CMSC 330: Organization of Programming Languages

Introduction to Ruby

Introduction

- Ruby is an object-oriented, imperative scripting language

  - “I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That’s why I decided to design my own language.”

  - “I believe people want to express themselves when they program. They don’t want to fight with the language. Programming languages must feel natural to programmers. I tried to make people enjoy programming and concentrate on the fun and creative part of programming when they use Ruby.”

  – Yukihiro Matsumoto ("Matz")

Last Lecture

- Many types of programming languages
  - Imperative, functional, logical, OO, scripting

- Many programming language attributes
  - Clear, orthogonal, natural…

- Programming language implementation
  - Compiled, interpreted

Books on Ruby

- Earlier version of Thomas book available on web
  - See course web page
Applications of Scripting Languages

- Scripting languages have many uses
  - Automating system administration
  - Automating user tasks
  - Quick-and-dirty development

- Major application
  - Text processing

Output from Command-Line Tool

```
% wc *
271 674 5323 AST.c
100 392 3219 AST.h
117 1459 238788 AST.o
1874 5428 47461 AST_defs.c
371 884 9483 AST_utils.c
810 2328 24589 AST_print.c
640 3070 33530 AST_types.h
285 846 7081 AST_utils.h
59 274 2154 AST_utils.h
50 400 28756 AST_utils.o
866 2757 25873 Makefile
270 725 5578 Makefile.am
866 2743 27320 Makefile.in
38 175 1154 alloca.c
2035 4516 47721 aloctypes.c
86 350 3286 aloctypes.h
104 1051 66848 aloctypes.o
```

Climate Data for IAD in August, 2005

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6A</th>
<th>6B</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>MX</td>
<td>MIN</td>
<td>AVG</td>
<td>DEP</td>
<td>HDD</td>
<td>WTR</td>
<td>SWW</td>
<td>DPTH</td>
<td>SPD</td>
<td>SPD</td>
<td>DTR</td>
<td>DIR</td>
<td>MIN</td>
<td>PSBL</td>
<td>S-S</td>
<td>WX</td>
<td>SPB</td>
<td>DR</td>
</tr>
<tr>
<td>1</td>
<td>87</td>
<td>66</td>
<td>77</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0</td>
<td>2.5</td>
<td>9</td>
<td>200</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>18</td>
<td>12</td>
<td>210</td>
</tr>
<tr>
<td>2</td>
<td>92</td>
<td>67</td>
<td>80</td>
<td>4</td>
<td>0</td>
<td>15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0</td>
<td>3.5</td>
<td>10</td>
<td>10</td>
<td>M</td>
<td>M</td>
<td>3</td>
<td>18</td>
<td>17</td>
<td>320</td>
</tr>
<tr>
<td>3</td>
<td>93</td>
<td>69</td>
<td>81</td>
<td>5</td>
<td>0</td>
<td>16</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0</td>
<td>4.1</td>
<td>13</td>
<td>360</td>
<td>M</td>
<td>M</td>
<td>2</td>
<td>18</td>
<td>17</td>
<td>360</td>
</tr>
<tr>
<td>4</td>
<td>95</td>
<td>69</td>
<td>82</td>
<td>6</td>
<td>0</td>
<td>17</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0</td>
<td>3.6</td>
<td>9</td>
<td>310</td>
<td>M</td>
<td>M</td>
<td>3</td>
<td>18</td>
<td>12</td>
<td>290</td>
</tr>
<tr>
<td>5</td>
<td>94</td>
<td>73</td>
<td>84</td>
<td>8</td>
<td>0</td>
<td>19</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0</td>
<td>5.9</td>
<td>18</td>
<td>10</td>
<td>M</td>
<td>M</td>
<td>3</td>
<td>18</td>
<td>25</td>
<td>360</td>
</tr>
<tr>
<td>6</td>
<td>89</td>
<td>70</td>
<td>80</td>
<td>4</td>
<td>0</td>
<td>15</td>
<td>0.02</td>
<td>0.00</td>
<td>0.0</td>
<td>5.3</td>
<td>20</td>
<td>200</td>
<td>M</td>
<td>M</td>
<td>6</td>
<td>138</td>
<td>23</td>
<td>210</td>
</tr>
<tr>
<td>7</td>
<td>89</td>
<td>69</td>
<td>79</td>
<td>3</td>
<td>0</td>
<td>14</td>
<td>0.74</td>
<td>0.00</td>
<td>0.0</td>
<td>3.6</td>
<td>14</td>
<td>200</td>
<td>M</td>
<td>M</td>
<td>7</td>
<td>16</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>86</td>
<td>70</td>
<td>78</td>
<td>3</td>
<td>0</td>
<td>13</td>
<td>0.74</td>
<td>0.00</td>
<td>0.0</td>
<td>4.4</td>
<td>17</td>
<td>150</td>
<td>M</td>
<td>M</td>
<td>10</td>
<td>18</td>
<td>23</td>
<td>150</td>
</tr>
<tr>
<td>9</td>
<td>76</td>
<td>70</td>
<td>73</td>
<td>-2</td>
<td>0</td>
<td>8</td>
<td>0.19</td>
<td>0.00</td>
<td>0.0</td>
<td>4.1</td>
<td>9</td>
<td>90</td>
<td>M</td>
<td>M</td>
<td>9</td>
<td>18</td>
<td>13</td>
<td>90</td>
</tr>
<tr>
<td>10</td>
<td>87</td>
<td>71</td>
<td>79</td>
<td>4</td>
<td>0</td>
<td>14</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0</td>
<td>2.3</td>
<td>8</td>
<td>260</td>
<td>M</td>
<td>M</td>
<td>8</td>
<td>10</td>
<td>210</td>
<td></td>
</tr>
</tbody>
</table>

Raw Census 2000 Data for DC

```
u108_S,DC,000,01,000001,572059,72264,572059,572059,0,0,0,0,0,57
2059,175306,343213,000001,572059,572059,0,0,0,0,0,57
61,1560,14605,291,1638,10272,45015,16689,3152,446,125,92,20090,4389,572
059,268827,3362,3170,3241,3504,3286,3270,3475,3939,3647,3525,3044,2928,
2913,2769,2752,2933,2703,4056,5501,5217,4969,4699,3935,5363,73,
370675,115963,55603,60360,57949,129440,122518,3754,3168,42448,9967,4
638,14110,165698,61049,47694,13355,71578,60875,10703,33071,35686,7573,
8213,248590,108569,47694,60875,140021,115963,58050,21654,36396,57
913,10355,4065,6280,47558,25229,23239,24058,13355,10703,70088,65737,371
12,21742,12267,9475,7923,2573,2314,760,28625,8207,7469,738,19185,18172,
1013,1233,4351,3610,741,248590,199456,94221,46274,21443,42831,47947,870
5,3979,4726,39242,25175,14067,105235,82928,22073,49134,21742,14776,211,
11565,3966,1650,85,1564,316,54,8262,27392,25641,1751,248590,115963,499
9,22466,26165,24062,16529,12409,7594,1739,132627,11670,32445,23255,2166
1,16234,12795,10563,4034,248590,115963,48738,28914,19259,10312,4748,399
2,132627,108569,19284,7213,1209,509,218,125
```

A Simple Example

Let’s start with a simple Ruby program

```ruby
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

```ruby
% ruby -w ruby1.rb
42
%
```

Language Basics

comments begin with #, go to end of line

```ruby
# This is a ruby program
x = 37
y = x + 5
print(y)
print("\n")
```

variables need not be declared

line break separates expressions (can also use “;” to be safe)

Run Ruby, Run

There are several ways to run a Ruby program

- `ruby -w filename` – execute script in `filename`
  - tip: the `-w` will cause Ruby to print a bit more if something bad happens
  - Ruby filenames should end with `.rb` extension
- `irb` – launch interactive Ruby shell
  - Can type in Ruby programs one line at a time, and watch as each line is executed
    ```ruby
    irb(main):001:0> 3+4
    7
    ```
  - Can load Ruby programs via `load` command
    - Form: `load string`
    - String must be name of file containing Ruby program
    - E.g.: `load ‘foo.rb’`

Ruby 1.9.3 is installed on linuxlab, Grace clusters

Run Ruby, Run (cont.)

- `fxri` – launch standalone interactive Ruby shell

![fxri.png](attachment://fxri.png)
Run Ruby, Run (cont.)

- Suppose you want to run a Ruby script as if it were an executable (e.g. “double-click”, or as a command)
  - Windows
    - Must associate .rb file extension with ruby command
    - If you installed Ruby using the Windows installer, this was done automatically
    - The Ruby web site has information on how to make this association

Run Ruby, Run (cont.)

- Suppose you want to run a Ruby script as if it were an executable (cont.)
  - *nix (Linux / Unix / etc.)
    - The first line (“shebang”) tells the system where to find the program to interpret this text file
    - Must chmod u+x filename first, or chmod a+x filename so everyone has exec permission
    - Warning: Not very portable: Depends on location of Ruby interpreter
      - /usr/local/bin/ruby vs. /usr/bin/ruby vs. /opt/local/bin/ruby etc.

Creating Ruby Programs

- As with most programming languages, Ruby programs are text files.
  - Note: there are actually different versions of “plain text”! E.g. ASCII, Unicode, Utf-8, etc.
  - You won’t need to worry about this in this course.
- To create a Ruby program, you can use your favorite text editor, e.g.
  - notepad++ (free, much better than notepad)
  - emacs (free, infinitely configurable)
  - vim
  - Eclipse (see web page for plugin instructions)
  - Many others

Explicit vs. Implicit Declarations

- Java and C/C++ use explicit variable declarations
  - Variables are named and typed before they are used
    - int x, y; x = 37; y = x + 5;
- In Ruby, variables are implicitly declared
  - First use of a variable declares it and determines type
    - x = 37; y = x + 5;
    - x, y exist, will be integers
  - Ruby allows multi-assignment, too
    - x,y = 37, 5; y += x
      - x,y = 37,x+5 would have failed; x was not yet assigned
Tradeoffs?

<table>
<thead>
<tr>
<th>Explicit Declarations</th>
<th>Implicit Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>More text to type</td>
<td>Less text to type</td>
</tr>
<tr>
<td>Helps prevent typos</td>
<td>Easy to mistype variable name</td>
</tr>
<tr>
<td>Forces programmer to document types</td>
<td>Variable not held to a fixed type (could imagine variable declarations without types)</td>
</tr>
</tbody>
</table>

Methods in Ruby

Methods are declared with `def...end`...
- List parameters at definition
- May omit parens on call
- Invoke method
- Like `print`, but adds newline

<table>
<thead>
<tr>
<th>def sayN(message, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
</tr>
<tr>
<td>while i &lt; n</td>
</tr>
<tr>
<td>puts message</td>
</tr>
<tr>
<td>i = i + 1</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>return i</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>x = sayN(&quot;hello&quot;, 3)</td>
</tr>
<tr>
<td>puts(x)</td>
</tr>
</tbody>
</table>

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)

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 a list)

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

Terminology

- Formal parameters
  - Parameters used in the body of the method
  - `def sayN(message, n)` in our example

- Actual parameters
  - Arguments passed in to the method at a call
  - `x = sayN("hello", 3)` in our example
Style

- Names of methods that return a boolean 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

Control Statements in Ruby

- A control statement is one that affects which instruction is executed next.
  - We’ve seen two so far in Ruby:
    - `while` and `method call`
  - Ruby also has conditionals:
    ```ruby
    if grade >= 90 then
      puts "You got an A"
    elsif grade >= 80 then
      puts "You got a B"
    elsif grade >= 70 then
      puts "You got a C"
    else
      puts "You’re not doing so well"
    end
    ```

Ruby Conditionals Must End!

- All Ruby conditional statements must be terminated with the `end` keyword.

Examples
  - `if grade >= 90 then
    puts "You got an A"
  end`
  - `if grade >= 90 then
    puts "You got an A"
  else
    puts "No A, sorry"
  end`

What is True?

- The guard of a conditional is the expression that determines which branch is taken.
  ```ruby
  if grade >= 90 then
    ... Guard ...
  end
  ```

- The true branch is taken if the guard evaluates to anything except:
  - `false`
  - `nil`

- Warning to C programmers: `0` is not `false`!
Yet More Control Statements in Ruby

- **unless** *cond*  *then*  *stmt-f*  *else*  *stmt-t*  *end*
  - Same as “if not *cond* then *stmt-t* else *stmt-f* end”

- **until**  *cond*  *body*  *end*
  - Same as “while not *cond* body end”

Using If and Unless as Modifiers

- Can write `if` and **unless** after an expression
  - puts "You got an A" if grade >= 90
  - puts "You got an A" unless grade < 90

- Why so many control statements?
  - Is this a good idea? Why or why not?
    - **Good**: can make program more readable, expressing programs more directly. In natural language, many ways to say the same thing, which supports brevity and adds style.
    - **Bad**: many ways to do the same thing may lead to confusion and hurt maintainability (if future programmers don’t understand all styles)

Classes and Objects

- 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**

Everything is an Object

- In Ruby, everything is an object
  - `(-4).abs`
    - integers are instances of 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
Objects and Classes

- Objects are data
- Classes are types (the kind of data which things are)
- But in Ruby, classes themselves are objects!

<table>
<thead>
<tr>
<th>Object</th>
<th>Class (aka type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Fixnum</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>Fixnum</td>
<td>Class</td>
</tr>
</tbody>
</table>

Fixnum, Float, and String are objects of type Class
  • So is Class itself!

The nil Object

- Ruby uses a special object nil
  • All uninitialized fields set to nil (@ prefix used for fields)
    `irb(main):004:0> @x => nil`
  • Like NULL or 0 in C/C++ and null in Java
- nil is an object of class NilClass
  • It’s 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`

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`:
    ```ruby
    => %w[send name class_eval object_id new autoload ...]
    ```

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)
  puts(x)
  ```
  defines a method of `Object`
  invokes `self.sayN`
  invokes `self.puts` (part of `Object`)

Ruby is Dynamically Typed

- Recall we don’t declare types of variables
  ```ruby
  x = 3; x.foo
  NoMethodError: undefined method ‘foo’ for 3:Fixnum
  ```
- We say that Ruby is dynamically typed
  - Types are determined and checked at run time
  ```ruby
  # Ruby
  x = 3
  x = "foo"  # gives x a new type
  ```
- Compare to C, which is statically typed
  ```c
  /* C */
  x = 3;
  x = "foo"; /* not allowed */
  ```

Types in Java and C++

- Are Java and C++ statically or dynamically typed?
  - A little of both
  - Many things are checked statically
    ```java
    Object x = new Object();
    x.println("hello");  // No such method error at compile time
    ```
  - But other things are checked dynamically
    ```java
    Object o = new Object();
    String s = (String) o;  // No compiler warning, fails at run time
    // (Some Java compilers may be smart enough to warn about above cast)
    ```

Tradeoffs?

<table>
<thead>
<tr>
<th>Static types</th>
<th>Dynamic types</th>
</tr>
</thead>
<tbody>
<tr>
<td>More work when coding</td>
<td>Less work when coding</td>
</tr>
<tr>
<td>Helps prevent some subtle errors</td>
<td>Can use objects incorrectly and not discover until run time</td>
</tr>
<tr>
<td>Fewer programs type check</td>
<td>More programs type check</td>
</tr>
</tbody>
</table>
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

Standard Library: 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
    x = a[0]; a[1] = 37
- Arrays are growable
  - Increase in size automatically as you access elements
    irb(main):001:0> b = []; b[0] = 0; b[5] = 0; puts b.inspect
    [0, nil, nil, nil, nil, 0]
  - [] is the empty array, same as Array.new

Standard Library: Arrays (cont.)

- 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]
- Can use arrays to 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"

Iterating Through Arrays

- It's easy to iterate over an array with while
  a = [1,2,3,4,5]
  i = 0
  while i < a.length
    puts a[i]
    i = i + 1
  end
- Looping through all elements of an array is very common
  - And there’s a better way to do it in Ruby
Iteration and Code Blocks

- The **Array** class also has an **each** method
  - Takes a code block as an argument

```ruby
g = [1, 2, 3, 4, 5]
g.each { |x| puts x }
```

- We’ll consider code blocks generally a bit later

Ranges

- **1..3** is an object of class **Range**
  - Integers between 1 and 3 inclusively

- **1…3** also has class **Range**
  - Integers between 1 and 3 but not including 3 itself.

- Not just for integers
  - ‘a’..'z' represents the range of letters ‘a’ to ‘z’
  - **1.3...2.7** is the **continuous** range [1.3, 2.7)
    - `(1.3...2.7).include? 2.0 # => true`

- Discrete ranges offer the **each** method to iterate
  - And can convert to an array via **to_a**; e.g., (1..2).to_a

Other Useful Control Statements

```ruby
for elt in [1, "math", 3.4]
  puts elt.to_s
end
```

```ruby
while i>n
  break
end
```

```ruby
for i in (1..3)
  puts i
end
```

```ruby
(1..3).each { |elt|
  puts elt
}
```

```ruby
IO.foreach(filename)
  { |x|
    puts x
  }
end
```

More Data-driven Control Statements

Ruby function to print all even numbers from 0 up to (but not including) some given number **x**

```ruby
def even(x)
  for i in (0...x)
    if i % 2 == 0
      puts i
    end
  end
end
```

```ruby
def even(x)
  x.times { |i| if i % 2 == 0
    puts i
  end
end
```

```ruby
def even(x)
  0.upto(x-1) { |i| if i % 2 == 0
    puts i
  end
end
```
Standard Library: Hash

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

- Elements are referred to using [] like array elements, but Hash.new is the Hash constructor

```ruby
titaly[“population”] = 58103033
titaly[“continent”] = “europe”
titaly[1861] = “independence”
```

Hash (cont.)

- Hash methods
  - values returns array of a hash’s values (in some order)
  - keys returns an array of a hash’s keys (in some order)

- Iterating over a hash
  ```ruby
titaly.keys.each { |k|
    print “key: ”, k, “ value: ”, italy[k]
  }
```

  ```ruby
titaly.each { |k,v|
    print “key: ”, k, “ value: ”, v
  }
```

Hash (cont.)

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
credits[“cmsc311”] = 3
```

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)
p.puts(p.to_s)
```
No Access To Internal State

- 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
  ```

- 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
  - 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

Classes and Objects in Ruby (cont.)

- Recall classes begin with an uppercase letter
- `inspect` converts any instance to a string

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

- The `to_s` method can be invoked implicitly
  - Could have written `puts(p)`
    - Like Java’s `toString()` methods

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
  ```

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

  ```ruby
  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

Mixins

- Another form of code reuse is “mix-in” inclusion
  - `include A` “inlines” A’s methods at that point
  - Referred-to variables/methods captured from context
  - In effect: it adds those methods to the current class

Global Variables in Ruby

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

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

```ruby
gets # implicitly reads input line into $_
puts($_) # implicitly prints out $_
```

- Using `$_` leads to shorter programs
  - And confusion
  - We suggest you avoid using it
Creating Strings in Ruby

Substitution in double-quoted strings with `{ }`

• course = "330"; msg = "Welcome to #{course}"
• "It is now #{Time.now}".
• The contents of `{ }` may be an arbitrary expression
• Can also use single-quote as delimiter
  ➢ No expression substitution, fewer escaping characters

Here-documents

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

Ruby also has `printf` and `sprintf`

• printf("Hello, \%s\n", name);
• sprintf("%d: \%s", count, Time.now)
  ➢ Returns a string

The `to_s` method returns a `String` representation of a class object

Standard Library: 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 except newline at end of line removed
• `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
• "r1\nr2\nr3\nr4".each_line { |rec| puts rec }
  ➢ Apply code block to each newline-separated substring

Consider these three examples again

• All involve searching in a string for a certain pattern
• What if we want to find more complicated patterns?
  ➢ Find first occurrence of "a" or "b"
  ➢ Split string at tabs, spaces, and newlines

```
Regular Expressions!
```
Object Copy vs. Reference Copy

- Consider the following code
  - Assume an object/reference model like Java or Ruby
    - Or even two pointers pointing to the same structure
    
```java
x = "groundhog" ; y = x
```

- Which of these occur?

<table>
<thead>
<tr>
<th>x (reference)</th>
<th>&quot;groundhog&quot; (object)</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>&quot;groundhog&quot; (object)</td>
</tr>
</tbody>
</table>

| Object copy | Reference copy |

Object Copy vs. Reference Copy (cont.)

- For
  - Ruby and Java would both do a reference copy
    
```java
x = "groundhog" ; y = x
```

- But for
  - Ruby would cause an object copy
  - Unnecessary in Java since Strings are immutable

Physical vs. Structural Equality

- Consider these cases again:

<table>
<thead>
<tr>
<th>x (reference)</th>
<th>&quot;groundhog&quot; (object)</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>&quot;groundhog&quot; (object)</td>
</tr>
</tbody>
</table>

- If we compare x and y, what is compared?
  - The references, or the contents of the objects they point to?
- If references are compared (physical equality)
  - the first would return false but the second true
- If objects are compared both would return true

String Equality

- In Java, \( x == y \) is physical equality, always
  - Compares references, not string contents
- In Ruby, \( x == y \) for strings uses structural equality
  - Compares contents, not references
    - \( == \) is a method that can be overridden in Ruby!
    - To check physical equality, use the `equal?` method
      - Inherited from the `Object` class
- It’s always important to know whether you’re doing a reference or object copy
  - And physical or structural comparison
Comparing Equality

<table>
<thead>
<tr>
<th>Language</th>
<th>Physical equality</th>
<th>Structural equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td><code>a == b</code></td>
<td><code>a.equals(b)</code></td>
</tr>
<tr>
<td>C</td>
<td><code>a == b</code></td>
<td><code>*a == *b</code></td>
</tr>
<tr>
<td>Ruby</td>
<td><code>a.equal?(b)</code></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td>Ocaml</td>
<td><code>a == b</code></td>
<td><code>a = b</code></td>
</tr>
<tr>
<td>Python</td>
<td><code>a is b</code></td>
<td><code>a == b</code></td>
</tr>
<tr>
<td>Scheme</td>
<td><code>(eq? a b)</code></td>
<td><code>(equal? a b)</code></td>
</tr>
<tr>
<td>Visual Basic .NET</td>
<td><code>a is b</code></td>
<td><code>a = b</code></td>
</tr>
</tbody>
</table>

Summary

- **Scripting languages**
- **Ruby language**
  - Implicit variable declarations
  - Dynamic typing
  - Many control statements
  - Classes & objects
  - Strings