

What is Ruby?

CSc 372

Comparative Programming Languages

27 : Ruby — Introduction

Department of Computer Science
University of Arizona

collberg@gmail.com

Copyright © 2011 Christian Collberg

Christian Collberg

- Everything is an object.
- Everything can be changed: method can be added to classes at runtime, for example.
- There's no "compile-time": everything happens at runtime.
- Variables have no type, they can contain different kinds of objects at different times.
- Classes are not "types" the way they are in Java — A class is identified by the messages (method calls) it responds to.

Ducks!

- Create a class and a constructor (it's called `initialize`).
- Instance variables start with `@`.
- `Duck.new` is a standard class (static) method that creates a new object.
- The class definition is actually executable: it's executed at runtime and creates the class.
- The statements after the class definition are also executed, as the file is loaded.

Defining a class

```
class Duck
  def initialize(name,type)
    @name = name
    @type = type
  end

end

d1 = Duck.new("larry", "rubber")
puts d1
```

Running Ruby

- Run like this:

```
> ruby ducks.rb
```
- Or like this:

```
> irb --prompt simple -r ducks.rb  
>> d1 = Duck.new("larry","rubber")  
>> puts d1
```
- irb is the interactive Ruby shell.
- <http://ruby-doc.org/docs/ProgrammingRuby/html/irb.html>

Running Ruby

- You can also (if you're on a Unix system) put your script in a file like this:

```
#!/usr/local/bin/ruby  
puts "hello ducks!"
```
- Make the file executable, and then you can execute it like any other Unix program:

```
> chmod a+rx hello.rb  
> hello.rb  
hello ducks!
```

Syntax

- No semi-colons, as long as you keep one statement per line.
- Comments start with a # and go to the end of the line.

```
# This is a comment.
```
- You can leave out parentheses around method arguments (but don't). These are the same:

```
# This is a comment.  
puts("arg1","arg2")  
puts "arg1","arg2"
```

Pretty printing

- To print an object in a pretty way, we can redefine to_s. This is like overriding Java's toString.

```
class Duck  
  def to_s  
    @name + " : " + @type  
  end  
end  
  
puts d1
```

Adding a method

- Let's add a new method, `quack!`.
- Method names can end in `!` (typically for methods that change some data), `?` (methods that return `true/false`), and `=` (setter methods).
- Notice that we're not actually editing the class definition, but simply adding another method at runtime!

```
class Duck
  def quack!
    puts "quack!"
  end
end

d1.quack!
```

Method parameters

- Class names should start with an upper case letter, method names with a lower case.
- Add a parameter to `quack!`. The new definition replaces the old one. There's no overloading (methods with different types/number of parameters are different) like in Java.

```
class Duck
  def quack!(times)
    puts "quack! " * times
  end
end

d1.quack!(5)
```

Overloaded operators

- There is plenty of operator overloading, however, and you can add your own overloaded operators, if you want.

```
>> 5*7
=> 35
>> 5*"7"
TypeError: String can't be coerced into Fixnum
>> "7"*6
=> "777777"
```

Types

- Variables don't have type, but objects have. You can ask an object's type using `.class`.

```
>> 5**57
=> 6938893903907228377647697925567626953125
>> 5.class
=> Fixnum
>> (5**57).class
=> Bignum
>> "duck!".class
=> String
>> r1 = Duck.new("larry", "rubber")
>> r1.class
=> Duck
```

Arrays

- Arrays can contain any type of object.
- Arrays are indexed by integers, starting from 0.
- You can break a line into two parts if you end the first one with an operator (, in this case).

```
flock = [d1,Duck.new("ruby","rubber"),
        "roast duck"]
puts flock

puts flock[0]
puts flock[1..2]
flock[0] = "kao ya"
puts flock
```

Hashes

- Hashtables are indexed by, well, anything. You can map one object to any other kind of object.

```
flock = {
  d1 => "hot",
  Duck.new("ruby","rubber") => "cute",
  "roast duck" => "tasty"
}
puts flock
```

Hashes

- OK, that's ugly. We need to change the way the Hash class prints out a table. No problem!

```
class Hash
  def to_s
    s = ""
    self.each do |key,value|
      s = s + key.to_s + "\t=>\t" +
          value.to_s + "\n"
    end
    return s
  end
end
```

Hashes

- This is a Ruby *iterator*. `each` is a method which generates all pairs of keys and values.
- `|key,value|` are local variables within the `do...end` block. each will invoke this block (giving `key` and `value` their values) for every pair in the hashtable.

```
self.each do |key,value|
  s = s + key.to_s + "\t=>\t" +
      value.to_s + "\n"
end
```

- Many classes define the <<-operator. For strings, it appends a value onto the end of the string. For arrays, it adds an element to the end of the array.

```
s = "yo"
s << ",dude"
a = [1,2,3]
a << "ducks are cute as can be!"
```

Hashes

- To look up an element in a hashtable, use `hash[key]`.
- To delete an element, use `hash.delete(key)`.
- To add/override an element, use `hash[key]=value`.

```
puts flock[d1]
puts flock[Duck.new("larry","rubber")]

flock.delete(d1)
puts flock
```

String interpolation

- Inside strings you can put arbitrary Ruby code contained within `#{...}`. It gets executed and the result filled in inside the string.

```
self.each do |key,value|
  s << "#{key.to_s}\t=>\t #{value.to_s}\n"
end
```

```
balloons = 98
puts "#{balloons} luftballons!"
puts "#{balloons+1} luftballons!"
```

if-expressions

- `hash.has_key?(key)` returns true if the hash table contains a value for that key.

```
if flock.has_key?("roast duck") then
  puts "found supper!"
end
```

```
if flock.has_key?("roast duck") then
  puts "found supper!"
else
  puts "I'm hungry! ):"
end
```

if-expressions

- Everything in Ruby produces a value, even `if`, `while`, etc.

```
x = if flock.has_key?("roast duck") then
  "(-:" else "):" end
```

while-loops

- Like the `if`-expression, `while` ends with an `end`.

```
ducks = 0
while ducks < 10
  puts "I love ducks!"
  ducks += 1
end
```

Statement modifiers

- `if` and `while` have shortcuts called *modifiers*. These can be used when the body of the `if` or `while` is a single expression.

```
ducklovers = 1
puts "Some people love ducks!" if ducklovers > 0
```

```
puts "Some people love ducks!" \
  unless ducklovers == 0
```

```
ducklovers += 1 while ducklovers < 100
puts ducklovers
```

Regular expressions

- Ruby has regular expressions (REs) built in.
- REs are used to parse and take strings apart.
- An RE is given within `/.../`.
- `string.scan(re)` searches through the string and returns any matches.
- `scan` either returns an array of the results, or can be used as an iterator.
- You can either use `each` or the `for i in iterator do ...i ...end` construction.

Regular expressions

- . (period) matches any character:

```
"duck".scan(/./)
```

```
for i in "duck".scan(/./) do
  puts i
end
```

```
"duck".scan(/./).each do |i|
  puts i
end
```

Regular expressions

- .. (period) matches any *two* characters.
- “Normal characters” (like letters and digits) match themselves.
- “Special characters” (or *meta*-characters) have to be escaped (preceded by a backslash). This includes characters like the brackets and parentheses that have special meanings in REs.

```
"duck".scan(/../)
"duck42,duck46".scan(/4/)
"duck42,duck46".scan(/du/)
"duck42/duck46".scan(/\//)
```

Regular expressions

- [...] defines a *character class*, a set of characters we want to match.
- [from-to] defines a *range* of characters

```
"pluckyducky".scan(/[uc]/)
"ducky".scan(/[a-k]/)
```

Regular expressions

- Assume that we've got a file of ducks, where consecutive ducks are separated by , (commas), and the name and type of duck is separated by / (slash).
- Assume that names and types consist of the characters a-z.
- Start by separating the ducks:

```
data = "larry/rubber,ruby/rubber,carl/roast"
for i in data.scan(/[a-z\//]+/)
  puts i
end
```

Regular expressions

- Next, scan for the name and the type, and print them out.
- Here we're both using the *return result as array* and return results one at a time in an iterator versions of scan.

```
for i in data.scan(/[a-z\ ]+/)
  a = i.scan(/[a-z]+/)
  puts a[0] + "=>" + a[1]
end
```

Regular expressions

- Finally, create a hashtable containing the data we just read in and parsed:

```
flock = {}
for i in data.scan(/[a-z\ ]+/)
  a = i.scan(/[a-z]+/)
  flock[a[0]] = a[1]
end
puts flock
```

Regular expressions

- The =~ returns the position of the match if the string matches the regular expression, nil otherwise.
- x+ matches one or more xs.
- x* matches zero or more xs.
- x|y matches x or y.

Regular expressions

```
if "donald" =~ /daisy|donald/ then
  puts "duck match!"
end
```

```
if "ddddduck" =~ /d+uck/ then
  puts "duck match!"
end
```

```
if "uck" =~ /d*uck/ then
  puts "duck match!"
end
```

```
if "duck" =~ /d*uck/ then
  puts "duck match!"
end
```


Regular expressions

- `string.sub(pattern, replace)` replaces the first occurrence of `pattern` with `replace`, in `string`.
- `gsub` does the same, but replaces all occurrences.

```
puts "duckduckduck".sub(/duck/, "ruby")
puts "duckduckduck".gsub(/duck/, "ruby")
puts "duck4luck!".gsub(/[a-z]/, "-")
puts "daisydonaldruby".gsub(/daisy|donald/, "duck")
```

Global Variables

- Global variables are prefixed with a \$ (dollar) sign.

```
$MyDucks = ["larry duck", "sally duck"]
puts $MyDucks
```

nil

- `nil` is an object, like any other. It is returned by many operations. It represents “nothing.”
- `nil` means `false` in conditional expressions.

```
nil.class
a = []
a[5]
```

Class methods and variables

- Class variables start with `@@`.
- Class methods start with the class name followed by a `.` (period).

```
class Duck
  @@count = 0
  def initialize(name, type)
    @name = name
    @type = type
    @@count += 1
  end
  def Duck.howMany
    return @@count
  end
end
```

Class methods and variables. . .

```
d1 = Duck.new("larry","rubber")
d2 = Duck.new("sally","rubber")
d3 = Duck.new("jessie","rubber")
puts Duck.howMany
```

Constants

- Constants start with an uppercase letter.
- This is actually why classes must start with an uppercase letter — they are constants inserted into an internal dictionary.

Class methods and variables

```
MAXDUCKS = 2
class Duck
  @@count = 0
  def initialize(name,type)
    if @@count == MAXDUCKS then
      puts "no more ducks for you!"
      raise RangeError
    end
    @name = name; @type = type; @@count += 1
  end
  def Duck.howMany
    return @@count
  end
end
```

Blocks and iterators

- A *block* of code goes between curly braces or within `do...end`:
`[1,2,3].each {|x| puts x}`
`[1,2,3].each do |x|
 puts x
end`
- Curly braces are used for short pieces of code.
- Arguments to the block is given within `|...|`.

Blocks and iterators

- So, what does this really mean?

```
[1,2,3].each do |x|
  puts x
end
```

- each is a method, invoked on the array [1,2,3].
- The do...end block is passed to each.
- Control then “jumps” back-and-forth between each and the block: each generates a value from the array, passes it to the block (in the x variable), the block prints it out, and passes control back to each so it can generate the next value.

Blocks and iterators

- Of course, nothing stops us from writing our own iterators, or to extend standard classes with new ones!

```
class Array
  def myEach
    i = 0
    while i < self.length
      yield self[i]
      i += 1
    end
  end
end
```

```
[1,2,3].each {|x| puts x}
[1,2,3].myEach {|x| puts x}
```

Blocks and iterators

- yield “jumps” into the block, passing one or more values along.
- This is sometimes known as a *co-routine*: You have two pieces of code, both active at the same time, and control bounces back and forth between them.

```
class Duck
  def Duck.kindsOf
    yield "roast"
    yield "rubber"
    yield "poached"
  end
end
```

```
Duck.kindsOf {|x| puts x}
```

Readings

- Read Chapter 2, page 3–41, in *Programming Ruby — The Pragmatic Programmers Guide*, by Dave Thomas.
- Read Chapter 13, page 163–170,173, in *Programming Ruby*.
- Read Chapter 15, page 185–187,195, in *Programming Ruby*.
- The first edition of this book is available online at

<http://www.rubycentral.com/pickaxe/index.html>.

... you're the one. . .

