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
  ➢ 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

<table>
<thead>
<tr>
<th>Object</th>
<th>Class (aka type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Integer</td>
</tr>
<tr>
<td>-3.30</td>
<td>Float</td>
</tr>
<tr>
<td>&quot;CMSC 330&quot;</td>
<td>String</td>
</tr>
<tr>
<td>String.new</td>
<td>String</td>
</tr>
<tr>
<td>['a', 'b', 'c']</td>
<td>Array</td>
</tr>
<tr>
<td>Integer</td>
<td>Class</td>
</tr>
</tbody>
</table>

- 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

```ruby
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", ... ]
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

```ruby
"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)
    ```ruby
    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
    ```ruby
    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 = "hello"
else
    x = nil
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?

\[
p = \text{nil} \\
x = 3 \\
\text{if } p \text{ then} \\
\quad x = \text{“hello”} \\
\text{else} \\
\quad x = \text{nil} \\
\text{end}
\]

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

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

```ruby
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

```javascript
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.
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

```ruby
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

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

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

• Use `{ }` for the empty hash
Quiz 2: What is the output?

```python
a = {"foo" => "bar"}
a[0] = "baz"
print a[1]
print a["foo"]
```

A. Error  
B. bar  
C. bazbar  
D. baznilbar
Quiz 2: What is the output?

```plaintext
a = {"foo" => "bar"}
a[0] = "baz"
print a[1]
print a["foo"]
```

A. Error
B. bar
C. bazbar
D. baznilbar
Quiz 3: What is the output?

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

A. Green
B. *(nothing)*
C. Blue
D. *Error*
Quiz 3: What is the output?

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

A. Green
B. (nothing)
C. Blue
D. Error
Quiz 4: What is the output?

\[
\begin{align*}
a &= [1,2,3] \\
a[1] &= 0 \\
a\text{.shift} \\
\text{print } a[1]
\end{align*}
\]

A. Error
B. 2
C. 3
D. 0
Quiz 4: What is the output?

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

A. Error
B. 2
C. 3
D. 0
Defining Your Own Classes

```ruby
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**: `Point` is defined in uppercase.
- **Constructor**: `initialize` method is defined to set the instance variables.
- **Instance Variables**: Prefixed with `@`.
- **Method**: `add_x` is a method with no arguments.
- **Instantiation**: Creating an instance of `Point` using `new`.
- **Invoking Method**: Calling `add_x` and `to_s` methods on the instance variable `p`.¸
Methods in Ruby

Methods are declared with `def...end`

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

List parameters at definition

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

May omit parens on call

Invoke method

Like print, but Adds newline

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:

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

- Methods can return multiple results (as an Array)

```ruby
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:

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

- A typical getter
- A typical setter

- Very common, so Ruby provides a shortcut:

  ```ruby
  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 runtime 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?

```ruby
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
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?

```ruby
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 3 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`

```ruby
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))
```

- `b.is_a? A`  `true`
- `b.instance_of? A`  `false`
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
Ruby has two kinds of global variables
- Class variables beginning with @@ (static in Java)
- Global variables across classes beginning with $

```ruby
class Global
  @@x = 0

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

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

```ruby
$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()
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()
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

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

x = sayN("hello", 3)
defines a method of Object
(i.e., top-level methods belong to Object)

invokes self.sayN

invokes self.puts
(part of Object)
```

CMSC 330 - Summer 2020