CSE 413 Autumn 2008

Introduction to Ruby

Credit: Dan Grossman, CSE341

Why Ruby?

- Because:
  - Pure object-oriented language
    - Interesting, not entirely obvious implications
  - Interesting design decisions (compare Java)
    - Particularly type system, mixins, etc.
- Interesting, but not our focus
  - Scripting language
  - RAILS and other frameworks

Getting Ruby

- Link to www.ruby-lang.org/en on course web. Documentation & downloads
- Implementations:
  - Windows: get the “one-click installer”
  - OS X: Ruby 1.8 is part of developer tools
  - Linux: Should be available from your distro. Be sure to include the irb interactive interpreter too.

Ruby

- Pure object-oriented: *all* values are objects
  - Contrast w/Java primitive vs reference types
- Class-based
- Dynamically Typed
  - vs static typing in Java
- Convenient reflection

Languages Compared

- One way to get an overview of what these mean and how other languages relate

	dynamically typed	statically typed
functional	Scheme	ML (not in 413)
object-oriented	Ruby	Java

Ruby vs Smalltalk (1)

- Smalltalk is the classic example of a pure OO, class-based, dynamically-typed language
  - Basically unchanged since the 80’s
  - Tiny language, regular, can learn whole thing
  - Integrated into a powerful, malleable, GUI environment
  - Uses blocks (closures) for control structures

Ruby vs Smalltalk (2)

- Ruby
  - Large language, “why not” attitude
    - “make programmers happy”
  - Scripting language, minimal syntax
  - Huge library (strings, regexps, RAILS)
  - Mixins (somewhere between Java interfaces and C++ multiple inheritance – very neat)
  - Blocks and libraries for control structures and functional-programming idioms

Ruby Key Ideas (1)

- *Everything* is an object (with constructor, fields, methods)
- Every object has a class, which determines how it responds to messages
- Dynamic typing (everything is an object)
- Dynamic dispatch (like Java; later)
- 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 are *almost* first-class anonymous functions (later)
  - Can convert to/from real lambdas
- And a few C/Java-like features (loops, return,etc.)

No Variable Declarations

- If you assign to a variable, it’s mutation
- If the variable is not in scope, it is created(!)   (Don’t mispell things!!)
  - Scope is the current method
- 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(!)

Naming Conventions

- Used to distinguish kinds of variables
  - Constants and ClassNames start with caps
  - local_vars and parameters start w/lower case
  - @instance_variables
    - @thing = thing sets and instance variable from a local name – and creates @thing if it doesn’t exist!
  - @@class_variables
  - $global $VARS $CONSTANTS

Visibility. Protection?

- Fields are inaccessible outside instance
  - Define accessor/mutator methods as needed
- Methods are public, protected, private
  - protected: only callable from class or subclass object
  - private: only callable from *self*
  - Both of these differ from Java

Unusual syntax

(add to this list as you discover things)

- Newlines often matter – example: don’t need semi-colon when a statement ends a line
- Message sends (function calls) often don’t need parentheses
- Infix operations are just message sends
- Can define operators including =, [ ]
- Classes don’t need to be defined in one place
- Loops, conditionals, classes, methods are self-bracketing (end with “end”)
  - Actually not unusual except to programmers who have too much prior exposure to C/Java, etc.

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}