CSE 413
Programming Languages & Implementation

Hal Perkins
Winter 2019

Introduction to Ruby
(adapted from CSE 341, Dan Grossman)
The Plan

• Why Ruby?
• Some basics of Ruby programs
  – Syntax
  – Classes, methods
  – Fields, variables, scope
  – Dynamic typing
• We won’t cover all (or most) of the details in class
• Focus on OO, dynamic typing, blocks, mixins
• References: cse341 videos, etc. + online library docs +
  – Thomas Programming Ruby (3rd or 4th eds, v1.9-2.0, chs. 1-10; 1st ed online, ch 1-8)
    • Electronic copies available ($) from publisher
Logistics

• We’ll use version 2.x for some recent x
  – REPL (irb) + full Ruby
• Installation instructions, etc. on course web:
  – Windows: use “one click installer”
  – OS X: Recent OS X should have it already (run irb in a terminal window to see if it’s there); if not, get command-line tools and install
    • (If you install homebrew that should also have installed command-line tools with Ruby)
  – Linux: use your favorite package manager
Why?

• Because:
  – Pure object-oriented language
    • Interesting, not entirely obvious implications
  – Interesting design decisions
    • Type system, mixins, syntax (“friendly”), etc.
• Also interesting, but we’re skipping: RAILS web framework
  – Major reason for industry interest in Ruby, but no time to cover (would take a month)
  – But you should be able to pick it up after 413
Where Ruby fits

- Design choices for O-O and functional languages

<table>
<thead>
<tr>
<th></th>
<th>dynamically typed</th>
<th>statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>functional</td>
<td>Scheme/Racket</td>
<td>Haskell, ML (not in 413)</td>
</tr>
<tr>
<td>object-oriented</td>
<td>Ruby</td>
<td>Java</td>
</tr>
</tbody>
</table>

- Dynamic typed OO helps isolate OO’s essence without details of type system
- Historical note: Smalltalk
  - Classic dynamically typed, class-based, pure OO
  - Ruby takes much from this tradition
Rules for 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 the instance of a class
5. An object’s class determines the object’s *behavior*
   - How it handles method calls (responds to messages)
   - 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.
Ruby key ideas (1)

- *Everything* is an object (with constructor, fields, methods); even numbers, even classes(!)
- Class based: every object has a class, which determines how it responds to messages
  - Like Java, not like Javascript
- Dynamic typing
  - vs static typing in Java
- Convenient reflection (runtime inspection of objects)
- Dynamic dispatch (like Java)
- Sends to *self* (same as *this* in Java)
Ruby Key Ideas (2)

• Everything is “dynamic”
  – Evaluation can add/remove classes, add/remove methods, add/remove fields, etc.
• Blocks and libraries encourage use of closure idioms
• mixins: interesting modularity feature (not like Java interfaces or C++ multiple inheritance)
• Syntax and scoping rules of a “scripting language”
  – Often many ways to say something – “why not” attitude
  – Variables “spring to life” on first use
  – Some interesting (odd?) scoping rules
• And a few C/Java-like features (loops, return, etc.)
  – Rarely need loops because of blocks, iterators
Defining a class

(class definition from course web)

class Rat =
  # no instance variable (field) declarations
  # just assign to @foo to create field foo
  def initialize (num, den=1)
    ...
    @num = num
    @den = den
  end

  def print ... end
  def add r ... edn
end
Using a class (1)

- `ClassName.new(args)` creates a new instance of `ClassName` and calls its `initialize` method with `args`
- Every variable references an object (possibly the `nil` object – and `nil` really *is* an object)
  - Local variables (in a method) `foo`
  - Instance variables (fields) `@foo`
  - Class variables (static fields) `@@foo`
  - Global variables and constants `$foo $MAX`
Using a class (2)

• You use an object with a method call
  – Also known as message send
  – Object’s class determines its behavior

• Examples: \( x.m \ 4 \ \ x.m1.m2(y.m3) \ -42.abs \)
  – \( m \) and \( m(...) \) are syntactic sugar for \( self.m \) and \( self.m(...) \)
  – \( e1+e2 \) is sugar for \( e1.+(e2) \)  (yup, really!!!)
No Variable Declarations

• If you assign to a variable, it’s mutation
• If the variable is not in scope, it is created(!) (Do not mispeal things!!)
  – Scope of new variable is the method you are in
• Same with fields: if you assign to a field, that object has that field
  – So different objects of the same class can have different fields(!)
• Fewer keystrokes in programs, “cuts down on typing”, but compiler catches fewer bugs
  – A hallmark of “scripting languages”
  – Thorough testing will catch bugs anyway (claim)
Protection?

- Fields are inaccessible outside (individual) instances
  - All instance variables are private
  - Define getter/setter methods as needed
- Methods are public, protected, private
  - public is the default
  - protected: only callable from class or subclass object
  - private: only callable from self
  - protected & private differ from Java (how?)
Getters and setters

• If you want outside access, must define methods
  ```ruby
def foo
  @foo
  end
  end
def foo= x
  @foo = x
  end
  end
  ```
• The foo= convention allows sugar via extra spaces
  ```ruby
  x.foo  x.foo = 42
  ```
• Shorter syntax for defining getters/setters
  ```ruby
  attr_reader :foo  attr_writer :foo
  ```
• Overall, requiring getters/setters is more uniform, OO
  – Can change methods later without changing clients
Class definitions are dynamic

- All definitions in Ruby are dynamic
- Example: Any code can add or remove methods on existing classes
  - Very occasionally useful (or cute) to add your own method to an existing class that is then visible to all instances of that class
- Changing a class affects all instances – even if already created
  - Disastrous example: changing Fixnum’s + method
- Overall: a simple language where everything can be changed and method lookup uses instance’s classes
Unusual syntax
(add to this list as you discover things)

• Newlines often matter – example: don’t need semicolon if a statement ends a line
• Message sends (function calls) with 0 or 1 arguments often don’t need parentheses
• Infix operations like + are just message sends
• Can define operators including = [ ]
• Conditional expressions e1 if e2 and similar things (as well is if e1 then e2)
Unusual syntax
(add to this list as you discover things)

• Classes don’t need to be defined in one place (similar to C#, not Java or C++)
  – A class definition can span multiple files
• Class names must be capitalized
• self is Java’s “this”
• Loops, conditionals, classes, methods are self-bracketing (end with end)
  – Actually not unusual except for programmers with too much exposure to C/Java/C#/C++ and other languages of the curly brace persuasion
A bit about Expressions

• Everything is an expression and produces a value
• nil means “nothing”, but it is an object (an instance of class NilClass)
• nil and false are false in a boolean context; everything else is true (including 0)
• ‘strings’ are taken literally (almost)
• “strings” allow more substitutions
  – including #{expressions}
  – (Elaborate regular expression package. Won’t cover in class but learn/use when needed, like in hw7😊)
Top-level

• Expressions at top-level are evaluated in the context of an implicit “main” object with class Object
  – That is how a standalone program can “get started” rather than requiring creating an object and calling a method (particularly useful in irb)

• Top-level methods are added to Object, which makes them available everywhere

• irb: Ruby REPL/interpreter
  – Use load “filename.rb” to read code from file