Классы, объекты и переменные

Андрей Васильев

2019

Задача

Мы управляем магазином поддержанных книг. Каждую неделю проводится инвентаризация. Работники сканируют бар-коды на книгах и сохраняют их в CSV-списки.

Пример файла

```
"Date","ISBN","Price"
"2013-04-12","978-1-9343561-0-4",39.45
"2013-04-13","978-1-9343561-6-6",45.67
"2013-04-14","978-1-9343560-7-4",36.95
```

Задачи системы

- Выяснить количество книг каждого наименования
- Общую стоимость всех книг

Идентификация ключевых элементов

При проектировании решения в объектно-ориентированном подходе сначала необходимо идентифицировать элементы

Для нашего случая выделим следующие сущности

- Сущность, описывающая одну книгу, `BookInStock`
- Коллекция книг, содержащихся в наличии, `Books`

Описание классов в Ruby

Для описания классов используется конструкция

```ruby
class ClassName
  # Содержимое класса
end
```

Создание объектов в Ruby

Для создания объектов класса в Ruby используется метод new

```
object_one = ClassName.new
object_two = ClassName.new
```

- Метод создаёт новый объект и возвращает ссылку на него
- Метод принадлежит классу и вызывается на классе
- Метод `::new` есть у каждого класса

В примере выше мы создали 2 объекта класса `ClassName` и записали их в переменные `object_one` и `object_two`

Методу можно передать данные, которые будут использованы при инициализации объекта:

```
book = BookInStock.new('978-1-9343560-7-4', 10.2)
```

Инициализация объектов в Ruby

После создания каждого объекта Ruby инициализирует объект, вызывая метод `initiailize` и передавая параметры из `new`

```
class BookInStock
  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
  end
end
```

- Метод используется для установки значения переменным экземпляра класса, описывающие состояние объекта
- Переменные экземпляра начинаются с символа “@”
- Установленные значения должны быть корректны и позволять вызывать любые методы класса в любом порядке
- Метод может проверять переданные в него данные

«Печать» объектов

- Методы `p` и `pp` показывают внутреннее состояние объекта
- Метод `puts` пытается преобразовать объект к строке
- Стандартный способ представления: имя класса и уникальный идентификатор

```
#<Object:0x000056409cbcf7e0>
```

Преобразование к строке

При преобразовании объекта к строке вызывается метод `to_s`, который можно переопределить для своего класса

- Метод `to_s` не принимает аргументов
- Метод `to_s` должен вернуть строку

Для работы методов `p` иногда разумно реализовать метод `inspect`, который имеет такую же семантику, как и `to_s`

Атрибуты объекта

- Все переменные экземпляра являются приватными
- Для доступа к значениям переменных экземпляра и изменения их состояния определяются методы
- Такие методы определяют внешнее состояние объекта, видимое другими классами. Они называются атрибутами

```ruby
def isbn
  @isbn
end
```

Метод `attr_reader` создаёт методы для чтения значения переменных экземпляра класса

```ruby
attr_reader :isbn, :price
```

- Символами описываем имена переменных экземпляра
- Создаёт методы, а не меняет видимость переменных

Изменение атрибутов

Обычным для объектно-ориентированных языков способом изменения атрибута является создание специального метода

```
public void setPrice(double newPrice) {
    price = newPrice
}
```

- В Ruby принято оформлять взаимодействие с атрибутами, как с обычными переменными
- Для этого метод, устанавливающий значение атрибута, имеет на конце символ =

```
def isbn=(isbn)
  @isbn = isbn
end
book.isbn = '978-1-9343561-0-4'
```

Методы для создания атрибутов

Метод `attr_accessor` создаёт методы для чтения и записи данных в переменные экземпляра

```
attr_accessor :isbn, :price
```

- В качестве аргументов методу передаются символы с именами переменных экземпляра класса
- При его использовании класс предоставляет полный доступ для внешних объектов, т.е. теряет контроль над своими данными

Метод `attr_writer` создаёт метод для записи данных. Зачастую не используется, так как сложно представить ситуацию, когда надо записывать данные, но не считывать их.

Виртуальные атрибуты

В Ruby вы всегда взаимодействуете с методами, а не с переменными экземпляра. Это позволяет зафиксировать интерфейс объекта и легко менять его реализацию в будущем

Можно описать атрибут, который выполняет более сложные действия по сравнению с чтением и записью

Виртуальный атрибут - стоимость книги в копейках

```ruby
def price_in_copeks
  Integer(@price * 100 + 0.5)
end
def price_in_copeks=(copeks)
  @price = copeks / 100.0
end
```

- Чтение значения: `book.price_in_copeks`
- Присваивание значения: `book.price_in_copeks = 15`

Взаимодействие между классами

Во время решения реальных задач с помощью классов мы описываем не только реальные объекты, но также и технические элементы, необходимые для достижения цели

В нашем приложении необходимо обрабатывать информацию о множестве книг, которая записана в CSV-файлы

Класс Books - чтение и обработка набора данных

Определим интерфейс класса, который мы хотим реализовать

- Чтение информации из нескольких CVS-файлов
  - `read_in_csv_data`
- Вычисление нужных характеристик
  - `total_value_in_stock`
  - `number_of_each_isbn`

Чтение данных из CSV-файла

Класс Books должен считывать данные из нескольких CSV-файлов, которые собираются разными устройствами

- Стандартная поставка включает в себя библиотеку csv
- Библиотека предоставляет класс CSV, позволяющий читать и записывать CSV-документы
- Для чтения можно воспользоваться методом foreach

```
CSV.foreach('file.csv', headers:true) do |row|
  puts "#{row['ISBN']}, #{row['Price']}"
end
```

```
"Date","ISBN","Price"
"2013-04-12","978-1-9343561-0-4",39.45
```

Хранение информации о книгах

Класс `Books` должен сохранять информацию о всех считанных книгах. Для её хранения будем использовать массив.

- Пустые массивы обычно создаются с помощью литерала `[]`
- Метод `<<` добавляет объект в конец массива
- Метод `push` добавляет один или несколько объектов в конец массива

Альтернативные имена методов

Ruby позволяет разработчикам определить альтернативные имена для публичных методов.
В Ruby 2.5 ввели альтернативное название для метода `push` - `append`.

Структурирование файлов приложения

Обычно один исходный файл на языке Ruby содержит один класс или один модуль, что

- Позволяет легко находить файл с нужным классом
- Облегчает рефакторинг исходного кода
- Облегчает модульное тестирование

Желательно разделять модули, ответственные за взаимодействие с внешним миром (пользователи, файлы) от модулей, которые реализуют обработку данных

Подключение внешних файлов

Для подключения других файлов используются методы

- `require` для подключения внешних библиотек
- `require_relative` для подключения собственных файлов по относительному пути

Относительный путь строится относительно текущего файла

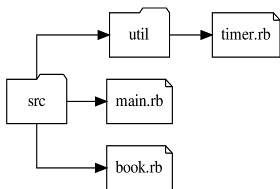


Рис.: Пример структуры ФС

```
require_relative 'book'
require_relative 'util/timer'
require_relative '../book'
```

- Подключение файла `book` из файла `main`
- Подключение файла `timer` из файла `main`
- Подключение файла `book` из файла `timer`

Контроль доступа к методам класса

Ruby предоставляет 3 уровня контроля доступа к методам

- Публичные (`public`) методы могут быть вызваны любым объектом
  - По умолчанию все методы кроме `initialize` являются публичными
- Защищённые (`protected`) методы могут быть вызваны внутри дерева наследования, включая другие объекты данного класса
- Приватные (`private`) методы могут быть вызваны только лишь внутри данного класса

Отличия от знакомых вам языков программирования

- Приватные методы нельзя вызывать из других объектов этого же класса
- Контроль за вызовом методов осуществляется во время выполнения приложения, а не во время

Указание контроля доступа

Для указания контроля доступа используются методы `public`, `protected`, `private`

Указание уровня доступа для секции

```
class MyClass
  private
  def method_one
  end
  def method_two
  end
end
```

Указание уровня доступа для конкретных методов

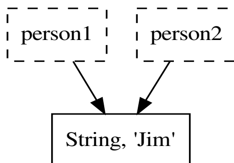
```
class MyClass
  pritave :method_one, :method_two
end
```

Переменные

- Основная задача переменных - хранение ссылки на объект
- Переменные *не являются* объектами

```
person1 = 'Tim'
person2 = person1
person1[0] = 'J' # => 'Jim'
person2[0] # => 'Jim'
```

- Оператор присваивания записывает ссылку на объект
- Все изменения объекта доступны из всех переменных, которые содержат в себе ссылку на объект



Предотвращение непродуманных изменений

Использование явного копирования

```
person1 = 'Tim'
person2 = person1.dup
person1[0] = 'J'
```

- Объект `person2` содержит копию данных
- При изменении объекта `person2` объект `person1` не изменяется

Запрет всех последующих изменений

```
person1 = 'Tim'
person2 = person1
person1.freeze
person2[0] = 'J' # => Ошибка изменения константы
```

Проектирование объекта неизменяемым

При применении данной техники вместо изменения текущего объекта создаётся копия оригинального объекта

```
class Maslo
  attr_reader :weight # Только лишь чтение
  def initialize(weight)
    @weight = weight
  end
  def take(weight)
    new Maslo(@weight - weight) # Новый объект
  end
end
```

- Удобно для многопоточного программирования
- Внешний разработчик не может привести объект в некорректное состояние