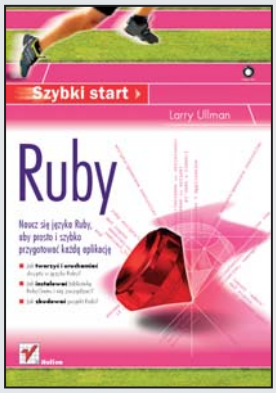
Ruby. Szybki start



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 niestandaryzowany, 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!

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

Spis treści

7 Klasy

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 są 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 **atrybutami** 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:

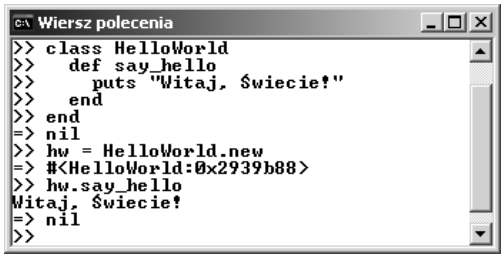
```
rubyvqs = Book.new('Ruby: VQS', 'Larry Ullman', '978-0-321-55385-0')
```

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
```



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):

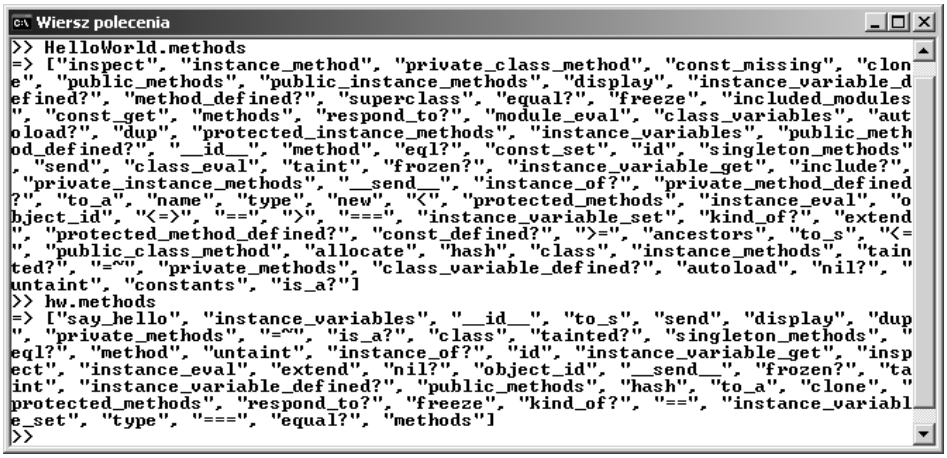
```
hw.say_hello
```

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").



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:

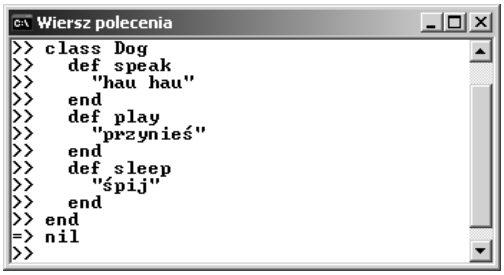
```
def speak
  "hau"
end
def play
  "przynieś"
end
def sleep
  "śpij"
end
```

   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).



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*).

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
```

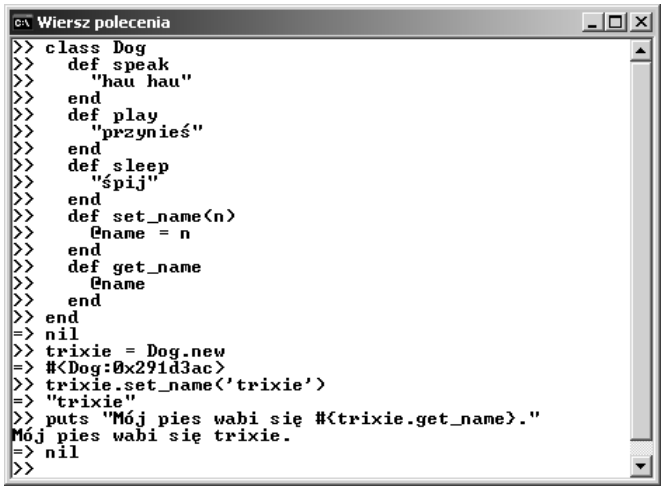
Taka wersja klasy będzie używana w następujący sposób (rysunek 7.5):

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

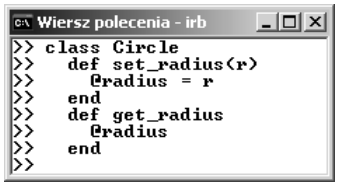
Oczywiście każda metoda klasy może korzystać ze zmiennej instancji:

```
def sleep
  "#@name śpi"
end
```

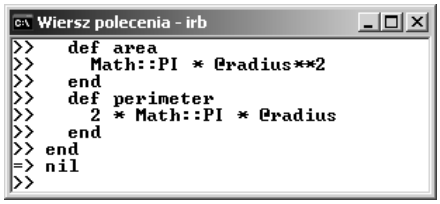
W metodzie `sleep` pokazano nowy przykład **interpolacji** (wstawienie wewnątrz ciągu znaków wartości zwróconej w wyniku polecenia). Aby wstawić zwykłe zmienne, musisz użyć `#{nazwa_zmiennej}`, ale by wstawić wartość zmiennej instancji, możesz użyć `#@nazwa_zmiennej`.



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



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



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:

```
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):

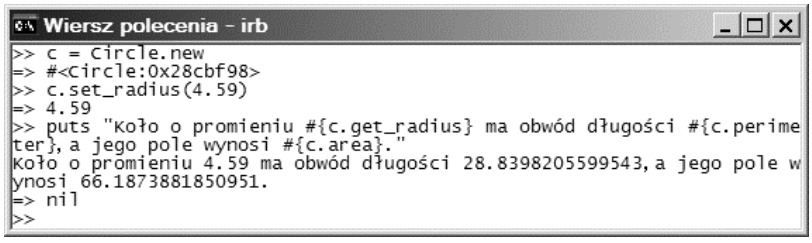
```
puts "Koło o promieniu #{c.get_radius} ma
↪obwód długości #{c.perimeter}, a jego pole
↪wynosi #{c.area}."
```

Trzy metody obiektu są wywoływane, żeby wyświetlić odpowiednie wartości.

Wskazówki

- **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 # Źle!
  def set_name(n)
    @name = n
```



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_accesor`, aby wskazać zmienne, dla których powinny istnieć metody `setter` i `getter`.

```
class NazwaKlasy
  attr_accesor :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 "Moj pies, #{d.name}, to #{d.gender} rasy
#{d.breed}."
```

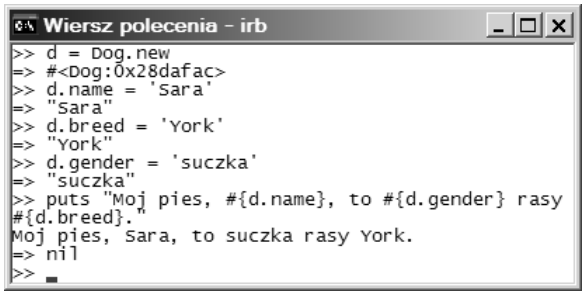
Wiersze te pokazują, jak łatwo możesz instancji przypisać zmiennym wartości (to znaczy **ustawić je**) oraz uzyskać ich wartości (to znaczy **pobrać je**, tak jak w powyższej instrukcji `puts`). Jeśli użyjesz słowa kluczowego `attr_reader`, tylko metody `getter` zostaną utworzone, oznacza to, że możesz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



Rysunek 7.9. *Dzięki użyciu attr_accesor 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):

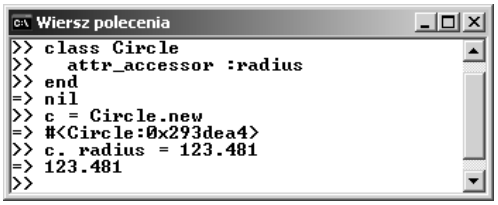
```
c = Circle.new
c.radius = 123.481
```

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

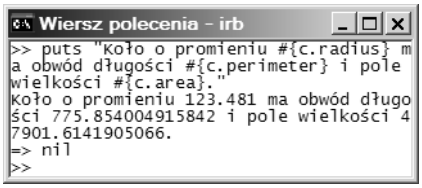
3. Wyświetl wymiary koła (rysunek 7.11):

```
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`.



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



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).

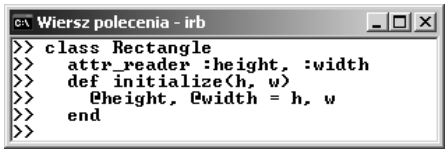
```
class NazwaKlasy
  def initialize
    # zrób cokolwiek
  end
end
```

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}."
```



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`).

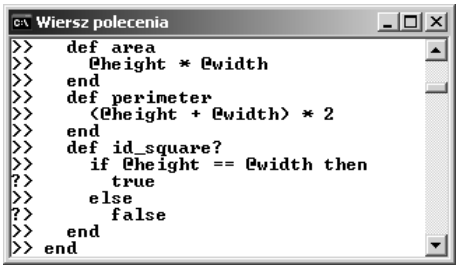
5. Zakończ definicję klasy:

```
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`.



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."
```

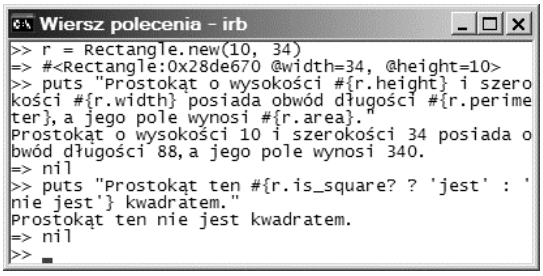
Pierwsze polecenie `puts` pokazuje, jak można wykorzystać automatycznie stworzone metody do pobrania wymiarów prostokąta. Polecenie to wywołuje również metody `perimeter` i `area`.

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”.

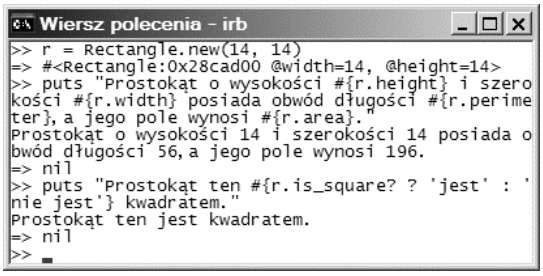
8. Utwórz nowy prostokąt i przepisz wiersze z kroku 7. (rysunek 7.15):

```
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`.



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



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

Definiowanie operatorów

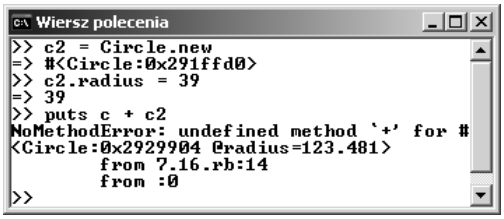
Tak jak obiekty posiadają metody służące do wykonywania różnych zadań, tak samo mogą wykorzystywać operatory — arytmetyczne, przypisania i logiczne — których sposób działania możesz zdefiniować. Aby to zrobić, definiujesz metodę, której nazwa jest operatorem: `+`, `-`, `==` itd. Operatory takie wymagają 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
```

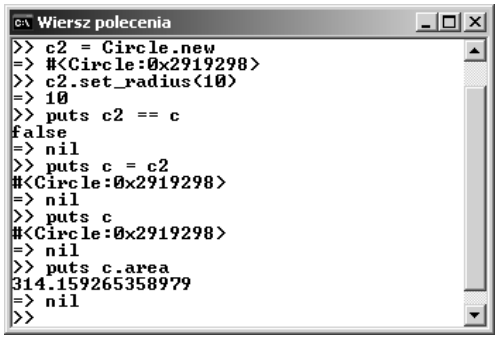
To podstawowa składnia implementacji operatora mnożenia dla klasy. Od Ciebie, twórcy klasy, zależy, co metoda powinna robić w momencie użycia operatora. Możesz zdecydować się na pozostawienie operatora niezdefiniowanego, co spowoduje wystąpienie błędu, jeśli ktoś użyje go z danym obiektem (rysunek 7.16).

Jako przykład weźmy dodawanie dwóch obiektów typu `Rectangle`:

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



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



Rysunek 7.17. *Program, w którym wykorzystano dwa obiekty Circle w celu demonstracji działania operatorów przypisania i przyrównania 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 przyrównania (`==`) (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: stolarkę[1]. 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.

Tabela 7.1. *Powierzchnia całkowita*

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

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 ¾, 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.

[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] *
        ↪b[:length]) }
23      t
24    end
25
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 przyrównania:
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 definicje klasy `Wood`:

   ```
   class Wood
     attr_reader :boards
     def initialize(*args)
       @boards = args
     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 umożliwiająca 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!"
```

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 sama 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
```

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 wywołaj metodę, która zmienia bieżący obiekt:

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

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 wiążą się obowiązki 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`.

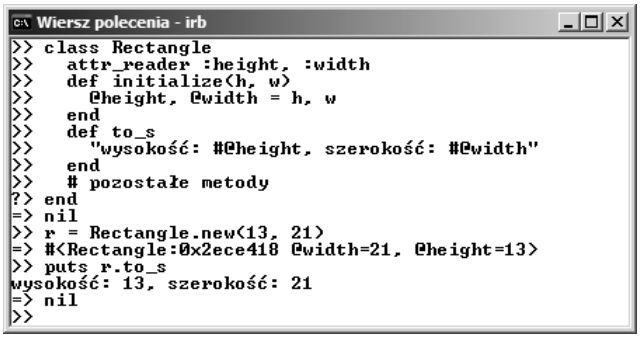
Aby utworzyć własną metodę `to_s`, po prostu zdefiniuj metodę o takiej nazwie. Można się spodziewać, że metoda taka może zwracać wartości zmiennych klas:

```
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.



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
        ↪+= "#{k}: #{v}, "}
27      str.slice(0, str.length - 2).to_s
28    end
29
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
```

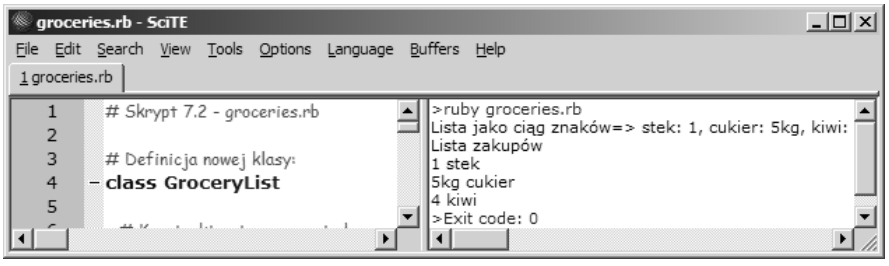
Aby to zrobić, możesz skorzystać z metody `to_s`. Jako ciąg znaków może zostać wyświetlona, zamieniona w tablicę, można ją przeszukiwać, można z nią zrobić wszystko to, co robi się ze standardowym obiektem klasy `String`.

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).



Rysunek 7.20. *Wynik wykonania skryptu groceries.rb w edytorze SciTE pod systemem Windows*

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:

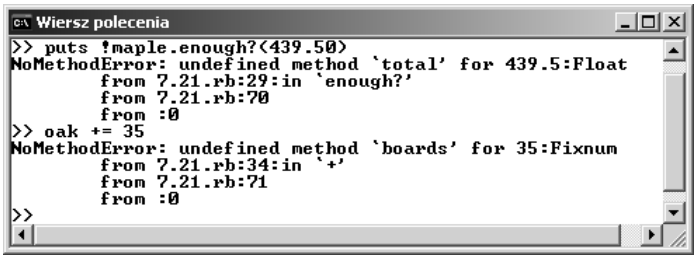
```
class ThisClass
  def -(rh)
    return nil unless rh.instance_of?
    ↪ThisClass
    # W porządku, wykonaj odejmowanie.
  end
end
```

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.



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

Na przykład załóżmy, 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
```

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 nazwa ***kacze typowanie*** (duck typing). Nazwa pochodzi od powiedzenia: „Jeśli chodzi jak kaczka i kwacze jak kaczka, 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
```

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
      # obiektami Rectangle!
17    def +(rh)
18      return nil unless rh.instance_of?
        ↪Rectangle
19      @height += rh.height
20      @width += rh.width
21      self
22  end
23
24    # Porównywać można z każdym obiektem, który
25    # posiada metodę „area".
26    def <=>(rh)
27      return nil unless
        ↪rh.respond_to?('area')
28      if (self.area < rh.area) then -1
29      elsif (self.area > rh.area) then 1
30      else 0 end
31    end
32
33  end # Koniec definicji klasy Rectangle.
34
35  # Prosta definicja klasy Circle:
36  class Circle
37    attr_reader :radius
38    def initialize(r)
39      @radius = r
40    end
41    def area
42      Math::PI * @radius**2
43    end
44  end # Koniec definicji klasy Circle.
45
```

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
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.

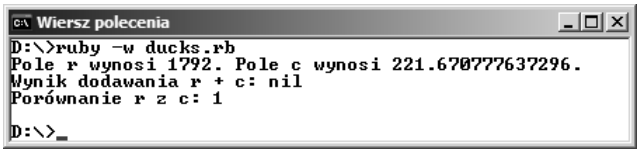
10. Zapisz i uruchom skrypt (rysunek 7.22).

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., „Debuggowanie 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`.



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