19. Obiektowość

1 Kacze typowanie

Skąd interpreter wie, jakiego typu są np. przekazywane do metody argumenty? Tak naprawdę wcale nie musi wiedzieć, do poprawnego działania programu wystarczy, że przekazywany obiekt będzie posiadał wymagane metody lub atrybuty. Jeżeli nie będzie miał, dopiero wówczas zostanie zgłoszony błąd — podczas próby wywołania nieistniejącej metody.

Tzw. kacze typowanie (ang. *duck typing*) jest to rodzaj dynamicznego typowania (ang. *dynamic typing*), w którym rozpoznanie typu obiektu następuje na podstawie posiadanych przez niego metod i właściwości (atrybutów):

> *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*

2 Klasy

W języku Ruby klasę definiuje się za pomocą słowa kluczowego `class`:

```ruby
class MyClass
  def initialize(arg)   # metoda wywolywana podczas tworzenia obiektu
    @attr = arg         # zmienne poprzedzone @ to atrybuty
  end

  def my_method
    puts "Attribute value is #{@attr}"
  end
end
```

Atrybutem jest każda zmienna poprzedzona znakiem `@` wewnątrz dowolnej metody (w zależności od tego, jakie metody zostały wywołane, obiekt danej klasy może mieć różne atrybuty podczas swojego życia). Atrybuty nazywane są składowymi lub zmiennymi instancji (ang. *instance variables*).

Podczas tworzenia obiektu wywoływana jest metoda `initialize` z odpowiednią arnością:

```ruby
obj = MyClass.new(123)
obj.my_method
```

Zadanie 01 Napisz klasę `Point3D`, która będzie reprezentować punkt w przestrzeni trójwymiarowej. Trzy atrybuty `@x`, `@y`, `@z` powinny być inicjalizowane w metodzie `initialize` przyjmującej trzy argumenty.

Stwórz obiekt klasy `Point3D`. Czy można odczytać składowe `x`, `y` lub `z`?

2.1 Modyfikatory dostępu

W Rubym domyślnie wszystkie atrybuty są prywatne. Zamiast definiować metody pomocnicze w postaci `get_attr` oraz `set_attr(x)` dostępne są makra, które tworzą odpowiednie metody:

- `attr_reader` — tylko do odczytu,
- `attr_writer` — tylko do zapisu,
- `attr_accessor` — swobodnego dostępu.

```
class MyClass
  attr_accessor :attr   # atrybut @attr bedzie mozna odczytywac
                        # i zapisywac z poziomu obiektu
  ...
end
```

```
obj.attr = 123
puts obj.attr
```

Ponadto, dla metod istnieją standardowe modyfikatory zakresu dostępu (`private`, `public`, `protected`), które działają do końca definicji klasy.

Zadanie 02 Zmodyfikuj klasę `Point3D` tak, aby wartości atrybutów można było odczytywać i modyfikować. Przetestuj odpowiednim fragmentem kodu.

Zadanie 03 Do klasy `Point3D` dopisz publiczną metodę `set(x, y, z)` ustawiającą wartości atrybutów.

Jakiego typu mogą być argumenty wykorzystywane do utworzenia obiektu tej klasy? Przetestuj różne typy argumentów, np. liczby, łańcuchy znaków, tablice.

2.2 Składowe klasy

Zmienne wewnątrz definicji klasy poprzedzone `@@` są zmiennymi klasowymi (ang. *class variables*) — mogą być rozumiane jako atrybuty statyczne, tzn. obiekty tej samej klasy współdzielą ich wartość.

Zadanie 04 Do klasy `Point3D` dopisz zmienną klasową `instances`, która będzie zliczała wszystkie utworzone obiekty (w metodzie `#initialize`). Przetestuj działanie powyższego mechanizmu dodając do klasy poniższą metodę:

```
def self.instances  # self odnosi sie do klasy nie obiektu, dzieki
  @@instances       # czemu mozliwe bedzie wywolanie Point3D.instances
end
```

3 Metody

Argumenty metod nie wymagają podania ich typu, dlatego nie ma możliwości przeciążania metod. Aby zdefiniować metodę o różnej arności argumentów należy ustawić domyślne wartości argumentów:

```
def foo(bar, baz="quux")
  @bar = bar
  @baz = baz
  "bar = #{@bar} baz = #{@baz}"
end
```

```
foo(123, 456)           # => "bar = 123 baz = 456"
foo(123)                # => "bar = 123 baz = quux"
```

Innym, często wykorzystywanym sposobem jest wykorzystanie tablic asocjacyjnych:

```
def foo(bar, options={})
  @bar = bar
  @baz = options[:baz] || "quux"
end
```

```
foo(123, :baz => 456)   # => "bar = 123 baz = 456"
foo(123)                # => "bar = 123 baz = quux"
```

Zadanie 05 Popraw `#initialize` w klasie `Point3D` w taki sposób, aby możliwe było utworzenie obiektu bez podawania argumentów, tylko z jednym argumentem lub tylko z dwoma. Nie podane wartości `x`, `y`, `z` powinny przyjmować domyślną wartość 0.0.

W języku Ruby nazwy metod mogą kończyć się znakami `!` oraz `?`. Przyjęło się, że metody kończące się wykrzyknikiem to „destrukcyjne" (modyfikujące obiekt w miejscu) wersje metod o tej samej nazwie, ale bez wykrzyknika na końcu.

Zadanie 06 Do klasy `Point3D` dodaj metodę `move(x,y,z)`, która zwróci nowy punkt przesunięty o podane współrzędne. Dopisz również metodę `move!(x,y,z)`, która zmodyfikuje obiekt, na którym metoda zostanie wywołana.

Zadanie 07 Do klasy `Point3D` dodaj metodę `zero?` zwracającą prawdę, jeżeli punkt leży we współrzędnych zerowych.

Aliasy pozwalają na wprowadzenie nowej nazwy dla już zdefiniowanej metody. Pozwalają również zachować oryginalną wersję metody przed jej zredefiniowaniem.

```
def foo
  "foo!"
end
alias :bar :foo

foo()   # => "foo!"
bar()   # => "foo!"
```

Zadanie 08 Do klasy `Point3D` dodaj metodę `euclidean_distance(point)`, która zwróci odległość euklidesową między dwoma punktami.

Zadanie 09 W klasie `Point3D` zdefiniuj alias dla `#euclidean_distance` o nazwie `dist`.

Zadanie domowe 10 (1 pkt) Stwórz klasę `Path` (reprezentującą ścieżkę), która będzie spełniała poniższe wymagania:

- ścieżka będzie reprezentowana przez tablicę punktów w atrybucie `points` możliwym do odczytu,
- maksymalna długość ścieżki będzie przechowywana w atrybucie `max_length` — wartość do odczytu,
- domyślna maksymalna długość ścieżki będzie przechowywana w stałej `MAX_LENGTH` przechowywanej wewnątrz klasy,
- możliwość tworzenia obiektu bez podawania żadnego argumentu, z jednym argumentem (maksymalna długość) lub z dwoma (maksymalna długość i początkowe punkty ścieżki),
- poprzez publiczną metodę `#add_point` będzie można dodawać punkty do ścieżki (z ograniczeniem na maksymalną długość),
- prywatna metoda `pairs` dla tablicy o elementach $i_1, i_2, i_3, \ldots, i_n$ zwróci tablicę dwuelementowych tablic w postaci $(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n)$
- metoda `length` zwróci łączną długość ścieżki (wykorzystaj metodę `pairs`).

Przetestuj działanie klasy i jej metod.

4 Pełna obiektowość

Z faktu, że można definiować nowe funkcje poza klasą, wydawać by się mogło, że język Ruby oferuje możliwość programowania bez paradygmatu obiektowego. Nic bardziej mylnego: wszystkie te „funkcje" są automatycznie wykonywane w kontekście klasy `Kernel`, zatem należą do klasy.

Nawet operatory matematyczne są metodami opatrzonymi lukrem składniowym (patrz poniżej), a liczby (literały) są obiektami klas.

4.1 Lukier składniowy

Lukier składniowy (ang. *syntactic sugar*) to cecha składni języka, którą można wyeliminować poprzez proste przekształcenia składniowe, wprowadzana najczęściej dla wygody programisty.

W języku C przykładem lukru składniowego jest dostęp do elementów tablicy poprzez zapis `tab[i]`, który jest tłumaczony na `*(tab+i)`.

W języku Ruby lukrem składniowym są np.:

- opcjonalność nawiasów przy wywoływaniu lub definiowaniu metod,
- operatory matematyczne, zapis `123 + 456` jest jednoznaczny z `123.+(456)`,
- metody, których nazwy kończą się znakiem równości, dopuszczają wstawienie znaków spacji przed tym znakiem (np. metody wygenerowane przez `attr_writer`), zatem `obj.attr = "foo"` to tak naprawdę `obj.attr=("foo")`,
- odwołanie się do elementów tablicy `tab[i]` to również wywołanie metody `tab.[](i)`.

Zadanie 11 dodatkowe (1 pkt) Rozszerz klasę `Numeric` o możliwość intuicyjnego operowania na czasie przy pomocy metod `weeks`, `days`, `hours`, `minutes`, `seconds` (i metod konwertujących do odpowiednich jednostek czasu):

```
1 time = 1.days - 2.0.hours + 30.minutes
2 puts time.to_hours    # => 22.5
3 puts time.to_days     # => 0.9375
```