej zmiennej z listy. Po zakończeniu definiowania klasy `Dog` możesz więc napisać tak (rysunek 7.9):

```
d = Dog.new
d.name = 'Sara'
d.breed = 'York'
d.gender = 'suczka'
puts "Mój pies, #{d.name}, to #{d.gender} rasy
#{d.breed}."
```

Wiersze te pokazują, jak łatwo możesz instancji przypisać zmiennym wartości (to znaczy **ustawić je**) oraz uzyskać ich wartości (to znaczy **pobrać je**, tak jak w powyższej instrukcji `puts`). Jeśli użyjesz słowa kluczowego `attr_reader`, tylko metody getter zostaną utworzone, oznacza to, że możesz zrobić to —

```
puts nazwa_obiektu.nazwa_zmiennej
```

ale nie to —

```
nazwa_obiektu.nazwa_zmiennej = wartość
```

Zmieńmy teraz klasę `Circle`, korzystając z opisanej powyżej techniki. Jeśli do pisania tych przykładów korzystasz z powłoki `irb`, następne kroki prowadzą do ciekawego spostrzeżenia: w języku Ruby możesz modyfikować istniejące klasy w locie.

```
Wiersz polecenia - irb
>> d = Dog.new
=> #<Dog:0x28dafac>
>> d.name = 'Sara'
=> "Sara"
>> d.breed = 'York'
=> "York"
>> d.gender = 'suczka'
=> "suczka"
>> puts "Mój pies, #{d.name}, to #{d.gender} rasy
#{d.breed}."
Mój pies, Sara, to suczka rasy York.
=> nil
>>
```

Rysunek 7.9. Dzięki użyciu `attr_accessor` łatwo jest przypisywać i odczytywać wartości zmiennych instancji

Aby użyć akcesorów:

1. Dodaj akcesor do klasy Circle:

```
class Circle
  attr_accessor :radius
end
```

Te trzy wiersze oznaczają, mówiąc wprost: weź istniejącą definicję klasy `Circle` i zmień ją, dodając metody setter i getter dla zmiennej `@radius`.

Jeśli do ćwiczeń wykorzystujesz powłokę `irb` oraz już wcześniej zdefiniowałeś klasę `Circle`, wówczas nadal posiada ona metody `set_radius`, `get_radius`, `area` oraz `perimeter`. Jeżeli nie korzystasz z powłoki `irb` lub wcześniej nie zdefiniowałeś klasy `Circle`, będziesz musiał dopisać definicje metod `area` i `perimeter`.

2. Utwórz nowe koło i ustaw długość promienia (rysunek 7.10):

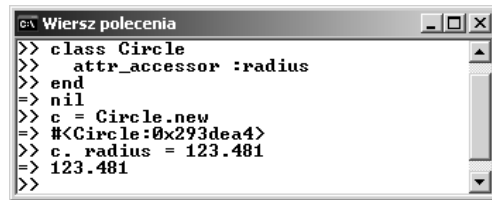
```
c = Circle.new
c.radius = 123.481
```

W pierwszym wierszu tworzysz obiekt klasy `Circle`. W drugim wierszu przypisujesz wartość zmiennej `@radius` w bardziej bezpośredni i czytelny sposób.

3. Wyświetl wymiary koła (rysunek 7.11):

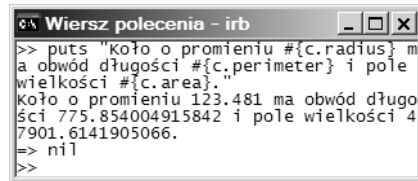
```
puts "Koło o promieniu #{c.radius} ma obwód
↳ długości #{c.perimeter} i pole wielkości
↳ #{c.area}."
```

Wiersz ten jest podobny do tego z poprzedniego przykładu, ale teraz promień jest zwracany poprzez wywołanie po prostu `c.radius`.



```
Wiersz polecenia
>> class Circle
>>   attr_accessor :radius
>> end
=> nil
>> c = Circle.new
=> #<Circle:0x293dea4>
>> c.radius = 123.481
=> 123.481
>>
```

Rysunek 7.10. Mała modyfikacja klasy `Circle` oraz utworzenie nowego obiektu tego typu



```
Wiersz polecenia - irb
>> puts "Koło o promieniu #{c.radius} ma
obwód długości #{c.perimeter} i pole
wielkości #{c.area}."
Koło o promieniu 123.481 ma obwód długo
ści 775.854004915842 i pole wielkości 4
7901.6141905066.
=> nil
>>
```

Rysunek 7.11. Dzięki użyciu akcesora polecenie `puts` może teraz pobrać długość promienia za pomocą `c.radius`



Wskazówki

- Metody setter i getter automatycznie wygenerowane przez wiersz:

```
attr_accessor :nazwazmiennej
```

są równoważne

```
def nazwazmiennej
  @nazwazmiennej
end
def nazwazmiennej(wartość)
  @nazwazmiennej = wartość
end
```

- W rzeczywistości słowa kluczowe `attr_reader` i `attr_accessor` są metodami zdefiniowanymi w klasie `Module`, której klasą potomną jest klasa `Class` (patrz rozdział 8., „Dziedziczenie i cała reszta”).
- W języku Ruby istnieje również metoda `attr`. W wersji 1.8 i starszych tworzy ona metodę getter dla pojedynczej zmiennej, jeśli wpiszesz:

```
attr :nazwa_zmiennej
```

Użyta jak w przykładzie poniżej tworzy obie metody, getter i setter:

```
attr :nazwa_zmiennej, true
```

W wersji 1.9 języka Ruby `attr` może być stosowana tak jak w poprzednich wersjach lub jako alias metody `attr_reader`.

- Metody `attr_reader` oraz `attr_accessor` tworzą proste metody getter i setter. Jeśli potrzebujesz bardziej wyrafinowanych lub elastycznych metod, musisz napisać je sam. Na przykład bardziej dokładna metoda setter klasy `Circle` może sprawdzać, czy promień jest większy od zera.
- Użycie metod `attr_reader` i `attr_accessor` jest przykładem **metaprogramowania** (ang. *metaprogramming*), kiedy to Ruby generuje kod za Ciebie.

Konstruktory

W większości języków zorientowanych obiektowo istnieje metoda, która jest automatycznie wywoływana podczas tworzenia obiektu klasy. W innych językach nazywa ją **konstruktorem**, w języku Ruby określa się je również mianem **inicjalizatora** (ang. *initializer*), ponieważ jej nazwa to zawsze `initialize`. (W języku Java i C++ konstruktor jest metodą o takiej samej nazwie jak nazwa klasy, do której należy).

```
class NazwaKlasy
  def initialize
    #zrób cokolwiek
  end
end
```

Metoda ta będzie automatycznie wywołana w momencie tworzenia obiektu tej klasy za pomocą `NazwaKlasy.new`. Zazwyczaj metoda `initialize` jest wykorzystywana do ustawiania wartości atrybutów klasy, w takim przypadku przyjmuje argumenty:

```
class Dog
  def initialize(name)
    @name = name
  end
end
d = Dog.new('Reksio')
```

Zdefiniowana tutaj klasa `Dog` posiada tylko jedną metodę — `initialize`. W obecnej postaci w każdym obiekcie klasy `Dog` można zmiennej `@name` przypisać wartość, ale nie ma sposobu jej pobrania (to znaczy nie ma metody `getter` dla zmiennej `@name`). Jednym z rozwiązań jest użycie `attr_reader`, żeby Ruby utworzył metodę `getter` za Ciebie.

```
class Dog
  attr_reader :name
  def initialize(name)
    @name = name
  end
end
d = Dog.new('Reksio')
puts "Mój pies wabi się #{d.name}."
```

```

c:\> Wiersz polecenia - irb
>> class Rectangle
>>   attr_reader :height, :width
>>   def initialize(h, w)
>>     @height, @width = h, w
>>   end
>>

```

Rysunek 7.12. Początek definicji klasy *Rectangle* z metodami *getter* dla jej dwóch zmiennych oraz konstruktorem, który przypisuje im wartości

Aby utworzyć konstruktor:

1. Rozpocznij definiowanie nowej klasy:

```

class Rectangle
  attr_reader :height, :width

```

Nowa klasa o nazwie *Rectangle* (prostokąt) będzie reprezentować figurę geometryczną. Posiada ona dwie zmienne instancji, *height* (wysokość) i *width* (szerokość), które reprezentują wymiary prostokąta. Metody *getter* dla nich zostaną stworzone automatycznie dzięki zastosowaniu *attr_reader*.

2. Zdefiniuj konstruktor (rysunek 7.12):

```

  def initialize(h, w)
    @height, @width = h, w
  end

```

Metoda *initialize* przyjmuje dwa argumenty *h* i *w*. Za pomocą równoległego przypisania wartość *h* zostanie przypisana atrybutowi *@height*, a wartość *w* zostanie przypisana atrybutowi *@width*.

3. Zdefiniuj jeszcze dwie metody:

```

  def area
    @height * @width
  end
  def perimeter
    (@height + @width) * 2
  end

```

Podobnie jak metody klasy *Circle* o takich samych nazwach, te będą zwracać pole i obwód figury.

4. Zdefiniuj jeszcze jedną metodę (rysunek 7.13):

```
def is_square?
  if @height == @width then
    true
  else
    false
  end
end
```

Metoda ta zwraca wartość logiczną, która określa, czy prostokąt jest kwadratem (ang. *square*), czy nie. Stosuje się do przyjętej w języku Ruby konwencji kończenia nazw metod znakiem pytajnika (?), jeśli zwracają wartości logiczne (Boolean).

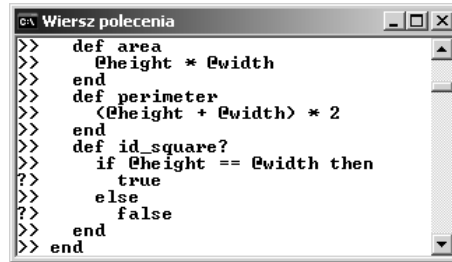
5. Zakończ definicję klasy:

```
end
```

6. Utwórz obiekt typu Rectangle:

```
r = Rectangle.new(10, 34)
```

Ponieważ klasa posiada konstruktor oczekujący dwóch argumentów, dwie wartości muszą być przekazane metodzie new w momencie tworzenia obiektu. Ten prostokąt będzie miał wysokość 10 i szerokość 34.



```
Wiersz polecenia
>> def area
>>   @height * @width
>> end
>> def perimeter
>>   (<@height + @width> * 2
>> end
>> def id_square?
>>   if @height == @width then
?>     true
>>   else
?>     false
>>   end
>> end
```

Rysunek 7.13. Trzy metody dodane do klasy Rectangle dostarczą podstawowych informacji o figurze

7. Użyj metod obiektu (rysunek 7.14):

```
puts "Prostokąt o wysokości #{r.height}
↳ i szerokości #{r.width} posiada obwód
↳ długości #{r.perimeter}, a jego pole
↳ wynosi #{r.area}."
puts "Prostokąt ten #{r.is_square? ? 'jest' :
↳ 'nie jest'} kwadratem."
```

Pierwsze polecenie puts pokazuje, jak można wykorzystać automatycznie stworzone metody do pobrania wymiarów prostokąta. Polecenie to wywołuje również metody perimeter i area.

Drugie polecenie puts wykorzystuje metodę is_square? z operatorem trzyargumentowym (ang. *ternary*), aby wyświetlić *Prostokąt ten jest kwadratem* lub *Prostokąt ten nie jest kwadratem*. Więcej o składni tego operatora możesz przeczytać w rozdziale 5., „Struktury sterujące”.

8. Utwórz nowy prostokąt i przepisz wiersze z kroku 7. (rysunek 7.15):

```
r = Rectangle.new(14, 14)
puts "Prostokąt o wysokości #{r.height}
↳ i szerokości #{r.width} posiada obwód
↳ długości #{r.perimeter}, a jego pole
↳ wynosi #{r.area}."
puts "Prostokąt ten #{r.is_square? ? 'jest' :
↳ 'nie jest'} kwadratem."
```

Polecam ćwiczenia z innym prostokątem, który jest również kwadratem, choćby po to, żeby zobaczyć wynik. Możesz utworzyć nowy obiekt klasy Rectangle, aby nadpisać istniejącą zmienną r.

```

C:\ Wiersz polecenia - irb
>> r = Rectangle.new(10, 34)
=> #<Rectangle:0x28de670 @width=34, @height=10>
>> puts "Prostokąt o wysokości #{r.height} i szerokości #{r.width} posiada obwód długości #{r.perimeter}, a jego pole wynosi #{r.area}."
Prostokąt o wysokości 10 i szerokości 34 posiada obwód długości 88, a jego pole wynosi 340.
=> nil
>> puts "Prostokąt ten #{r.is_square? ? 'jest' : 'nie jest'} kwadratem."
Prostokąt ten nie jest kwadratem.
=> nil
>>

```

Rysunek 7.14. Utworzenie obiektu klasy Rectangle i wywołanie jego pięciu metod

```

C:\ Wiersz polecenia - irb
>> r = Rectangle.new(14, 14)
=> #<Rectangle:0x28cad00 @width=14, @height=14>
>> puts "Prostokąt o wysokości #{r.height} i szerokości #{r.width} posiada obwód długości #{r.perimeter}, a jego pole wynosi #{r.area}."
Prostokąt o wysokości 14 i szerokości 14 posiada obwód długości 56, a jego pole wynosi 196.
=> nil
>> puts "Prostokąt ten #{r.is_square? ? 'jest' : 'nie jest'} kwadratem."
Prostokąt ten jest kwadratem.
=> nil
>>

```

Rysunek 7.15. Utworzenie obiektu klasy Rectangle, który jest równocześnie kwadratem

Definiowanie operatorów

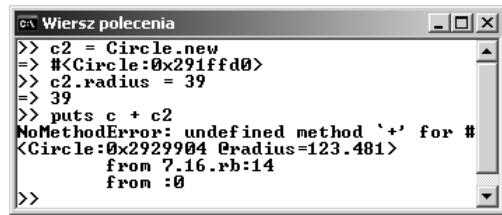
Tak jak obiekty posiadają metody służące do wykonywania różnych zadań, tak samo mogą wykorzystywać operatory — arytmetyczne, przypisania i logiczne — których sposób działania możesz zdefiniować. Aby to zrobić, definiujesz metodę, której nazwa jest operatorem: +, -, == itd. Operatory takie wymagają dwóch argumentów — `4 - 2`, `ciąg_znaków + inny_ciąg_znaków`, więc właściwa metoda musi operować na dwóch obiektach. Pierwszym obiektem będzie operand z lewej strony (`4` i `ciąg_znaków` w przedstawionym przykładzie). Ten obiekt jest dostępny w metodzie poprzez odwołanie do `self` (patrz rozdział 6., „Dziedziczenie i cała reszta”) lub do zmiennych instancji klasy. Operand z prawej strony (`2` i `inny_ciąg_znaków`) powinien być przyjmowany jako jedyny argument metody:

```
class NazwaKlasy
  def *(rh)
    # Rób coś z self i rh
  end
end
```

To podstawowa składnia implementacji operatora mnożenia dla klasy. Od Ciebie, twórcy klasy, zależy, co metoda powinna robić w momencie użycia operatora. Możesz zdecydować się na pozostawienie operatora niezdefiniowanego, co spowoduje wystąpienie błędu, jeśli ktoś użyje go z danym obiektem (rysunek 7.16).

Jako przykład weźmy dodawanie dwóch obiektów typu `Rectangle`:

```
r1 = Rectangle.new(10, 20)
r2 = Rectangle.new(6, 8)
puts r1 + r2
```



```
c:\ Wiersz polecenia
>> c2 = Circle.new
=> #<Circle:0x291ffd0>
>> c2.radius = 39
=> 39
>> puts c + c2
NoMethodError: undefined method '+' for #
<Circle:0x2929904 @radius=123.481>
      from 7.16.rb:14
      from :0
>>
```

Rysunek 7.16. Jeśli klasa nie posiada zdefiniowanej metody dla danego operatora, użycie go z obiektami klasy spowoduje wystąpienie błędu `NoMethodError`

```

c:\ Wiersz polecenia
>> c2 = Circle.new
=> #<Circle:0x2919298>
>> c2.set_radius<10>
=> 10
>> puts c2 == c
false
=> nil
>> puts c = c2
#<Circle:0x2919298>
=> nil
>> puts c
#<Circle:0x2919298>
=> nil
>> puts c.area
314.159265358979
=> nil
>>

```

Rysunek 7.17. Program, w którym wykorzystano dwa obiekty `Circle` w celu demonstracji działania operatorów przypisania i porównania bez konieczności definiowania ich w klasie

Czy w tym przypadku wynikiem powinna być suma pól prostokątów ($200 + 48 = 248$), czy suma ich obwodów ($60 + 28 = 88$)? Być może powinna obliczać sumę wysokości i szerokości prostokątów:

```

def +(rh)
  @height += rh.height
  @width += rh.width
  self
end

```

W przypadku klasy `Rectangle` każde podejście powoduje powstanie problemów. Raz jeszcze — od Ciebie zależy, jak tego typu sytuacje będą obsługiwane, jeśli w ogóle: może okazać się, że dla danej klasy operacje dodawania, odejmowania, mnożenia czy dzielenia nie mają sensu.

Inną decyzją, którą należy podjąć, jest: czy operacja ma zwrócić nowy obiekt, czy modyfikować istniejący (patrz ramka „Obiekty zmieniające same siebie” na stronie 171. W poprzednim przykładzie modyfikowane są wartości bieżącego obiektu, a następnie zwracane. Aby utworzyć i zwrócić nowy obiekt, metoda może wyglądać następująco:

```

def +(rh)
  Rectangle.new(@height + rh.height, @width +
    rh.width)
end

```

Na koniec niektóre obiekty posiadają po dwa operatory lub metody robiące to samo, a różniące się tym, że jedna modyfikuje bieżący obiekt, a druga zwraca nowy obiekt, pozostawiając bieżący bez zmian. Jako przykład weźmy klasę `String`, która posiada dwie formy łączenia (konkatenacji): `+`, zwracający nowy obiekt, oraz `<<`, modyfikujący bieżący.

Klasy mogą chcieć definiować własne wersje operatorów przypisania i logicznych. Język Ruby automatycznie generuje operatory przypisania (`=`) i porównania (`==`) (rysunek 7.17), możesz jednak chcieć zmienić ich zachowanie.

W przeciwieństwie do poprzednich ogólny operator porównania, `<=>`, nie jest automatycznie definiowany. Definiując metodę `<=>`, możesz sprawić, że obiekty danego typu będzie można porównywać i sortować. Po prostu zwracaj `-1`, jeśli lewy operand jest mniejszy, `1`, jeśli prawy operand jest mniejszy, i `0`, jeśli są równe. Na przykład jako podstawę do porównywania dwóch obiektów typu `Rectangle` weźmy ich pola:

```
def <=>(rh)
  if self.area < rh.area then -1
  elsif self.area > rh.area then 1
  else 0 end
end
```

Jako „z życia wzięty” przykład wykorzystania tej wiedzy podam moje hobby: stolarkę¹. Kiedy zaczynam nowy projekt, muszę wiedzieć, ile desek posiadam, żeby wiedzieć, czy ich wystarczy. Powierzchnia całkowita desek — które posiadam lub których projekt wymaga — jest sumą powierzchni pojedynczych deseczek. Tabela 7.1 zawiera przykładowe dane.

Podczas definiowania klasy przyjąłem kilka założeń. Zakładam, że dwie deski są takie same, jeśli mają tę samą powierzchnię. Liczba desek jest raczej nieistotna, dwie mniejsze mogą posłużyć do stworzenia większej. Po drugie, dla uproszczenia pomijam grubość desek. W rzeczywistości możesz mieć deski, które mają półtorej cala grubości (tak zwane 6/4), a powierzchnia, którą możesz uzyskać, zależy od ich grubości (zakładając tę samą kubaturę). Mając deseczki o grubości 3/4, będziesz miał dwa razy większą powierzchnię; użycie deseczek 2/4 oznacza trzy razy większą powierzchnię. Możesz uwzględnić to w klasie, oznacza to jedynie trochę więcej matematyki.

Tabela 7.1. Powierzchnia całkowita

szerokość	długość	powierzchnia
4	30	120
5.5	22	121
6	14	84
Powierzchnia całkowita		325

¹ Aby zrozumieć poniższy „stolarski” przykład „z życia wzięty”, należy się małe wytłumaczenie. W oryginale autor używa amerykańskiej miary drewna — *board foot*. *board foot* — deska o długości jednej stopy, szerokości jednej stopy i grubości jednego cala lub odpowiednik o takiej samej kubaturze. Źródło: http://www.woodweb.com/knowledge_base/What_is_a_Board_Foot.html. Tak więc *board foot* jest miarą objętości (kubatury), a nie powierzchni. Z tego powodu deski o różnych grubościach posiadają różne powierzchnie przy takiej samej objętości — proste. Autor zbija stół ... — *przyp. tłum.*

Listing 7.1. Klasa `Wood` umożliwiająca dodawanie i porównywanie obiektów typu `Wood`

```
Listing
1 # Skrypt 7.1 - wood.rb
2
3 # Klasa reprezentująca zbiór desek.
4 class Wood
5
6   # Dostęp do-odczytu do tablicy boards:
7   attr_reader :boards
8
9   # Konstruktor, może inicjalizować tablicę boards.
10  def initialize(*args)
11    @boards = args
12  end
13
14  # Metoda dodaje pojedynczą deseczkę.
15  def add_board(board)
16    @boards.push(board)
17  end
18
19  # Metoda zwraca całkowitą powierzchnię desek.
20  def total
21    t = 0
22    @boards.each { |b| t += (b[:width] *
23      ↳b[:length]) }
24    t
25  end
26  # Metoda sprawdzająca, czy wystarczy drewna
27  # dla danego projektu
28  def enough?(project)
29    project.total <= self.total
30  end
31
32  # Operator dodawania:
33  def +(rh)
34    @boards += rh.boards
35    self
36  end
37
38  # Operator porównania:
39  def ==(rh)
40    self.total == rh.total
41  end
42
43  # Operator identyczność:
44  def ==(rh)
45    @boards == rh.boards
46  end
47
```

Aby zdefiniować operatory dla klas:

1. Utwórz nowy skrypt w edytorze tekstu lub IDE (Listing 7.1):

```
# Skrypt 7.1 - wood.rb
```

Ponieważ przykład ten będzie zawierał dosyć dużo kodu, znajdowanie błędów i ich poprawianie będzie bardzo kłopotliwe, jeśli używasz powłoki `irb` lub `fxri`. Dlatego też zdecydowałem się napisać go jako skrypt w osobnym pliku. Po więcej informacji, jak pisać i uruchamiać skrypty w języku Ruby, odsyłam do rozdziału 2., „Proste skrypty”.

2. Rozpocznij definicję klasy `Wood`:

```
class Wood
  attr_reader :boards
  def initialize(*args)
    @boards = args
  end
end
```

Najpierw metoda getter jest definiowana dla zmiennej `@boards`. Następnie definiowana jest metoda `initialize` w taki sposób, aby mogła przyjmować zmienną liczbę argumentów pod postacią tablicy (o operatorze *splat* możesz się dowiedzieć więcej w rozdziale 6., „Tworzenie metod”). Zmienna `@boards` będzie tablicą haszów, po jednym dla każdej deseczki. Hasze będą używać symboli jako kluczy. Aby utworzyć obiekt zawierający dwie deseczki, napiszesz:

```
walnut = Wood.new({:width => 5, :length =>
↳26.25}, {:width => 4.75, :length => 32})
```

Jeśli podczas tworzenia nowego obiektu nie będą przekazane żadne dane, wówczas `@boards` będzie pustą tablicą (ponieważ `args` będzie pustą tablicą).

3. Zdefiniuj metodę add_board:

```
def add_board(board)
  @boards.push(board)
end
```

Jest to prosta metoda służąca do dodawania deseczek do kolekcji poprzez wstawianie przekazanego argumentu do tablicy.

4. Zdefiniuj metodę total:

```
def total
  t = 0
  @boards.each { |b| t +=(b[:width] *
    ↪b[:length]) }
  t
end
```

Metoda total zwraca całkowitą powierzchnię reprezentowaną przez obiekt. Aby to zrobić, inicjalizuje zmienną liczbą 0, a następnie iteruje po elementach tablicy @boards. Powiązany z iteratorem blok kodu będzie otrzymywał jedną deseczkę (zmienną typu Hash). Następnie doda do zmiennej t wynik mnożenia szerokości i długości deseczki. Na koniec zwracana jest suma.

5. Zdefiniuj metodę enough?:

```
def enough?(project)
  project.total <= self.total
end
```

Metoda ta zwraca informację, czy posiadany zapas drewna jest wystarczający dla danego projektu. Aby to zrobić, zwraca true, jeśli powierzchnia potrzebna dla danego projektu jest mniejsza lub równa całkowitej powierzchni desek w obiekcie. Metoda będzie używana w następujący sposób (maple i table są obiektami typu Wood):

```
puts "Wybierz inny rodzaj drewna!" if
  ↪!maple.enough?(table)
```

Listing 7.1. Klasa Wood umożliwia dodawanie i porównywanie obiektów typu Wood — ciąg dalszy

```
Listing
48 # Ogólny operator porównania:
49 def <=>(rh)
50   if self.total < rh.total then -1
51     elsif self.total > rh.total then 1
52     else 0 end
53   end
54
55 end # Koniec definicji klasy Wood.
56
57 # Utwórz nowy obiekt klasy Wood:
58 maple = Wood.new({:width => 7, :length =>
  ↪42}, {:width => 4.5, :length => 22},
  ↪{:width => 6.75, :length => 32.5},
  ↪{:width => 5, :length => 26.25})
59
60 # Utwórz drugi obiekt klasy Wood:
61 oak = Wood.new
62 oak.add_board({:width => 5, :length =>
  ↪26.25})
63 oak.add_board({:width => 6.25, :length =>
  ↪27.5})
64
65 # Kilka testów:
66 print "Całkowita powierzchnia desek
  ↪z klonu: "
67 puts maple.total
68 print "Całkowita powierzchnia desek
  ↪dębowych: "
69 puts oak.total
70 print "Czy całkowite powierzchnie desek
  ↪z klonu i dębowych są równe? "
71 puts maple == oak
72 print "Porównanie całkowitych powierzchni
  ↪desek z klonu i dębowych: "
73 puts maple <=> oak
74
75 # Kup więcej desek dębowych:
76 oak += Wood.new({:width => 7, :length =>
  ↪19}, {:width => 5.5, :length => 22.2})
77
78 # Sprawdź, czy wystarczy dla projektu:
79 table = Wood.new({:width => 4.5, :length
  ↪=> 22}, {:width => 6.75, :length =>
  ↪32.5}, {:width => 5, :length => 26.25},
  ↪{:width => 5, :length => 26.25})
80 print "Wymagana powierzchnia dla stołu: "
81 puts table.total
82 print "Czy wystarczy desek z klonu? "
83 puts maple.enough?(table) ? "Do pracy!" :
  ↪"Niestety!"
84 print "Czy wystarczy desek dębowych? "
85 puts oak.enough?(table) ? "Do pracy!" :
  ↪"Niestety!"
```



Obiekty zmieniające same siebie

Niektóre metody klas modyfikują obiekty, do których należą. Przyjęła się konwencja, że ich nazwa kończy się wykrzyknikiem, zaznaczając, że należy ich używać z ostrożnością (na przykład metoda `slice!` klasy `Array`). Metoda może aktualizować obiekt poprzez zmianę wartości zmiennych instancji.

```
def double!
  @some_var *= 2
end
```

Ważną sprawą, na którą należy zwrócić uwagę, jest wartość, którą zwraca metoda. Poprzednia metoda zwraca wartość zmiennej `@some_var` pomnożoną razy 2. Wszystko jest w porządku, dopóki obiekt nie zostanie wykorzystany w taki sposób:

```
my_obj = SomeClassName.new
my_obj = my_obj.double!
```

W tym momencie zmienna `my_obj` nie jest już instancją klasy `SomeClassName`, zamiast tego posiada wartość `@some_var` razy 2. Rozwiązaniem tego problemu jest zwracanie przez metodę bieżącego obiektu, reprezentowanego przez `self`:

```
def double!
  @some_var *= 2
  self
end
```

Jeśli potrzebujesz metody spełniającej to samo zadanie, ale bez modyfikowania bieżącego obiektu (tak jak metoda `slice` klasy `Array`), niech najpierw utworzy kopię bieżącego obiektu, a następnie wywoła metodę, która zmienia bieżący obiekt:

```
def double
  copy = self.dup
  copy.double!
end
```

6. Zdefiniuj operator dodawania:

```
def +(rh)
  @boards += rh.boards
  self
end
```

Jeśli posiadasz zapas drewna i ktoś sprezentuje Ci swój, oba zapasy będą do siebie dodane:

```
my_oak += your_oak
```

Użycie operatora `+` dodaje tablicę `@boards`, która jest prawym operandem do tablicy `@boards` bieżącego obiektu. Metoda zwraca bieżący obiekt (`self`).

7. Zdefiniuj operatory przyrównania i identyczności:

```
def ==(rh)
  self.total == rh.total
end
def ==(rh)
  @boards == rh.boards
end
```

Aby zademonstrować, w jaki sposób można to zrobić, nawet jeśli w rzeczywistości nigdy nie zostaną użyte, operatorom `==` (przyrównanie) i `===` (identyczność) przypisano funkcje. Obiekty będą równe, gdy dwie kolekcje będą miały taką samą powierzchnię, nawet jeśli będą miały różną liczbę deseczek. Identyczność jest bardziej restrykcyjną wersją przyrównania. Tutaj zdefiniowano ją jako sprawdzenie, czy dwa obiekty posiadają tablice `@boards` z takimi samymi elementami i w tej samej kolejności.

8. Zdefiniuj ogólny operator porównania:

```
def <=>(rh)
  if self.total < rh.total then -1
  elsif self.total > rh.total then 1
  else 0 end
end
```

Jeśli lewy operand (bieżący obiekt) ma mniejszą powierzchnię niż prawy operand, zwraca się -1. Jeżeli posiada większą, wówczas zwraca się 1. Jeśli oba mają taką samą powierzchnię, zwraca się 0 i przyjmuje się, że obiekty są równe.

9. Zakończ definiowanie klasy i utwórz kilka obiektów tego typu:

```
end
maple = Wood.new({:width => 7, :length => 42},
↳{:width => 4.5, :length => 22}, {:width =>
↳6.75, :length => 32.5}, {:width => 5,
↳:length => 26.25})
oak = Wood.new
oak.add_board({:width => 5, :length => 26.25})
oak.add_board({:width => 6.25, :length =>
↳27.5})
```

Najpierw utwórz nowy obiekt typu Wood o nazwie maple (klon). Podczas jego tworzenia przypisuje się mu deseczki. Następnie tworzę nowy obiekt typu Wood o nazwie oak (dąb). Na początku nie posiada on desek, więc dwie zostają dołożone.

10. Pobaw się z obiektami:

```
print "Całkowita powierzchnia desek z klonu: "
puts maple.total
print "Całkowita powierzchnia desek dębowych: "
puts oak.total
print "Czy całkowite powierzchnie desek z klonu
↳i dębowych są równe? "
puts maple == oak
print "Porównanie całkowitych powierzchni
↳desek z klonu i dębowych: "
puts maple <=> oak
```

Aby zobaczyć, co te obiekty potrafią zrobić, wywołujemy metodę total oraz wykorzystujemy działanie dwóch operatorów.

11. Dołóż kilka desek dębowych:

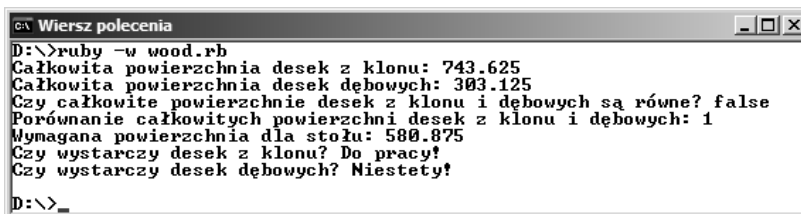
```
oak += Wood.new({:width => 7, :length => 19},
↳{:width => 5.5, :length => 22.2})
```

W ten sposób sprawdzamy działanie operatora + dla obiektów klasy Wood.

12. Sprawdź, czy wystarczy drewna klonowego lub dębowego dla danego projektu:

```
table = Wood.new({:width => 4.5, :length => 22},
↳{:width => 6.75, :length => 32.5}, {:width => 5,
↳:length => 26.25}, {:width => 5, :length =>
↳26.25})
print "Wymagana powierzchnia dla stołu: "
puts table.total
print "Czy wystarczy desek z klonu? "
puts maple.enough?(table) ? "Do pracy!":
↳"Niestety!"
print "Czy wystarczy desek dębowych? "
↳puts oak.enough?(table) ? "Do pracy!":
"Niestety!"
```

Tworzymy nowy obiekt typu Wood o nazwie table (stół). Symbolizuje on hipotetyczny stół, który chcemy zbić z desek. Mając ten obiekt, możemy sprawdzić, czy wystarczy desek dla naszego projektu, przez porównanie go z innymi obiektami klasy Wood.

13. Zapisz i uruchom skrypt (rysunek 7.18).


```

C:\> Wiersz polecenia
D:\> ruby -w wood.rb
Całkowita powierzchnia desek z klonu: 743.625
Całkowita powierzchnia desek dębowych: 303.125
Czy całkowite powierzchnie desek z klonu i dębowych są równe? false
Porównanie całkowitych powierzchni desek z klonu i dębowych: 1
Wymagana powierzchnia dla stołu: 580.875
Czy wystarczy desek z klonu? Do pracy!
Czy wystarczy desek dębowych? Niestety!
D:\> _

```

Rysunek 7.18. Wynik wykonania skryptu wood.rb

Wskazówki

- Jeśli potrzebujesz zdefiniować jednoargumentowy minus lub plus, nazwij metody w następujący sposób: `-@`, `+@`. Nie możesz po prostu użyć `-` i `+`, ponieważ reprezentują one odejmowanie i dodawanie.
- Definicję klasy zazwyczaj umieszcza się w osobnym pliku, a następnie dołącza w skryptach, które używają danej klasy. W rozdziale 9, „Moduły”, zobaczysz, jak to zrobić.
- Aby umożliwić traktowanie obiektu jak tablicy, należy zdefiniować metodę `[]`. Na przykład następna metoda umożliwi dostęp do atrybutu `@height` klasy `Rectangle` w następujący sposób: `nazwa_obiektu[0]` lub `nazwa_obiektu[-2]`, a do atrybutu `@width` w następujący: `nazwa_obiektu[1]` lub `nazwa_obiektu[-1]`:

```
def [](i)
  case i
  when 0, -2: @height
  when 1, -1: @width
  else nil
  end
end
```

Jeśli chcesz, aby obiekt mógł być taktowany jako hasz, możesz dodać klauzule `when` dla `:height` i `'height'` oraz `:width` i `'width'`.

Metody specjalne

Z wielkimi możliwościami definiowania i korzystania z własnych klas wiąże się obowiązek właściwego ich definiowania. Jednym z tych obowiązków jest takie ich definiowanie, aby zachowywały się tak jak inne obiekty języka Ruby (o ile to możliwe). Na przykład większość klas posiada metodę `to_s`, która zwraca tekstową reprezentację obiektu. Dla typów liczbowych `to_s` zwraca wartość jako ciąg znaków. Dla tablic `to_s` jest równoważna użyciu `join`.

Aby utworzyć własną metodę `to_s`, po prostu zdefiniuj metodę o takiej nazwie. Można się spodziewać, że metoda taka może zwracać wartości zmiennych klas:

```
class Dog
  def initialize(name)
    @name = name
  end
  def to_s
    @name
  end
end
```

Dla klasy `Rectangle` możesz zwrócić wartości umieszczone w kontekście (rysunek 7.19):

```
class Rectangle
  attr_reader :height, :width
  def initialize(h, w)
    @height, @width = h, w
  end
  def to_s
    "wysokość: #@height, szerokość: #@width"
  end
  # pozostałe metody
end
```

W niektórych klasach sensowne jest również zdefiniowanie metody `each`, aby umożliwić iterację. Definiowanie tej metody nie zawsze jest celowe, ale może być przydatne, jeśli klasa przechowuje wiele wartości. Zostanie to pokazane w następnym przykładzie — klasie zarządzającej listami zakupów. Lista przechowywana jest w haszu, którego klucze są nazwami produktów, a wartości liczbą produktów. Metoda `initialize` tworzy nowy hasz. Metoda `add` dodaje element do listy. W klasie zaimplementowano również metody `each` oraz `to_s`, dlatego w miarę potrzeby obiekt zawierający listę zakupów będzie mógł być traktowany jako hasz.

```
g:\ Wiersz polecenia - irb
>> class Rectangle
>>   attr_reader :height, :width
>>   def initialize(h, w)
>>     @height, @width = h, w
>>   end
>>   def to_s
>>     "wysokość: #@height, szerokość: #@width"
>>   end
>>   # pozostałe metody
?> end
=> nil
>> r = Rectangle.new(13, 21)
=> #<Rectangle:0x2ece418 @width=21, @height=13>
>> puts r.to_s
wysokość: 13, szerokość: 21
=> nil
>>
```

Rysunek 7.19. Przykład działania metody `to_s` dla obiektów typu `Rectangle`

Aby utworzyć metody to_s i each:

1. Utwórz nowy skrypt w edytorze lub IDE (listing 7.2):

```
# Skrypt 7.2 - groceries.rb
```

Tak jak poprzedni, ten przykład również będzie napisany jako skrypt.

2. Rozpocznij definicję klasy GroceryList:

```
class GroceryList
  def initialize
    @items = Hash.new
  end
```

Metoda `initialize` nie przyjmuje żadnych argumentów, tworzy pusty obiekt typu `Hash`. Zauważ, że w klasie nie zdefiniowano żadnych akcesorów. Wartości będą dodawane za pomocą metody `add`, listę możesz przeglądać przy użyciu `each`.

3. Zdefiniuj metodę `add`:

```
def add(item, qty = 1)
  @items[item] = qty
end
```

Metody tej będziemy używać do dodawania produktów do listy. Przyjmuje dwa argumenty: nazwę produktu i liczbę. Dzięki ustawieniu domyślnej liczby na 1 podanie jej jest opcjonalne. Wewnątrz tej metody nowe produkty są dodawane poprzez dodanie nowego elementu do hasza. Nazwa produktu jest kluczem, a liczba produktów stanowi wartość elementu hasza.

Jeśli dany element istnieje, liczba zostanie nadpisana.

4. Zdefiniuj metodę `each`:

```
def each
  @items.each_pair { |k, v| yield k, v }
end
```

Listing 7.2. Udostępniając metody `each` oraz `to_s`, klasa `GroceryList` zachowuje się podobnie do standardowych klas w języku Ruby

```
Listing
1 # Skrypt 7.2 - groceries.rb
2
3 # Definicja nowej klasy:
4 class GroceryList
5
6   # Konstruktor tworzy pusty hasz.
7   def initialize
8     @items = Hash.new
9   end
10
11  # Metoda dodająca elementy do listy.
12  # Liczba jest opcjonalna, domyślnie 1.
13  # Nazwa elementu jest używana jako klucz.
14  def add(item, qty = 1)
15    @items[item] = qty
16  end
17
18  # Metoda służąca do iterowania po elementach listy.
19  # Zwraca klucz i wartość.
20  def each
21    @items.each_pair { |k, v| yield k, v }
22  end
23
24  def to_s
25    str = ''
26    @items.each_pair { |k, v| str
27      += "#{k}: #{v}, "
28    }
29    str.slice(0, str.length - 2).to_s
30  end # Koniec definicji klasy GroceryList.
31
32  # Utwórz nową listę:
33  g = GroceryList.new
34
35  # Dodaj kilka elementów:
36  g.add('cukier', '5kg')
37  g.add('kiwi', 4)
38  g.add('stek')
39
40  # Wyświetl listę jako ciąg znaków:
41  puts "Lista jako ciąg znaków=> " + g.to_s
42
43  # Wyświetl każdy element listy w nowym wierszu:
44  puts "Lista zakupów"
45  g.each { |item, qty| puts "#{qty} #{item}" }
```

Metoda `each`, będąca iteratorem, powinna poruszać się po elementach hasza i zwracać wartości do powiązanego z nią bloku kodu. (Iteratory zawsze korzystają z `yield` i wywoływane są z blokiem kodu, patrz rozdziały 5., „Struktury sterujące”, i 6., „Tworzenie metod”).

Do poruszania się po elementach hasza wykorzystuje się iterator `each_pair`. W każdym kroku iteracji klucz i wartość hasza zostaną przypisane zmiennym `k` i `v` bloku. Następnie zostają one zwrócone przez iterator (do bloku kodu, w którym została wywołana metoda). Może się to wydawać niepotrzebne, ale można to wytłumaczyć w ten sposób: `@items` jest haszem, więc może korzystać z metody `each_pair` (lub `each`) klasy `Hash`; `GroceryList` nie jest haszem, więc musi zdefiniować własną metodę `each` (nawet jeśli ta korzysta z innego iteratora).

5. Zdefiniuj metodę `to_s`:

```
def to_s
  str = ''
  @items.each_pair { |k, v| str += "#{k}: "
    ↪ "#{v}, " }
  str.slice(0, str.length - 2).to_s
end
```

Metoda ta zwróci hasz zamieniony w ciąg znaków postaci *klucz: wartość*, gdzie każda para klucz – wartość będzie oddzielona przecinkiem i spacją. Najpierw zostaje zdefiniowany pusty ciąg znaków, następnie iterator `each_pair` zwraca każdy element hasza. Wewnątrz powiązanego bloku kodu każda para jest dołączana do ciągu znaków. Następnie ciąg znaków jest zwracany bez dwóch ostatnich znaków (kończącego przecinka i spacji).

Ponieważ metoda `slice` zwróci `nil`, jeśli ciąg znaków będzie pusty, na koniec wywołuje się metodę `to_s`, która w razie potrzeby zamiast `nil` zwróci pusty ciąg znaków.

6. Zakończ definicję klasy i utwórz obiekt tego typu:

```
end
g = GroceryList.new
```

7. Dodaj kilka elementów do listy:

```
g.add('cukier', '5kg')
g.add('kiwi', 4)
g.add('stek')
```

Nie został zdefiniowany sposób zapisu liczby produktów, więc możesz używać liczb oraz ciągów znaków. Ponieważ drugi argument posiada domyślną wartość 1, jest opcjonalny (jak pokazano w momencie dodania steku do listy).

8. Wyświetl listę zakupów w postaci ciągu znaków:

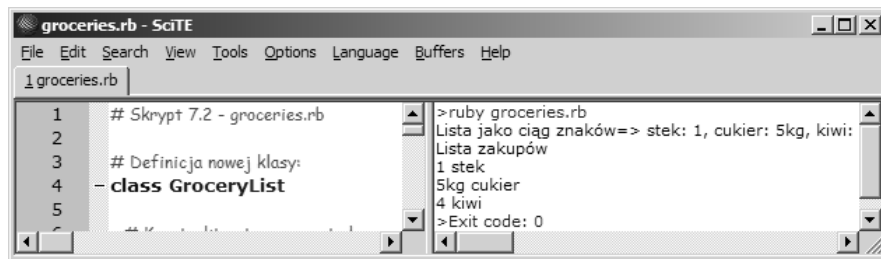
```
puts "Lista jako ciąg znaków=> " + g.to_s
```

Aby to zrobić, możesz skorzystać z metody `to_s`. Jako ciąg znaków może zostać wyświetlona, zamieniona w tablicę, można ją przeszukiwać, można z nią zrobić wszystko to, co robi się ze standardowym obiektem klasy `String`.

9. Wyświetl każdy element listy w osobnym wierszu:

```
puts "Lista zakupów"
g.each { |item, qty| puts "#{qty} #{item}" }
```

Najpierw wyświetlany jest tytuł, następnie wywołuje się metodę `each`. Zwraca ona każdy produkt i jego liczbę. Wartości te zostaną przypisane zmiennym `item` i `qty` w bloku, a następnie wyświetlone.

10. Zapisz i uruchom skrypt (rysunek 7.20).


```
groceries.rb - SciTE
File Edit Search View Tools Options Language Buffers Help
1 groceries.rb
1 # Skrypt 7.2 - groceries.rb
2
3 # Definicja nowej klasy:
4 - class GroceryList
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2
```

Wskazówki

- Innym sposobem implementacji metody `to_s` jest skorzystanie z metody `each` klasy:

```
def to_s
  str = ''
  self.each { |k, v| str += "#{k}: #{v}, " }
  str.slice(0, str.length - 2).to_s
end
```

- Inną metodą, której implementację powinieneś rozważyć, jest `coerce`. Metoda ta zamienia miejscami argumenty w wykonywanym poleceniu:

```
def coerce(rh)
  [self, rh]
end
```

Metoda ta jest niezbędna, jeśli klasa może zostać użyta w operacjach arytmetycznych z inną klasą. Na przykład klasa `Wood` obsługuje mnożenie, więc możesz podwoić lub potroić zapas drewna. W takim przypadku zadziała `maple * 2`, ale `2 * maple` — już nie (ponieważ klasa `Fixnum` nie pozwala na mnożenie jej przez obiekt `Wood`). Jeśli w klasie `Wood` zdefiniujemy metodę `coerce`, będzie ona mogła zamienić miejscami operandy i wykonać mnożenie.

- Jeśli w klasie `GroceryList` zdefiniujemy metodę odczytującą dla zmiennej `@items` —

```
attr_reader :items
```

— wówczas nie trzeba będzie definiować metody `each` wewnątrz klasy, ponieważ będzie można skorzystać z niej na zewnątrz klasy:

```
g.items.each { |item, qty| puts "#{qty}
↳#{item}" }
```

To jednak zakłada, że użytkownik wiedział, iż w klasie istnieje zmienna `@items` oraz że jest ona haszem. Najlepiej by było, gdyby klasy były tak tworzone, aby można było z nich korzystać bez znajomości, a nawet bez dostępu do ich implementacji. Nazywa się to **enkapsulacją**; szerzej opiszę to w następnym rozdziale.

Walidacja i duck typing

W klasie Wood przedstawionej w skrypcie 7.1 istnieją błędy projektowe: zakłada się, że jej metody i operatory będą używane właściwie. Metoda `enough?`, podobnie jak operator przyrównania, zakłada, że otrzyma obiekt z metodą `total`. Operatory `+` i `==` zakładają, że będą wykorzystywane jedynie z operandami posiadającymi zmienną `@boards`. Każde z poniższych wywołań spowoduje wystąpienie błędów (rysunek 7.21):

```
puts !maple.enough?(439.50)
oak += 35
```

Aby zapobiec niewłaściwemu użyciu metod, możesz dodać kod walidujący, który przerwie wykonywanie metody, jeśli nie będą spełnione określone warunki. Jednym z rodzajów walidacji jest sprawdzenie typu używanych obiektów. Metoda `instance_of?` zwraca wartość logiczną, która wskazuje, czy jest to obiekt ustalonej klasy:

```
class ThisClass
  def -(rh)
    return nil unless rh.instance_of?
      ↳ ThisClass
    # W porządku, wykonaj odejmowanie.
  end
end
```

Metoda będzie zwracać `nil`, jeśli prawy operand nie będzie typu `ThisClass`.

Jeśli chcesz być bardziej elastyczny, możesz sprawdzać, czy obiekt nie jest obiektem danego typu lub typu pochodnego. Dziedziczenie jest dokładnie opisane w następnym rozdziale, na razie wiedz, że umożliwia ono pochodzenie jednej klasy od innej.

```
class Rectangle
  def <=>(rh)
    return nil unless rh.is_a? Shape
    self.area <=> rh.area
  end
end
```

W powyższym przykładzie zakłada się, że można porównywać obiekty klasy `Rectangle` z obiektami klasy `Shape` oraz jej klas pochodnych. Może to dotyczyć klas `Rectangle`, `Circle` oraz `Triangle`.

W takich przypadkach można dokonać porównania, ponieważ każdy obiekt klasy pochodzącej od `Shape` powinien posiadać metodę `area` (porównanie sugeruje, że prostokąt jest większy od koła, jeśli jego pole jest większe od pola koła). Jednakże jeśli zależy Ci jedynie na tym, aby obiekt posiadał daną metodę, to istnieje jeszcze bardziej liberalny sposób walidacji.

```
ca: Wiersz polecenia
>> puts !maple.enough?(439.50)
NoMethodError: undefined method `total' for 439.5:Float
  from 7.21.rb:29:in `enough?'
  from 7.21.rb:70
  from :0
>> oak += 35
NoMethodError: undefined method `boards' for 35:Fixnum
  from 7.21.rb:34:in `+'
  from 7.21.rb:71
  from :0
>>
```

Rysunek 7.21. Różne rodzaje błędów, które mogą wystąpić, jeśli metody i operatory będą wywoływane z niewłaściwymi typami lub operandami

Na przykład założmy, że chcesz stworzyć klasę, która będzie generowała tablice w języku HTML. Klasa taka może zamieniać tablicę na tablicę w HTML-u. Ale dlaczego nie mogłaby zamieniać hasza albo zakresu? Tak naprawdę dlaczego nie mogłaby zamieniać obiektu typu `GroceryList`, który również jest listą wartości? To, czego taka klasa naprawdę wymaga, może być dowolnym obiektem posiadającym iterator `each`. Aby sprawdzić taki warunek, użyj metody `respond_to?`:

```
class HtmlTable
  # Kod klasy.
  def make_rows(data)
    return nil unless data.respond_to?('each')
    data.each { # blok kodu
    end
  end
end
```

Taki sposób tworzenia kodu, w którym większą wagę przywiązuje się do tego, co obiekt może robić, niż do tego, jakiego jest typu, znany jest pod nazwą *kacze typowanie* (duck typing). Nazwa pochodzi od powiedzenia: „Jeśli chodzi jak kaczką i kwacze jak kaczką, nazwałbym to kaczką” („If it walks like a duck and quacks like a duck, I would call it a duck” — słowa przypisywane amerykańskiemu poecie i pisarzowi, Jamesowi Whitcombowi Riley’emu). Gdy stosuje się takie podejście przy tworzeniu klasy `HtmlTable`, korzyścią jest to, iż działałaby z obiektem dowolnego typu, byle tylko posiadał implementację metody `each`. Dotyczy to również klas utworzonych dawno po napisaniu klasy `HtmlTable`.

Wsparcie dla duck typing, a nawet preferowanie takiego tworzenia kodu jest jedną z wielu cech, które odróżniają Ruby od innych języków programowania. Odzwierciedla to część filozofii języka Ruby: to, co obiekt **może zrobić**, jest ważniejsze od tego, czym **jest**.

Aby wypróbować to podejście, stwórzmy nowe wersje klas `Rectangle` oraz `Circle` i zobaczmy, co można z nimi zrobić.

Aby wykorzystać duck typing:

1. Utwórz nowy skrypt w edytorze lub IDE (listing 7.3):

```
# Skrypt 7.3 - ducks.rb
```

2. Rozpocznij definiowanie klasy Rectangle:

```
class Rectangle
  attr_reader :height, :width
  def initialize(h, w)
    @height, @width = h, w
  end
  def area
    @height * @width
  end
end
```

Większość definicji klasy pochodzi z poprzednich przykładów. Aby zaoszczędzić miejsce, nie będę implementował metod `perimeter` i `is_square?`.

3. Zdefiniuj metodę `+`:

```
def +(rh)
  return nil unless rh.instance_of? Rectangle
  @height += rh.height
  @width += rh.width
  self
end
```

Dodawanie będzie wykonywane jedynie dla dwóch obiektów typu `Rectangle`. Wewnątrz tej metody pierwszy wiersz zwraca `nil`, co powoduje wyjście z metody, jeśli otrzymany argument — prawy operand — nie jest obiektem typu `Rectangle`. Jeśli warunek `rh.instance_of?` zwróci `true`, nastąpi odpowiednio dodawanie obu wysokości i obu szerokości. Następnie zostanie zwrócony bieżący obiekt.

Listing 7.3. Skrypt, który korzysta z metod `instance_of?` oraz `responds_to?` w celu kontroli, jakie typy obiektów mogą być ze sobą używane

```
Listing
1 # Skrypt 7.3 - ducks.rb
2
3 # Bardzo prosta definicja klasy Rectangle:
4 class Rectangle
5
6   attr_reader :height, :width
7
8   def initialize(h, w)
9     @height, @width = h, w
10  end
11
12  def area
13    @height * @width
14  end
15
16  # + powinien działać jedynie z innymi
17  # obiektami Rectangle!
18  def +(rh)
19    return nil unless rh.instance_of?
20    ↪ Rectangle
21    @height += rh.height
22    @width += rh.width
23    self
24  end
25
26  # Porównywać można z każdym obiektem, który
27  # posiada metodę „area”.
28  def <=>(rh)
29    return nil unless
30    ↪ rh.respond_to?('area')
31    if (self.area < rh.area) then -1
32    elsif (self.area > rh.area) then 1
33    else 0 end
34  end
35 end # Koniec definicji klasy Rectangle.
36
37 # Prosta definicja klasy Circle:
38 class Circle
39   attr_reader :radius
40   def initialize(r)
41     @radius = r
42   end
43   def area
44     Math::PI * @radius**2
45   end
46 end # Koniec definicji klasy Circle.
47
```

Listing 7.3. Skrypt, który korzysta z metod `instance_of?` oraz `responds_to?` w celu kontroli, jakie typy obiektów mogą być ze sobą używane — ciąg dalszy

```
Listing
46 # Utwórz obiekty:
47 r = Rectangle.new(32, 56)
48 c = Circle.new(8.4)
49
50 # Wyświetl informację o nich:
51 puts "Pole r wynosi #{r.area}. Pole c
    ↳wynosi #{c.area}."
52
53 # Wyświetl wynik dodawania:
54 print "Wynik dodawania r + c: "
55 puts r + c
56
57 # Wyświetl wynik porównania:
58 print "Porównanie r z c: "
59 puts r <=> c
```

4. Zdefiniuj metodę `<=>` i zakończ definicję klasy:

```
def <=>(rh)
  return nil unless rh.respond_to?('area')
  if (self.area < rh.area) then -1
  elsif (self.area > rh.area) then 1
  else 0 end
end
```

Ogólny operator porównania korzysta z kaczego typowanie do porównywania obiektów klasy `Rectangle` z dowolnymi obiektami posiadającymi metodę `area`. Pierwsza linia metody kończy jej działanie, jeśli nie jest spełniony warunek. Następnie, w zależności od wyniku porównania, zwracana jest wartość `-1`, `1` lub `0`.

5. Zdefiniuj klasę `Circle`:

```
class Circle
  attr_reader :radius
  def initialize(r)
    @radius = r
  end
  def area
    Math::PI * @radius**2
  end
end
```

Jest to podstawowa wersja klasy `Circle`, zawierająca jedynie metody `initialize` i `area`. Jeśli chcesz, możesz również zdefiniować operator `<=>`, dokładnie tak, jak jest to zrobione w klasie `Rectangle`, w ten sposób obiekty typu `Circle` mogą być lewymi operandami porównania.

6. Utwórz obiekty obu klas:

```
r = Rectangle.new(32, 56)
c = Circle.new(8.4)
```

7. Wyświetl pole obu figur:

```
puts "Pole r wynosi #{r.area}. Pole c wynosi
    ↳#{c.area}."
```

8. Wyświetl wynik dodawania obu figur:

```
print "Wynik dodawania r + c: "
puts r + c
```

Ponieważ klasa `Rectangle` wymaga, aby prawy operand również był typu `Rectangle`, wynikiem dodawania będzie `nil` (wartość zwrócona przez metodę).

9. Wyświetl wynik porównania obu figur:

```
print "Porównanie r z c: "
puts r <=> c
```

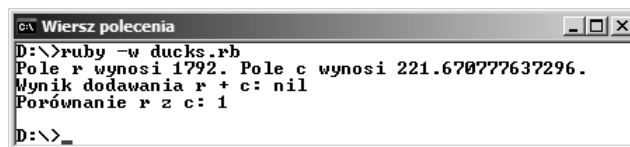
Dzięki korzystaniu z kaczego typowania można porównać obie figury.

10. Zapisz i uruchom skrypt (rysunek 7.22).**Wskazówki**

- Innym podejściem do definiowania metod w klasach jest przyjmowanie, że wszystkie dane otrzymane z wejścia są poprawne (to znaczy zakładanie, że wszystko na wejściu jest kaczką), a następnie zgłaszanie wyjątków w przypadku niepoprawnych danych wejściowych. Podejście to jest opisane w rozdziale 11., „Debugowanie i obsługa błędów”.
- Ponieważ w klasie `Circle` nie zdefiniowaliśmy operatora `<=>`, jeśli spróbujemy takiego porównania, wystąpi błąd:

```
c <=> r
```

Rozwiązaniem będzie zdefiniowanie operatora `<=>` w klasie `Circle` lub metody `coerce` w klasie `Rectangle`.



```
G:\ Wiersz polecenia
D:\>ruby -w ducks.rb
Pole r wynosi 1792. Pole c wynosi 221.670777637296.
Wynik dodawania r + c: nil
Porównanie r z c: 1
D:\>_
```

Rysunek 7.22. *Poprzez obserwację typów używanych obiektów skrypty mogą być bardziej lub mniej restrykcyjne, jeśli chodzi o dozwolone operacje na obiektach*