CS 416
Web Programming
Ruby on RAILS

Intro

Dr. Williams
Central Connecticut State University

Ruby

- Created in 1996
- Object oriented
- Cross platform
- Borrows from Python, Perl, Lisp
  - Initially created to get simplicity of Python without requiring OO if unnecessary
  - Yet everything is an Object
- Purely interpreted scripted language

Rails

- Created in 2005
- Rails is a development framework for Web-based applications
- Rails is written in Ruby and uses Ruby for its applications - Ruby on Rails (RoR)
- Based on MVC architecture for applications
- Principle theme:

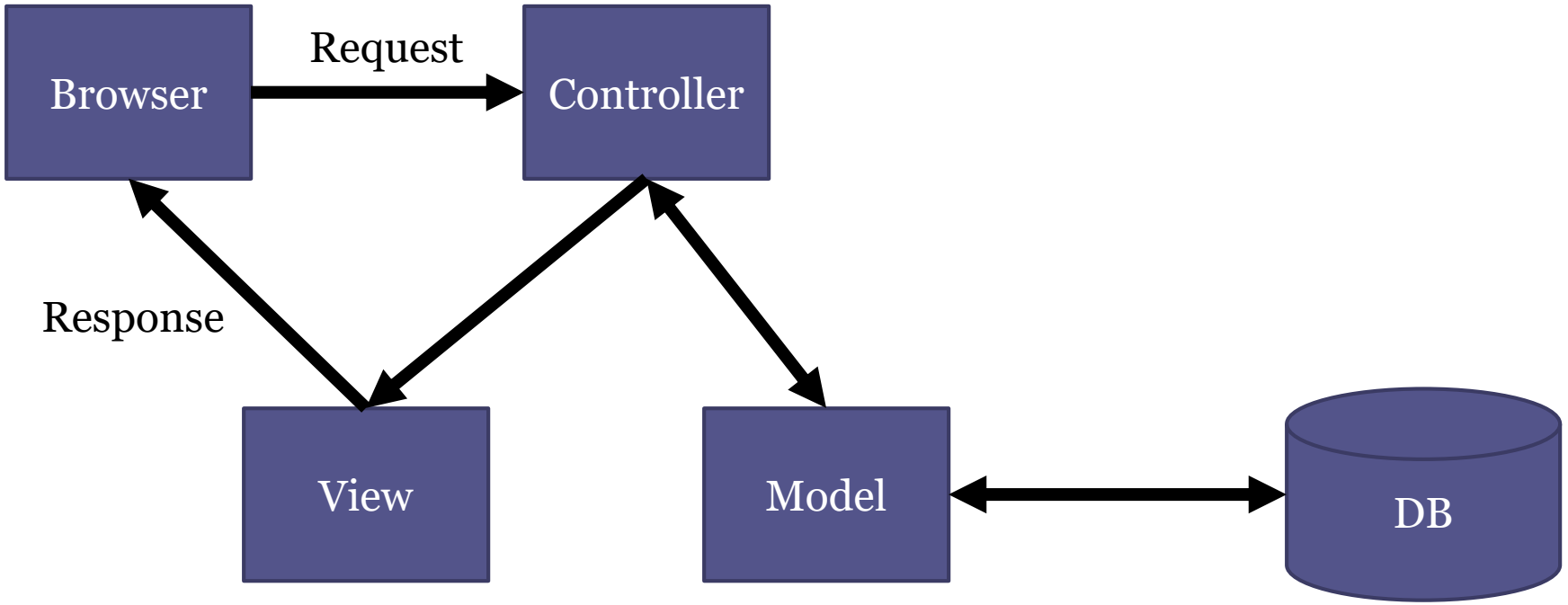
Convention over configuration

Developing in Rails

- Options:
  1. Lab has Rails installed on lab machines, note that command in class and those in book based on Unix command (with Windows version must add `ruby` in front of everything) – on Windows I would recommend either Atom (atom.io free) or RubyMine (currently $89, but students can get 1 year for free)
  2. Install it on your own machine or a Linux virtual machine – I would recommend install instructions at `gorails.com`, covers all operating systems
  3. Use cloud development environment - Cloud 9 (c9.io) – sets up Linux VM, IDE and very easy to use

For class I will be using Atom as my IDE and a VM with Ubuntu 14.04 or using Cloud 9

Rails flow



Rails cont.

- Rails sweet spot rapid prototyping
- In industry many companies use RoR for rapid prototyping, but change to compiled language for production system
  - Being interpreted it can be easy to unintentionally introduce code that is non-scalable -> several major Twitter outages blamed on scalability issues
  - However GitHub, Hulu, and Basecamp just a few that use it

Now the cart before the horse, why RoR…*the magic*

Rapid prototyping with Rails

- Creating a quick class directory stystem
- Create new application

```
rails new classdirectory

cd classdirectory
```

Create scaffolding

```
rails generate scaffold classroom
teacher:string grade:integer
classid:string

rails generate scaffold student
fullname:string phone:string
classid:string
```

- Note table will be named plural of object
- This will generate pages for all crud operations allowing you to quickly get up and running

Example continued

Now create the actual tables

```
rails db:migrate
```

Start the server

Local: `rails server`

C9: `rails server -b $IP -p $PORT`

Test if Rails working visit

`http://localhost:3000/`

Then visit

`http://localhost:3000/students`

Active record validation

- Creating the scaffolding created the model, now can add validation:

In models/classroom.rb

```
class Classroom < ActiveRecord::Base
  validates :teacher, :grade, :classid, :presence => true
  validates :teacher, length: { minimum: 2 }
  validates :grade, numericality: {greater_than: 0}
end
```

In models/student.rb

```
class Student < ActiveRecord::Base
  validates :phone, length:{in: 7..10,
    message:"Your phone number must have 7-10 digits" }
end
```

Add student listing to class

In controllers/classrooms_controller.rb

```
def show
    @students = Student.where(:classid =>
                          @classroom.classid)
end
```

_____ on view _____

In views/classrooms/show.html.erb

```
<table border="1">
<% @students.each do |student| %>
      <tr>
        <td><%= student.fullname %></td>
        <td><%= student.phone %></td>
      </tr>
  <% end %>
</table>
```

Generate a controller

Generate a controller

`rails generate controller` **`directory`** *`search results`*

- Directory – name of the controller
- Search, results – name of views controller will display

- *Generates controller, view, and adds name mapping*

Add basic search form

In views/directory/search.html.erb

```
<form action="/directory/results" method="GET">
  <input type="text" name="partial_name" />
  <input type="submit" value="search"/>
</form>
```

Add search

In views/directory/results.html.erb

```
<table border="1">
<% @students.each do |student| %>
      <tr>
        <td><%= student.fullname %></td>
        <td><%= student.phone %></td>
      </tr>
  <% end %>
</table>
```

In controllers/directory_controller.rb

```
  def results
    @partial_name = "%" + params[:partial_name] + "%"
    @students = Student.where("fullname like ?",
                              @partial_name)
  end
```

Open http://localhost:3000/directory/search

Bottom line

- If you are looking to throw together something quickly for proof of concept RoR can be great

Now, back to the basics…i.e. what you need to use it

The Ruby language

- A bit of a paradox in that everything is an object, but you don’t have to program things as objects

Scalar Types and Their Operations

- There are three categories of data types:
  - scalars, arrays, and hashes
- Two categories of scalars, numerics and strings
- - All numeric types are descendants of the
- `Numeric` class
- - Integers: `Fixnum` (usually 32 bits) and `Bignum`

Scalar types and their operations

- *Scalar Literals*

- An integer literal that fits in a machine word is a `Fixnum`

  - Other integer literals are `Bignum` objects

- Any numeric literal with an embedded decimal point or a following exponent is a `Float`

Scalar types cont.

- All string literals are `String` objects
- Single-quoted literals cannot include characters that are specified with escape sequences
  - Every thing between quotes taken exactly as is
- Double-quoted literals can include escape sequences and embedded variables *can be* interpolated

Variables

- Names of local variables begin with lowercase letters and are case sensitive
- Variables embedded in double-quoted literal strings are interpolated if they appear in braces and are preceded by a pound sign (#)

```
"The high was #{today_high}"
```

- Expressions can also be embedded in braces in double-quoted literals

```
"The total is #{quantity * cost}"
```

- Variables do not have types—they are not declared
- Assignment statement assign only object addresses

Interactive Ruby

- `irb` is an interactive interpreter for Ruby
  - Allows you to run lines individually to see individual execution
- Assignment

```
irb(main):011:0> mystr = "hello"
=> "hello"
irb(main):012:0> yourstr = "world"
=> "world"
irb(main):013:0> mystr = yourstr
=> "world"
irb(main):014:0> mystr = "hello"
=> "hello"
irb(main):015:0> yourstr
=> "world"
irb(main):016:0> mystr = yourstr
=> "world"
irb(main):017:0> mystr.replace("hello")
=> "hello"
irb(main):018:0> yourstr
=> "hello"
```

Useful string methods

All return new instance, unless **mutator operator** ! is used

Change cases
`capitalize,upcase, downcase, swapcase`

Strip whitespace
`strip, lstrip, rstrip`

`reverse` – reverses letters

`chop` – removes last character
`chomp` – removes new line from end if present

Tests

- "==" test equality of objects
  - `irb(main):022:0> 4 == 4.0`
  - `=> true`
- "equal?" test reference to same object
- "eql?" test variable **same type** and same value
  - `irb(main):021:0> 4.eql? 4.0`
  - `=> false`
- The <=> operator; it returns `0`, `1`, or `-1`

Input/Output

puts – writes to output
gets – reads from input

Simple program hello.rb

```
puts "What is your name?\n"
name = gets
!name.chomp
puts "Hi #{name}"
```

Selection statements

```
if control_expression
        statement_sequence
elsif control_expression
        statement_sequence
     ...
else
        statement_sequence
end
```

(also there is `unless`, inverse of if)

Select constructs

```
case expression
when value then
     statement_sequence
when value then
     statement_sequence
       …
[else
          statement_sequence]
end
```

The values could be expressions, ranges (e.g., `(1..10)`), class names, or regular expressions

Case statement

1. There is an implicit `break` at the end of every selectable segment

2. The value of the expression is compared with the when values, top to bottom, until a match is found

3. A different operator, ===, is used for the comparisons. If the value is a class name, it is a match if its class is the same as that of the expression or one of its superclasses;
   if the value is a regular expression, === is a simple pattern match

Case expression statement

```
leap =    case
          when year % 400 == 0 then true
          when year % 100 == 0 then false
          else year % 4 == 0
          end
```

Loops

```
while i < 11
    puts "#{i} "
    i+=1
end

for i in 1..10
    puts "#{i} "
end
```

Iterators

- Each iterator

```
[1,2,3,4,5,6,7,8,9,10].each {|value|
    print "#{value} "}
```

- Times iterator

```
10.times {|i|
    print "#{i} "}
```

- Upto and step iterators

```
1.upto(10) {|i|
    print "#{i} "}
```

1 2 3 4 5 6 7 8 9 10

```
1.step(10, 2) { |i|
    print "#{i} "}
```

1 3 5 7 9

Arrays

- Differences between Ruby arrays and those of other common languages:
  - Length is dynamic
  - An array can store different kinds of data
- Array Creation
  - Send new to the Array class

```
list1 = Array.new(100)
```

- Assign a list literal to a variable

```
list2 = [2, 4, 3.14159, "Fred", [] ]
```

```
for value in list
    sum += value
end
```

Associative arrays/hashes

- Two fundamental differences between arrays and hashes:
  1. Arrays use numeric subscripts; hashes use string values
  2. Elements of arrays are ordered and are stored in contiguous memory; elements of hashes are not
- Hash Creation
  - Send new to the Hash class

    ```
    my_hash = Hash.new
    ```

  - Assign a hash literal to a variable

    ```
    ages = ("Mike" => 14, "Mary" => 12)
    ```

- Element references – through subscripting

  ```
  ages["Mary"]
  ```

- Element are added by assignment

  ```
  ages["Fred"] = 9
  ```

Hashes cont.

- Element removal

```
ages.delete("Mike")
```

- - Hash deletion

```
ages = ()
```

or

```
ages.clear
```

- - Testing for the presence of a particular element

```
ages.has_key?("Scooter")
```

- - Extracting the keys or values

```
ages.keys
ages.values
```

Methods

- All Ruby subprograms are methods, but they can be defined outside classes

```
def method_name[(formal_parameters)]
  statement_sequence
end
```

- When a method is called from outside the class in which it is defined, it must be called through an object of that class

- When a method is called without an object reference, the default object is self

- When a method is defined outside any class, it is called without an object reference

Methods cont.

- Method names must begin with lowercase letters

- The parentheses around the formal parameters are optional

- Neither the types of the formal parameters nor that of the return type is given

- If the caller uses the returned value of the method, the call is in the place of an operand in an expression

- Scoping of variables same as other common languages

Method parameters

- All scalars passed by value
- Asterisks parameter can be used to specify an arbitrary number of parameters may be used

```
def fun2(sum, list, length = 10, *params)
    …
    sum = params[0] + 4
end
```

Classes

```
class Class_name
    …
end
```

- Class names must begin with uppercase letters
- The names of instance variables must begin with at signs (@)
- Each class implicitly has a constructor, new, which is called to create an instance
  - The new constructor calls the class initializer
- A class may have a single initializer, initialize
  - Initializes the instance variables
  - Parameters for initialize are passed to new

Classes cont.

- Classes are dynamic – subsequent definitions can include new members; methods can be removed with remove_method in subsequent definitions

- Access Control
  - All instance data has private access by default, and it cannot be changed

Classess access control

- If needed, external access to instance variables is provided with getters and setters

```
class My_class
  # Constructor
    def initialize
      @one = 1
      @two = 2
    end
  # A getter for @one
    def one
      @one
    end
  # A setter for @one
    def one=(my_one)
      @one = my_one
    end
End
```

- Shortcuts for getters and setters

```
  attr_reader :one, :two
  attr_writer :one
```

Classes access control cont.

Method access control: public, private, and protected
- Public and protected are as in Java, etc.
- There are two ways to specify access

1. Specify directly

```
private
      def meth1
      def meth2
protected
      def meth3
```

2. Following all method definitions in a class, call the access function, passing the method names as symbols

```
def meth1
def meth2
def meth3
private :meth1, meth2
protected: meth3
```

Classes cont.

- Inheritance

```
class My_Subclass < Base_class
```

- Modules

  - A way to collect related methods into an encapsulation

  - Access to a module is with include

    ```
    include Math
    ```

  - The methods in a module are mixed into those of the class that includes it – mixins

  - Provides the benefits of multiple inheritance