CMSC 330: Organization of Programming Languages

More Ruby:

Methods, Classes, Arrays, Hashes

In Ruby, everything is an Object

- Ruby is object-oriented
- All values are (references to) objects
  - Java/C/C++ distinguish *primitives* from *objects*
- Objects communicate via method calls
- Each object has its own (private) state
- Every object is an instance of a class
  - An object's class determines its behavior:
  - The class contains method and field definitions
    - Both instance fields and per-class ("static") fields

Everything is an Object

- Examples
  - (-4).abs  No-argument instance method of Fixnum
    - integers are instances of class Fixnum
  - 3 + 4
    - infix notation for "invoke the + method of 3 on argument 4"
  - "programming".length
    - strings are instances of String
  - String.new
    - classes are objects with a new method
  - 4.13.class
    - use the class method to get the class for an object
    - floating point numbers are instances of Float

Ruby Classes

- Class names begin with an uppercase letter
- The new method creates an object
  - s = String.new creates a new String and makes s refer to it
- Every class inherits from Object

Objects and Classes

- Objects are data
- Classes are types (the kind of data which things are)
- Classes are *also* objects

Object	Class (aka *type*)
10	Integer
-3.30	Float
"CMSC 330"	String
String.new	String
[‘a’, ‘b’, ‘c’]	Array
Integer	Class

- Integer, Float, and String are *objects* of type Class
  - So is Class itself!

Two Cool Things to Do with Classes

- Since classes are objects, you can manipulate them however you like
  - Here, the type of y depends on p
    - Either a String or a Time object

```
if p then
  x = String
else
  x = Time
End
y = x.new
```

- You can get names of all the methods of a class
  - Object.methods
    - => ["send", "name", "class_eval", "object_id", "new", "autoload?", "singleton_methods", ...]

Standard Library: String class

- Strings in Ruby have class String
  - “hello”.class == String
- The String class has many useful methods
  - s.length # length of string
  - s1 == s2 # structural equality (string contents)
  - s = "A line\n"; s.chomp # returns "A line"
    - Return new string with s's contents minus any trailing newline
  - s = "A line\n"; s.chomp!
    - Destructively removes newline from s
    - *Convention:* methods ending in ! modify the object
    - *Another convention:* methods ending in ? observe the object

Creating Strings in Ruby

- Substitution in double-quoted strings with #{ }
  - course = "330"; msg = "Welcome to #{course}"
  - "It is now #{Time.new}"
  - The contents of #{ } may be an arbitrary expression
  - Can also use single-quote as delimiter
    - No expression substitution, fewer escaping characters
- Here-documents

```
s = <<END
  This is a text message on multiple lines
  and typing \\n is annoying
END
```

Creating Strings in Ruby (cont.)

- Ruby has printf and sprintf
  - printf("Hello, %s\n", name);
  - sprintf("%d: %s", count, Time.now)
    - Returns a String
- to_s returns a String representation of an object
  - Can be invoked implicitly – write puts(p) instead of puts(p.to_s)
    - Like Java's toString()
- inspect converts any object to a string

```
irb(main):033:0> p.inspect
=> "#<Point:0x54574 @y=4, @x=7>"
```

Symbols

- Ruby *symbols* begin with a colon
  - :foo, :baz_42, :"Any string at all"
- Symbols are “interned” Strings
  - The same symbol is at the same physical address
  - Can be compared with physical equality

```
“foo” == “foo”      # true
“foo”.equal? “foo” # false
:foo == :foo        # true
:foo.equal :foo     # true
```

- Are symbols worth it? Probably not…

The nil Object

- Ruby uses nil (not null)
  - All uninitialized fields set to nil (@ prefix used for fields)

```
irb(main):004:0> @x
=> nil
```

- nil is an object of class NilClass
  - Unlike null in Java, which is a non-object
  - nil is a *singleton object* – there is only one instance of it
    - NilClass does not have a **new** method
  - nil has methods like to_s, but not other methods

```
irb(main):006:0> nil + 2
NoMethodError: undefined method `+' for nil:NilClass
```

Quiz 1

- What is the type of variable x at the end of the following program?

```
p = nil
x = 3
if p then
  x = nil
else
  x = "hello"
end
```

A. Integer
B. NilClass
C. String
D. *Nothing* – there's a type error

Quiz 1

- What is the type of variable x at the end of the following program?

```
p = nil
x = 3
if p then
  x = nil
else
  x = "hello"
end
```

A. Integer
B. NilClass
C. **String**
D. *Nothing* – there's a type error

Arrays and Hashes

- Ruby data structures are typically constructed from Arrays and Hashes
  - Built-in syntax for both
  - Each has a rich set of standard library methods
  - They are integrated/used by methods of other classes

Array

- Arrays of objects are instances of class Array
  - Arrays may be heterogeneous

```
a = [1, "foo", 2.14]
```

- C-like syntax for accessing elements
  - indexed from 0
  - return nil if no element at given index

```
irb(main):001:0> b = []; b[0] = 0; b[0]
 => 0
irb(main):002:0> b[1]  # no element at this index
 => nil
```

Arrays Grow and Shrink

- Arrays are growable
  - Increase in size automatically as you access elements

```
irb(main):001:0> b = []; b[0] = 0; b[5] = 0; b
 => [0, nil, nil, nil, nil, 0]
```

  - [ ] is the empty array, same as Array.new
- Arrays can also shrink
  - Contents shift left when you delete elements

```
a = [1, 2, 3, 4, 5]
a.delete_at(3)          # delete at position 3; a = [1,2,3,5]
a.delete(2)             # delete element = 2; a = [1,3,5]
```

Iterating Through Arrays

- It's easy to iterate over an array with while
  - length method returns array's current length

```
a = [1,2,3,4,5]
i = 0
while i < a.length
  puts a[i]
  i = i + 1
end
```

- Looping through elements of an array is common
  - We'll see a better way soon, using code blocks

Arrays as Stacks and Queues

- Arrays can model stacks and queues

```
a = [1, 2, 3]
a.push("a")      # a = [1, 2, 3, "a"]
x = a.pop        # x = "a"
a.unshift("b")   # a = ["b", 1, 2, 3]
y = a.shift      # y = "b"
```

Note that `push`, `pop`, `shift`, and `unshift` all permanently modify the array

Hash

- A hash acts like an associative array
  - Elements can be indexed by *any kind* of value
  - Every Ruby object can be used as a hash key, because the Object class has a hash method

- Elements are referred to like array elements

```
italy = Hash.new
italy["population"] = 58103033
italy["continent"] = "europe"
italy[1861] = "independence"
pop = italy["population"] # pop is 58103033
planet = italy["planet"]  # planet is nil
```

Hash methods

- new(o) returns hash whose default value is o
  - h = Hash.new("fish"); h["go"] # returns "fish"
- values returns array of a hash's values
- keys returns an array of a hash's keys
- delete(k) deletes mapping with key k
- has_key?(k) is true if mapping with key k present
  - has_value?(v) is similar

Hash creation

Convenient syntax for creating literal hashes

- Use { key => value, ... } to create hash table

```
credits = {
  "cmsc131" => 4,
  "cmsc330" => 3,
}

x = credits["cmsc330"]  # x now 3
credits["cmsc311"] = 3
```

- Use { } for the empty hash

Quiz 2: What is the output?

```
a = {“foo” => “bar”}
a[0] = ”baz”
print a[0]
print a[1]
print a[“foo”]
```

A. Error
B. barbaz
C. bazbar
D. baznilbar

Quiz 2: What is the output?

```
a = {“foo” => “bar”}
a[0] = ”baz”
print a[0]
print a[1]
print a[“foo”]
```

A. Error
B. barbaz
C. bazbar
D. baznilbar

Quiz 3: What is the output?

```
a = { "Yellow" => [] }
a["Yellow"] = {}
a["Yellow"]["Red"] = ["Green", "Blue"]
print a["Yellow"]["Green"][1]
```

A. Green
B. *(nothing)*
C. Blue
D. *Error*

Quiz 3: What is the output?

```
a = { "Yellow" => [] }
a["Yellow"] = {}
a["Yellow"]["Red"] = ["Green", "Blue"]
print a["Yellow"]["Green"][1]
```

A. Green
B. *(nothing)*
C. Blue
D. *Error – undefined method [] for NilClass*

Quiz 4: What is the output?

```
a = [1,2,3]
a[1] = 0
a.pop
print a[1]
```

A. *Error*
B. 2
C. 1
D. 0

Quiz 4: What is the output?

```
a = [1,2,3]
a[1] = 0
a.pop
print a[1]
```

A. *Error*
B. 2
C. 1
D. 0

Defining Your Own Classes

```
class Point
  def initialize(x, y)
    @x = x
    @y = y
  end

  def add_x(x)
    @x += x
  end

  def to_s
    return "(" + @x.to_s + "," + @y.to_s + ")"
  end
end

p = Point.new(3, 4)
p.add_x(4)
puts(p.to_s)
```

class name is uppercase

constructor definition

instance variables prefixed with "@"

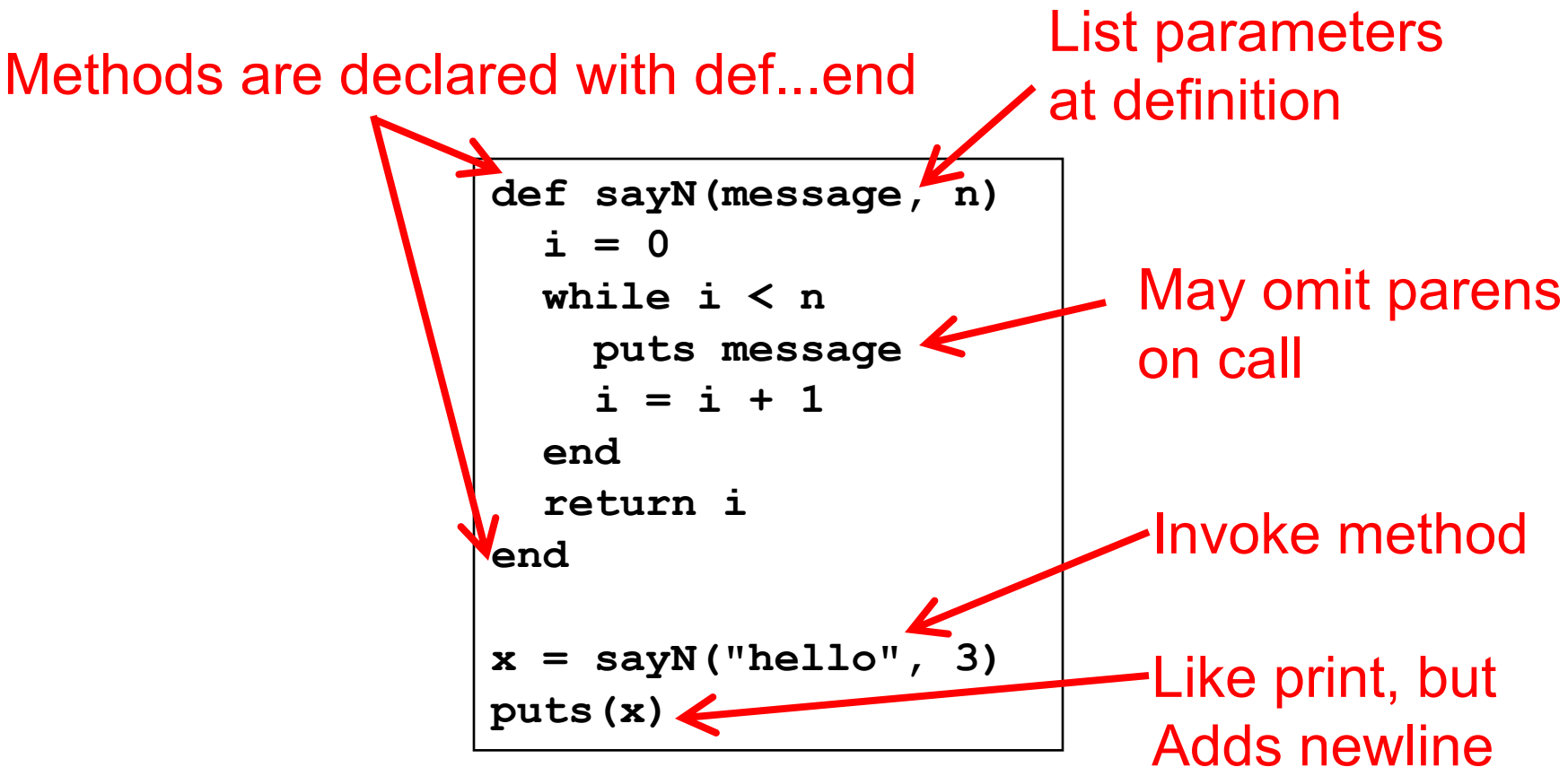
method with no arguments

instantiation

invoking no-arg method

Methods in Ruby

Note: Methods need not be part of a class



Methods should begin with lowercase letter and be defined before they are called
Variable names that begin with uppercase letter are *constants* (only assigned once)

Methods: Terminology

- Formal parameters
  - Variable parameters used in the method
  - def sayN(message, n) in our example
- Actual arguments
  - Values passed in to the method at a call
  - x = sayN("hello", 3) in our example
- Top-level methods are “global”
  - Not part of a class. sayN is a top-level method.

Method Return Values

- Value of the return is the value of the last executed statement in the method
  - These are the same:

```
def add_three(x)
  return x+3
end
```

```
def add_three(x)
  x+3
end
```

- Methods can return multiple results (as an Array)

```
def dup(x)
  return x,x
end
```

Method naming style

- Names of methods that return **`true`** or **`false`** should end in **`?`**
- Names of methods that modify an object's state should end in **`!`**
- Example: suppose **`x = [3,1,2]`** (this is an array)
  - **`x.member? 3`** returns true since **`3`** is in the array **`x`**
  - **`x.sort`** returns a **`new`** array that is sorted
  - **`x.sort!`** modifies **`x`** in place

No Outside Access To Internal State

- An object's instance variables (with `@`) can be directly accessed only by instance methods
- Outside class, they require accessors:

A typical getter

```
def x
   @x
end
```

A typical setter

```
def x= (value)
   @x = value
end
```

- Very common, so Ruby provides a shortcut

```
class ClassWithXandY
   attr_accessor :x, :y
end
```

Says to generate the x= and x and y= and y methods

No Method Overloading in Ruby

- Thus there can only be one initialize method
  - A typical Java class might have two or more constructors
- No overloading of methods in general
  - You can code up your own overloading by using a variable number of arguments, and checking at run-time the number/types of arguments
- Ruby does issue an exception or warning if a class defines more than one initialize method
  - But last initialize method defined is the valid one

Quiz 5: What is the output?

```
class Dog
   def smell(thing)
     "I smelled #{thing}”
   end
   def smell(thing,dur)
     "#{smell(thing)} for #{dur} seconds”
   end
end
fido = Dog.new
puts fido.smell(”Alice”,3)
```

A. I smelled Alice for nil seconds
B. I smelled #{thing}
C. I smelled Alice
D. *Error*

Quiz 5: What is the output?

```
class Dog
   def smell(thing)
     "I smelled #{thing}”
   end
   def smell(thing,dur)
     "#{smell(thing)} for #{dur} seconds”
   end
end
fido = Dog.new
puts fido.smell(”Alice”,3)
```

A. I smelled Alice for nil seconds
B. I smelled #{thing}
C. I smelled Alice
D. *Error – call from Dog expected two args*

Quiz 6: What is the output?

```
class Dog
   def smell(thing)
     "I smelled #{thing}”
   end
   def smelltime(thing,dur)
     "#{smell(thing)} for #{dur} seconds”
   end
end
fido = Dog.new
puts fido.smelltime(”Alice”,3)
```

A. I smelled Alice for seconds
B. I smelled #{thing} for #{dur} seconds
C. I smelled Alice for 3 seconds
D. *Error*

Quiz 6: What is the output?

```
class Dog
   def smell(thing)
     "I smelled #{thing}”
   end
   def smelltime(thing,dur)
     "#{smell(thing)} for #{dur} seconds”
   end
end
fido = Dog.new
puts fido.smelltime(”Alice”,3)
```

A. I smelled Alice for seconds
B. I smelled #{thing} for #{dur} seconds
C. **I smelled Alice for 3 seconds**
D. *Error*

Inheritance

- Recall that every class inherits from Object

```
class A          ## < Object
  def add(x)
    return x + 1
  end
end

class B < A
  def add(y)
    return (super(y) + 1)
  end
end

b = B.new
puts(b.add(3))
```

extend superclass

invoke add method of parent

```
b.is_a? A
true
b.instance_of? A
false
```

super( ) in Ruby

- Within the body of a method
  - Call to super( ) acts just like a call to that original method
  - Except that search for method body starts in the superclass of the object that was found to contain the original method

Quiz 7: What is the output?

```
class Gunslinger
  def initialize(name)
    @name = name
  end
  def full_name
    "#{@name}"
  end
end
class Outlaw < Gunslinger
   def full_name
      "Dirty, no good #{super}"
   end
end
d = Outlaw.new("Billy the Kid")
puts d.full_name
```

A. Dirty, no good Billy the kid
B. Dirty, no good
C. Billy the Kid
D. *Error*

Quiz 7: What is the output?

```
class Gunslinger
  def initialize(name)
    @name = name
  end
  def full_name
    "#{@name}"
  end
end
class Outlaw < Gunslinger
   def full_name
      "Dirty, no good #{super}"
   end
end
d = Outlaw.new("Billy the Kid")
puts d.full_name
```

A. **Dirty, no good Billy the kid**
B. Dirty, no good
C. Billy the Kid
D. *Error*

Global Variables in Ruby

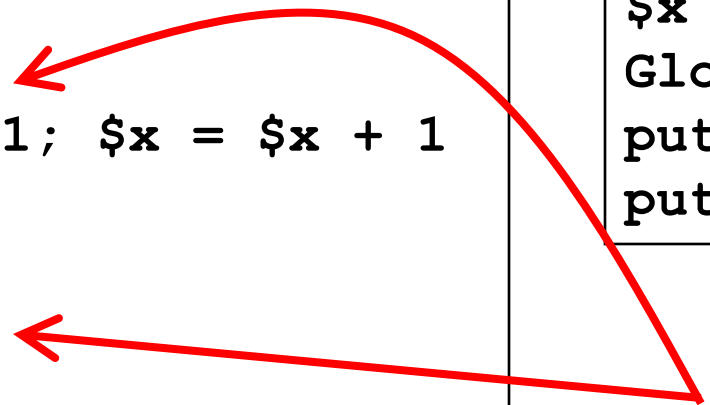
- Ruby has two kinds of global variables
  - Class variables beginning with @@ (`static` in Java)
  - Global variables across classes beginning with $

```
class Global
  @@x = 0

  def Global.inc
    @@x = @@x + 1; $x = $x + 1
  end

  def Global.get
    return @@x
  end
end
```

```
$x = 0
Global.inc
$x = $x + 1
Global.inc
puts(Global.get)
puts($x)
```



define a class ("singleton") method

Quiz 8: What is the output?

```
class Rectangle
 def initialize(h, w)
     @@h = h
     @w = w
 end
 def measure()
  return @@h + @w
 end
End
r = Rectangle.new(1,2)
s = Rectangle.new(3,4)
puts r.measure()
```

A. 0

B. 5

C. 3

D. 7

Quiz 8: What is the output?

```
class Rectangle
 def initialize(h, w)
     @@h = h
     @w = w
 end
 def measure()
  return @@h + @w
 end
End
r = Rectangle.new(1,2)
s = Rectangle.new(3,4)
puts r.measure()
```

A. 0

B. **5**

C. 3

D. 7

Special Global Variables

- Ruby has a special set of global variables that are implicitly set by methods
- The most insidious one: $_
  - Last line of input read by gets or readline
- Example program

```
gets     # implicitly reads input line into $_
print    # implicitly prints out $_
```

- Using $_ leads to shorter programs
  - And confusion
  - We suggest you avoid using it

What is a Program?

- In C/C++, a program is...
  - A collection of declarations and definitions
  - With a distinguished function definition
    - int main(int argc, char *argv[]) { ... }
  - When you run a C/C++ program, it's like the OS calls main(...)
- In Java, a program is...
  - A collection of class definitions
  - With some class (say, MyClass) containing a method
    - public static void main(String[] args)
  - When you run java MyClass, the main method of class MyClass is invoked

A Ruby Program is...

- The class Object
  - When the class is loaded, any expressions not in method bodies are executed

defines a method of Object
(i.e., top-level methods belong to Object)

invokes self.sayN

invokes self.puts
(part of Object)

```
def sayN(message, n)
  i = 0
  while i < n
    puts message
    i = i + 1
  end
  return i
end

x = sayN("hello", 3)
puts(x)
```