

Section 9 Intro to Ruby

Portions of slides adapted from Josh Pollock

Spring 2020

1

Learning Objectives

- Review Ruby classes and objects
- Introduce arrays, hashes, and ranges
- Ruby closures: blocks, procs, and lambdas

2

Getting Started with Ruby

- Make sure to follow the instructions for using a VM for Ruby on the [course website](#) (which also provides an image)
- **Please do this by tomorrow to account for any possible issues**

3

Review: 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

4

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

5

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

6

Ruby Class Exercise

Let's write a class `BankAccount` which:

- Can be initialized with an optional argument for starting balance otherwise has \$0 in funds initially
- Has a method `withdraw` to withdraw `x` funds, returning the amount withdrawn (if the balance is less than the argument, set the balance to 0)
- Has a method `deposit` to deposit `x` funds to the balance
- Has a `get_balance` method to return the current balance
- Has method `merge_accounts` which takes another `BankAccount` and adds its balance to the current object
- Has a `to_s` method to return a string representation of the balance in `$X.XX` format (e.g. "`$3.41`")

What are some possible invalid arguments to consider for different methods? Class invariants? Are there any appropriate helper methods to make protected or private?

7

Arrays

- Ruby uses dynamically sized arrays like Java's `ArrayLists`.
- These are nice middle ground between linked lists and statically sized arrays.
- Allow fast random access and asymptotically fast insertion and deletion.
- Ruby array entries don't need to have the same type
- ("natural" in dynamically typed languages)
- Ruby arrays are super flexible.
- Ruby uses arrays for lists, sets, stacks, and queues!

8

Examples

Let's see some code examples and more useful methods using arrays.

9

Hashes: Dynamic Records

- A map from keys to values.
- Keys don't have to have the same type!
- Keys and entries are mutable. They can be updated dynamically.
- See code for examples.

10

Ranges: The Power of Enumerators

- Ranges are enumerators, not lists.
- Somewhat like the streams we saw in Racket, they are lazy.
- The only do computation when necessary.
- Syntax:
 - `i..j [i, j]` -- includes `j`
 - `i...j [i, j)` -- excludes `j`
- For step size, use `.step`

11

The Takeaway

- Ruby has several flexible ways of constructing complex data.
- This flexibility is characteristic of dynamically typed languages (cf. Python).
- Consult the Ruby documentation. It's really good.

12

Ruby Closures

- Ruby gives us 3 ways to define a closure:
 - Block
 - Proc
 - Lambda
- Lexical scope, but variables are stored as references to objects
- E.g. Modifying an array referenced by a closure may change its behavior
- Use `.call` to call them

13

Block Cheat Sheet

- The most common type of closure in Ruby
- *All* methods take a block argument, it may not be used
- Call a block with `yield`
- Use `return` to return from an enclosing method
- Give a block an explicit name with `&block_name`

14

Procs

- Procs are essentially blocks as objects.
- Initialize like any other object.

15

Issues with Blocks and Procs

- `return` jumps out of the method where the block was called.
- They don't check they're passed the right number of arguments.

16

Lambda

- Lambda is a special kind of Proc with special behavior
- Create with `lambda` or `->`
- Work like "normal" closures
- `return` returns from the lambda
- Lambda checks it gets the right number of arguments

17

Practice Using Blocks and Procs

Let's write `Array#map`

18

The Takeaway

- Ruby takes a pragmatic, OO approach to first-class functions.
- The typical case is supported by blocks. You should use them most often.
- Ruby is a real-world language so it supports the long-tail of use cases with Proc and lambda.
- This makes the language more complex, especially because Proc and lambda extend the language implementation.