MI-RUB Objects and Classes

Pavel Strnad
pavel.strnad@fel.cvut.cz

Dept. of Computer Science, FEE CTU Prague,
Karlovo nám. 13, 121 35
Praha, Czech Republic

MI-RUB, WS 2011/12

Contents

Objects

Objects in Ruby

- almost everything in Ruby is an object,
- classes are objects,
- class instances are objects.

Are instance variables objects?

Classes

Class names start with an upper case letter!

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

Classes vs. Instances

Classes vs. Instances

Classes are *type of things* in our program.
We can create instances of classes, *the things*.

Example

Orange (imaginary orange) is a class in our mind. Orange (concrete orange) in our hands is its instance. Fruit (abstract class) is a superclass of Orange.

Instantiation

```
class BookInStock
  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
  end
end
b1 = BookInStock.new("isbn1", 3)
p b1
```

Instantiation

Initialize

If we want to pass arguments to *new* we have to override method called *initialize*. *Initialize* is called immediately by *new* when instance is created.

Instance variables

Instance variables properties

- starts with @,
- don’t need to be declared,
- are declared in place,
- are not visible outside object - encapsulation.

Attributes

Attributes properties

- represents "outside" properties of an object,
- can be represented by instance variables,
- provides interface to internal state,
- are visible outside object,
- attributes are methods.

Attributes

```
class BookInStock
  @@count = 0
  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
  end
  def isbn
    @isbn
  end
  def isbn=(anISBN)
    @isbn=anISBN
  end
end
```

Attributes

```ruby
class BookInStock
  attr_reader :isbn
  attr_accessor :price
  def initialize(isbn, price)
    @isbn = isbn
    @price = Float(price)
  end
end
```

Instance methods

Instance methods

- Instance methods are pieces of a code which is executed within instance context.

Instance methods

self is a reference to a current instance.

```
class Transaction
  def initialize(account_a, account_b)
    @account_a = account_a
    @account_b = account_b
  end
  def debit(account, amount)
    account.balance = amount
  end
  def credit(account, amount)
    account.balance += amount
  end
  def transfer(amount)
    self.debit(@account_a, amount)
    self.credit(@account_b, amount)
  end
end
tr = Transaction.new(a1, a2)
tr.transfer(20)
```

Class methods

Class methods

- Class methods are like instance methods buth their context is the class.

```
class Test
  def self.name
    "Test"
  end
end
Test.name
```

Access Control

Access Control

When designing a class interface, it's important to consider just how much of your class you'll be exposing to the outside world. Allow too much access into your class, and you risk increasing the coupling in your application—users of your class will be tempted to rely on details of your class's implementation, rather than on its logical interface. The good news is that the only easy way to change an object's state in Ruby is by calling one of its methods.

Access Control

Ruby has three levels of protection:

Levels of protection

- **Public methods** can be called by anyone—no access control is enforced. Methods are public by default (except for initialize, which is always private).
- **Protected methods** can be invoked only by objects of the defining class and its subclasses. Access is kept within the family.
- **Private methods** cannot be called with an explicit receiver—the receiver is always the current object, also known as self. This means that private methods can be called only in the context of the current object; you can't invoke another object's private methods.

Access Control

```
class MyClass
    def method1 # default is 'public'
      #...
    end
  protected # subsequent methods will be 'protected'
    def method2 # will be 'protected'
      #...
    end
  private # subsequent methods will be 'private'
    def method3 # will be 'private'
      #...
    end
  public # subsequent methods will be 'public'
    def method4 # so this will be 'public'
      #...
    end
end
```

Access Control

```ruby
class MyClass
  def method1
  end
  # ... and so on
  public :method1, :method4
  protected :method2
  private :method3
end
```

Today's exercise

Problem

For a given list of adjacent vertices of a graph and a chosen vertex v write down in the Depth First Search (DFS) or Breadth First Search (BFS) order all the vertices from the connected component of the graph containing v. Assume that the number of vertices of the graph is at most 1000.

Requirement

Implement a graph and nodes as objects. Try not using array...

Today's exercise

Info

Whole problem specification at:
`http://www.spoj.pl/problems/TDBFS/`

Solution

Solutions are accepted to pavel.strnad@fel.cvut.cz on github due 9.10.2011 23:59.