OOP – Ruby 1

OPL – 2/66

Based heavily on Dan Grossman's CSE341: Programming Languages, Lecture 19, Introduction to Ruby and OOP

Outline

- The Ruby programming language
- Class definition
- Instance/class variables
- Visibility

Ruby language

- Next three classes use the Ruby language
  - http://www.ruby-lang.org/
- Excellent documentation available
- –http://ruby-doc.org/
- –http://www.ruby-lang.org/en/documentation/

Ruby

- Pure object-oriented: all values are objects (even numbers)
- Class-based: Every object has a class that determines behavior
  - Like Java, unlike Javascript
  - Mixins (not [old] Java interfaces nor C++ multiple inheritance)

- Dynamically typed
- Convenient reflection: Run-time inspection of objects
- Very dynamic: Can change classes during execution
- Blocks and libraries encourage lots of closure idioms
- Syntax, scoping rules, semantics of a "scripting language"
  - Variables "spring to life" on use
  - Very flexible arrays

Ruby

	dynamically typed	statically typed
functional	Racket	SML
object-oriented (OOP)	Ruby	Java

Racket also has classes and objects when you want them

- In Ruby everything uses them (at least implicitly)

Historical note: *Smalltalk* also a dynamically typed, class-based, pure OOP language with blocks and convenient reflection

- Smaller just-as-powerful language
- Ruby less simple, more “modern and useful”

Dynamically typed OOP helps identify OOP's essence by not having to discuss types

The rules of class-based OOP

In Ruby:

1. All values are references to *objects*
2. Objects communicate via *method calls*, also known as *messages*
3. Each object has its own (private) *state*
4. Every object is an instance of a *class*
5. An object’s class determines the object’s *behavior*
   - How it handles method calls
   - Class contains method definitions

Java/C#/etc. similar but do not follow (1) (e.g., numbers, **`null`**) and allow objects to have non-private state

Defining classes and methods

```
class Name
  def method_name1 method_args1
    expression1
  end
  def method_name2 method_args2
    expression2
  end
  …
end
```

- Define a class with methods as defined
- Method returns its last expression
  - Ruby also has explicit return statement
- Syntax note: Line breaks often required (else need more syntax), but indentation always only style

Creating and using an object

- **`ClassName.new`** creates a new object whose class is **`ClassName`**
- **`e.m`** evaluates **`e`** to an object and then calls its **`m`** method
  - Also known as “sends the **`m`** message”
  - Can also write **`e.m()`** with no space
- Methods can take arguments, called like **`e.m(e1,…,en)`**
  - Parentheses optional in some places, but recommended

Variables

- Methods can use local variables
  - Syntax: starts with letter
  - Scope is method body
- No declaring them, just assign to them anywhere in method body (!)
- Variables are mutable, **`x=e`**
- Variables also allowed at "top-level" or in REPL
- Contents of variables are always references to objects because all values are objects

Self

- **`self`** is a special keyword/variable in Ruby
  - (Same as **`this`** in Java/C#/C++)
- Refers to "the current object"
  - The object whose method is executing
- So call another method on "same object" with **`self.m(`**…**`)`**
  - Syntactic sugar: can just write **`m(`**…**`)`**
- Also can pass/return/store "the whole object" with just **`self`**

Objects have state

- An object's state persists
  - Can grow and change from time object is created
- State only directly accessible from object's methods
  - Can read, write, extend the state
  - Effects persist for next method call
- State consists of *instance variables* (also known as fields)
  - Syntax: starts with an `@`, e.g., **`@foo`**
  - "Spring into being" with assignment
    - So mis-spellings silently add new state (!)
  - Using one not in state not an error; produces **`nil`** object

Aliasing

- Creating an object returns a reference to a new object
  - Different state from every other object
- Variable assignment (e.g., **`x=y`**) creates an alias
  - Aliasing means same object means same state

Initialization

- A method named **`initialize`** is special
  - Is called on a new object before **`new`** returns
  - Arguments to **`new`** are passed on to **`initialize`**
  - Excellent for creating object invariants
  - (Like constructors in Java/C#/etc.)
- Usually good *style* to create instance variables in **`initialize`**
  - Just a convention
  - Unlike OOP languages that make “what fields an object has” a (fixed) part of the class definition
    - In Ruby, different instances of same class can have different instance variables

Class variables

- There is also state shared by the entire class
- Shared by (and only accessible to) all instances of the class
  - (Like Java static fields)
- Called *class variables*
  - Syntax: starts with an `@@`, e.g., **`@@foo`**
- Less common, but sometimes useful
  - And helps explain via contrast that each object has its own instance variables

Class constants and methods

- *Class constants*
  - Syntax: start with capital letter, e.g., **`Foo`**
  - Should not be mutated
  - Visible outside class **C** as **`C::Foo`** (unlike class variables)
- *Class methods* (cf. Java/C# static methods)
  - Syntax (in some class **C**):

    ```
    def self.method_name (args)
      …
    end
    ```
  - Use (of class method in class **C**):

    ```
    C.method_name(args)
    ```
  - Part of the class, not a particular instance of it

Who can access what

- We know “hiding things” is essential for modularity and abstraction
- OOP languages generally have various ways to hide (or not) instance variables, methods, classes, etc.
  - Ruby is no exception
- Some basic Ruby rules here as an example…

Object state is private

- In Ruby, object state is always private
  - Only an object's methods can access its instance variables
  - Not even another instance of the same class
  - So can write **`@foo`**, but not **`e.@foo`**

- To make object-state publicly visible, define "getters" / "setters"
  - Better/shorter style coming next

```
def get_foo
  @foo
end
def set_foo x
  @foo = x
end
```

Conventions and sugar

- Actually, for field @foo the convention is to name the methods

```
def foo
  @foo
end
```

```
def foo= x
  @foo = x
end
```

- Cute sugar: When using a method ending in =, can have space before the =

```
e.foo = 42
```

- Because defining getters/setters is so common, there is shorthand for it in class definitions
  - Define just getters: attr_reader :foo, :bar, …
  - Define getters and setters: attr_accessor :foo, :bar, …

- Despite sugar: getters/setters are just methods

Why private object state

- This is "more OOP" than public instance variables

- Can later change class implementation without changing clients
  - Like we did with ML modules that hid representation
  - And like we will soon do with subclasses

- Can have methods that "seem like" setters even if they are not

```
def celsius_temp= x
  @kelvin_temp = x + 273.15
end
```

- Can have an unrelated class that implements the same methods and use it with same clients
  - See later discussion of "duck typing"

Method visibility

- Three *visibilities* for methods in Ruby:
  - **private**: only available to object itself
  - **protected**: available only to code in the class or subclasses
  - **public**: available to all code

- Methods are **public** by default
  - Multiple ways to change a method's visibility
  - Here is one way…

Method visibilities

```
class Foo =
# by default methods public
   …
 protected
# now methods will be protected until
# next visibility keyword
   …
 public
   …

 private
   …
end
```

One detail

If **`m`** is private, then you can only call it via **`m`** or **`m(args)`**

- As usual, this is shorthand for **`self.m`** …
- But for private methods, only the shorthand is allowed

Pure OOP

- Ruby is fully committed to OOP:

Every value is a reference to an object

- Simpler, smaller semantics
- Can call methods on anything
  - May just get a dynamic "undefined method" error
- Almost everything is a method call
  - Example: **`3 + 4`**

Some examples

- Numbers have methods like **`+`**, **`abs`**, **`nonzero?`**, etc.
- **`nil`** is an object used as a "nothing" object
  - Like **`null`** in Java/C#/C++ except it is an object
  - Every object has a **`nil?`** method, where **`nil`** returns **`true`** for it
  - Note: **`nil`** and **`false`** are "false", everything else is "true"
- Strings also have a **`+`** method
  - String concatenation
  - Example: **`"hello" + 3.to_s`**

All code is methods

- All methods you define are part of a class
- Top-level methods just added to **`Object`** class
  - Private in file, public in REPL, more or less (details are weird and not so important to us)
- Subclassing discussion coming later, but:
  - Since all classes you define are *subclasses* of **`Object`**, all *inherit* the top-level methods
  - So you can call these methods anywhere in the program
  - Unless a class overrides (*roughly-not-exactly*, shadows) it by defining a method with the same name

Reflection and exploratory programming

- All objects also have methods like:
  - **`methods`**
  - **`class`**
- Can use at run-time to query “what an object can do” and respond accordingly
  - Called *reflection*
- Also useful in the REPL to explore what methods are available
  - May be quicker than consulting full documentation
- Another example of “just objects and method calls”

Changing classes

- Ruby programs (or the REPL) can add/change/replace methods while a program is running
- Breaks abstractions and makes programs very difficult to analyze, but it does have plausible uses
  - Simple example: Add a useful helper method to a class you did not define
    - Controversial in large programs, but may be useful
- For us: Helps re-enforce “the rules of OOP”
  - Every object has a class
  - A class determines its instances’ behavior

The moral

- Dynamic features cause interesting semantic questions
- Example:
  - First create an instance of class **`C`**, e.g., **`x = C.new`**
  - Now replace method method **`m`** in **`C`**
  - Now call **`x.m`**

  Old method or new method? In Ruby, new method

  The point is Java/C#/C++ do not have to ask the question
  - May allow more optimized method-call implementations as a result

Exercise 1

- Objective: Develop a simple banking system using object-oriented programming principles in Ruby. The system should allow users to create accounts, perform transactions, and manage their balances.
- Requirements:

1. Create a **BankAccount** class with attributes such as account_number, account_holder_name, balance, and methods to deposit, withdraw, and display balance.
2. Implement error handling for cases like attempting to withdraw more money than the available balance or providing incorrect account information.
3. Create a **Bank** class to manage multiple bank accounts. Include methods to add new accounts, close accounts, and display account information.
4. Ensure that each bank account has a unique account number.
5. Write a simple command-line interface (CLI) to interact with the banking system, allowing users to create accounts, perform transactions, and view account information.

Exercise 2

- **Objective:**
- Extend a provided Vehicle class in Ruby by creating derived classes and implementing additional functionalities.

```
class Vehicle
  attr_accessor :make, :model, :year

  def initialize(make, model, year)
    @make = make
    @model = model
    @year = year
  end

  def info
    "#{@year} #{@make} #{@model}"
  end
end
```

Exercise 2 (cont.)

- **Requirements:**

1. Create two derived classes named Car and Motorcycle that inherit from the Vehicle class.
2. Extend the Car class by adding attributes specific to cars, such as num_doors, num_passengers, and methods to display car information and check the number of doors and passengers.
3. Extend the Motorcycle class with attributes like type (e.g., cruiser, sportbike), engine_size, and methods to display motorcycle information and obtain engine size details.
4. Implement error handling for validating input data when initializing objects or performing operations.

- **Tasks:**

1. Create instances of the Car and Motorcycle classes, initialize them with different attributes, and demonstrate the functionality of their respective methods.
2. Ensure that the methods from the Vehicle class (info) are accessible and functional in the derived classes.