



# CSE341: Programming Languages Lecture 19 Introduction to Ruby and OOP

Dan Grossman Spring 2013

# **Ruby logistics**

- Next two sections use the Ruby language
  - http://www.ruby-lang.org/
  - Installation / basic usage instructions on course website
    - Version 1.9.x required, but differences not so relevant
- Excellent documentation available, much of it free
  - So may not cover every language detail in course materials
  - http://ruby-doc.org/
  - http://www.ruby-lang.org/en/documentation/
  - Particularly recommend "Programming Ruby 1.9, The Pragmatic Programmers' Guide"
    - Not free

# Ruby: Our focus

- Pure object-oriented: all values are objects (even numbers)
- Class-based: Every object has a class that determines behavior
  - Like Java, unlike Javascript
  - *Mixins* (neither 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

Spring 2013

# Ruby: Not our focus

- Lots of support for string manipulation and regular expressions
- Popular for server-side web applications
   Ruby on Rails
- Often many ways to do the same thing
  - More of a "why not add that too?" approach

# Where Ruby fits

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

Note: 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

# A note on the homework

Next homework is about understanding and extending an *existing* program in an *unfamiliar* language

- Good practice
- Quite different feel than previous homeworks
- Read code: determine what you do and do not (!) need to understand

Homework requires the Tk graphics library to be installed such that the provided Ruby code can use it

### Getting started

- See . **rb** file for our first program
  - (There are much shorter ways to write the same thing)
- Can run file foo.rb at the command-line with ruby foo.rb
- Or can use irb, which is a REPL
  - Run file foo.rb with load "foo.rb"

# 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 new class called 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()
- 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
- 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**
- (Same as this in Java/C#/C++)

#### 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

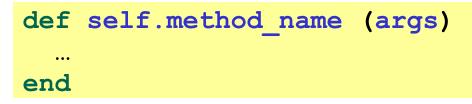
- A method named initialize is special
  - Is called on a new object before **new** returns
  - Arguments to **new** are passed on to **initialize**
  - Excellent for creating object invariants
  - (Like constructors in Java/C#/etc.)
- Usually good *style* to create instance variables in **initialize** 
  - 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
- 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**):



– 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...

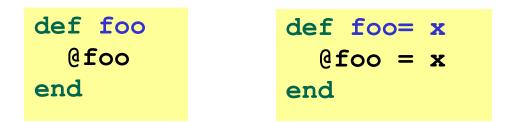
#### Object state is private

- 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 <code>@foo</code> the convention is to name the methods



- 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

# Now (see the code)

- Put together much of what we have learned to define and use a small class for rational numbers
  - Called MyRational because Ruby 1.9 has great built-in support for fractions using a class Rational
- Will also use several new and useful expression forms
  - Ruby is too big to show everything; see the documentation
- Way our class works: Keeps fractions in reduced form with a positive denominator
  - Like an ML-module example earlier in course

### 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 (in file or REPL) just added to Object class
- 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

### Examples

- Add a **double** method to our **MyRational** class
- Add a **double** method to the built-in **FixNum** class
- Defining top-level methods adds to the built-in Object class
   Or replaces methods
- Replace the + method in the built-in FixNum class
   Oops: watch irb crash

# 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

# Duck Typing

"If it walks like a duck and quacks like a duck, it's a duck"

- Or don't worry that it may not be a duck

When writing a method you might think, "I need a **Foo** argument" but really you need an object with enough methods similar to **Foo**'s methods that your method works

 Embracing duck typing is always making method calls rather than assuming/testing the class of arguments

Plus: More code reuse; very OOP approach

- What messages an object receive is "all that matters"

Minus: Almost nothing is equivalent

- x+x Versus x\*2 Versus 2\*x
- Callers may assume a lot about how callees are implemented

# Duck Typing Example

```
def mirror_update pt
   pt.x = pt.x * (-1)
end
```

- Natural thought: "Takes a Point object (definition not shown here), negates the x value"
  - Makes sense, though a **Point** instance method more OOP
- Closer: "Takes anything with getter and setter methods for @x instance variable and multiplies the x field by -1"
- Closer: "Takes anything with methods x= and x and calls x= with the result of multiplying result of x and -1"
- Duck typing: "Takes anything with method x= and x where result of x has a \* method that can take -1. Sends result of calling x the \* message with -1 and sends that result to x="

Spring 2013

#### With our example

```
def mirror_update pt
   pt.x = pt.x * (-1)
end
```

- Plus: Maybe mirror\_update is useful for classes we did not anticipate
- Minus: If someone does use (abuse?) duck typing here, then we cannot change the implementation of mirror\_update
  - For example, to pt.x
- Better (?) example: Can pass this method a number, a string, or a MyRational

def double x x + x end