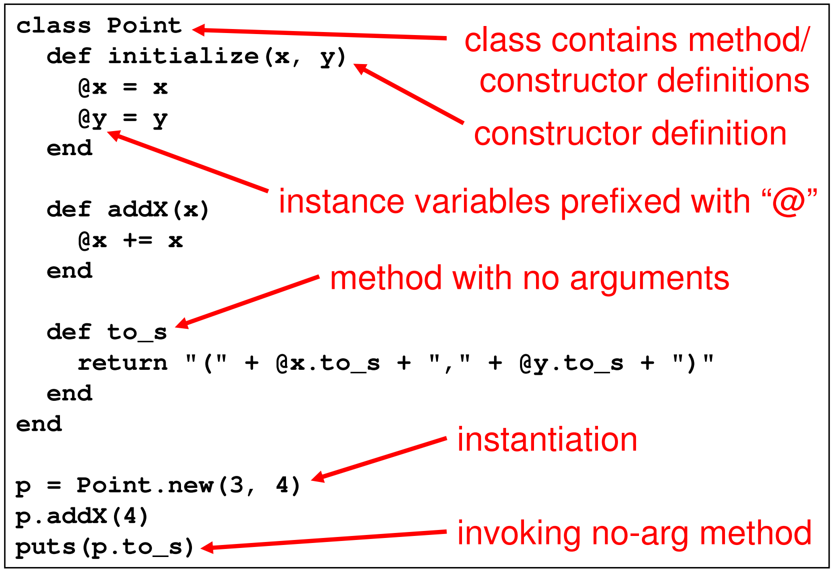
CMSC 330: Organization of Programming Languages

Ruby, Part 2

Classes and Objects in Ruby



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

  def addX(x)
    @x += x
  end

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

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

class contains method/ constructor definitions

constructor definition

instance variables prefixed with “@”

method with no arguments

instantiation

invoking no-arg method

Notes For Java Programmers

- Ruby does not support method overloading
  - A typical Java class might have two or more constructors
  - Since Ruby does not support method overloading there can only be one initialize method in a class
- Ruby does issue an exception or warning if classes defines more than one initialize method, but the last one defined will be the valid one

Classes and Objects in Ruby (cont’d)

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

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

- Instance variables are prefixed with @
  - compare to local variables with no prefix
  - *cannot be accessed outside of class*
- The to_s method can be invoked implicitly, like Java’s toString() methods
  - could have written puts(p)

Inheritance

- Recall that every class inherits from Object

```
class A
  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

super() in Ruby

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

Global Variables in Ruby

- Ruby has two kinds of global variables
  - class variables beginning with @@
  - 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

Special Global Variables

- Ruby has a bunch of global variables that are implicitly set by methods
- The most insidious one: $_
  - default method return, argument in many cases
- Example:

```
gets    # implicitly reads input into $_
print   # implicitly writes $_
```

- Using $_ leads to shorter programs (and confusion)
  - it's recommended you avoid using it

Creating Strings in Ruby

- Expression substitution in double-quoted strings with #{}

```
course = "330"; msg = "Welcome to #{course}"
"It is now #{Time.new}"
```

- Note: can also use single-quote as delimiter- no expression substitution, fewer escaping characters
- *Here-documents*

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

Substitution in Ruby Strings

Writing **elt** as **#{elt}** makes it clear that it is a variable to be evaluated, not a literal word to be printed. This is a cleaner way to express output; it builds a single string and presents it as a single argument to **puts**.

```
irb(main):001:0> for elt in [100,-9.6,"pickle"]
irb(main):002:1     puts "#{elt}\t(#{elt.class})"
irb(main):003::1   end
100     (Fixnum)
-9.6    (Float)
pickle  (String)
```

Creating Strings in Ruby (cont'd)

- 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

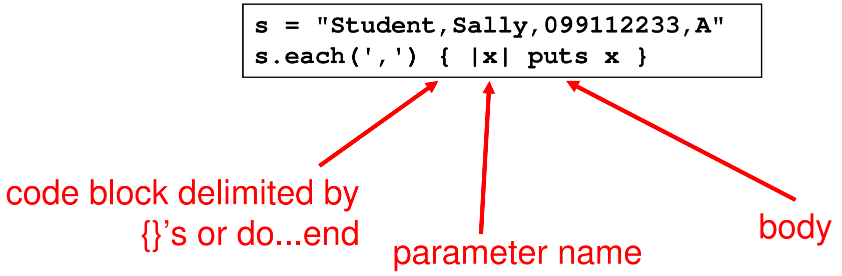
- The String class has many useful methods
  - s.length   # length of string
  - s = "A line\n"; s.chomp   # returns "A line"
    - return new string with s's contents except newline at end of line removed
  - s = "A line\n"; s.chomp!
    - destructively removes terminating newline from s
    - *convention:* methods ending in ! modify the object
    - *another convention:* methods ending in ? observe the object
  - s = "A line \n "; s.rstrip!
    - removes all trailing whitespace
  - "r1\tr2\t\tr4".each("\t") { |rec| puts rec }
    - apply code block to each tab-separated substring

Standard Library: String (cont'd)

- "hello".index("l", 0)
  - return index of the first occurrence of string in s, starting at n
- "hello".sub("h", "j")
  - replace first occurrence of "h" by "j" in string
  - use gsub ("global" sub) to replace all occurrences
- "r1\tr2\t\tr3".split("\t")
  - return array of substrings delimited by tab
  - "delimiter" = symbol used to denote boundaries
- s1 == s2   # compares string contents

Breaking up strings

- The each method and a *code block* applies the code block to every part of the string between a specified delimiter



```
s = "Student,Sally,099112233,A"
s.each(',') { |x| puts x }
```

So What Are Code Blocks?

- A code block is just a special kind of method
  - { |y| x = y + 1; puts x }
    is almost the same as
    def m(y) x = y + 1; puts x end
- The each method takes a code block as an argument; this is called *higher-order programming*
  - In other words, methods take other methods as arguments
  - We'll see a lot more of this in OCaml
- We'll see other library classes with each methods
  - And other methods that take code blocks as arguments

Using Yield To Call Code Blocks

- Your methods can be called with codes block too
  - Inside the method, the block is called with yield
- After the code block completes control returns to the caller after the yield instruction

```
def countx(x)
  for i in (1..x)
     puts i
     yield
  end
end

countx(4) { puts "foo" }
```

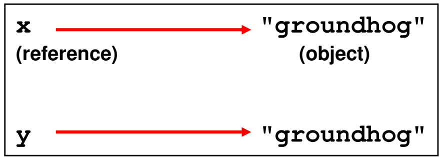
```
1
foo
2
foo
3
foo
4
foo
```

Object Copy vs. Reference Copy

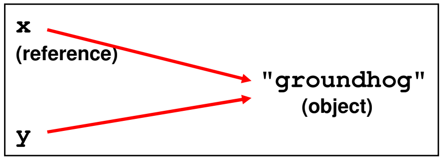
- Suppose we have something like the following in a language with an object/reference model like Java or Ruby (or even if two pointers point to data structures in a language like C):

```
x = "groundhog" ; y = x
```

- Which of these occurs?



object copy



reference copy

Object copy vs. Reference Copy, con't.

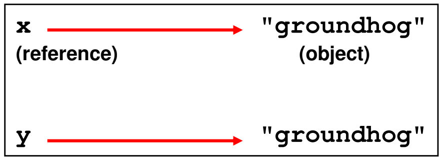
- Ruby and Java would both do a reference copy in this case
- But this Ruby example would cause object copy:

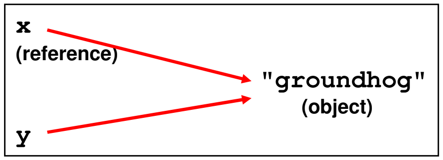
```
x = "groundhog"
y = String.new(x)
```

- Is this necessary in Java?

Physical vs. Structural Equality

- Consider these cases again:





- If we compare x and y, what's 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
- In Ruby, == compares objects (structural equality)

String Equality

- In Java, x == y is always physical equality
  - 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

Language	Physical equality	Structural equality
Java	**a == b**	**a.equals(b)**
C	**a == b**	***a == *b**
Ruby	**a.equal?(b)**	**a == b**
Ocaml	**a == b**	**a = b**
Python	**a is b**	**a == b**
Scheme	**(eq? a b)**	**(equal? a b)**
Visual Basic .NET	**a Is b**	**a = b**

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: Array (cont'd)

- Arrays can also shrink
  - contents shift left when you delete elements

    ```
    a = [1, 2, 3, 4, 5]
    a.delete_at(3)          # delete at subscript 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"
  ```

  - to model a stack push and pop can be used; unshift and pop will model a queue

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, which also uses a code block

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

```
a = [1,2,3,4,5]
sum = 0
a.each { |x| sum = sum + x }
printf("sum is %d\n", sum)
```

More Examples of Code Blocks

```
3.times { puts "hello"; puts "goodbye" }
5.upto(10) { |x| puts(x + 1) }
[1,2,3,4,5].find { |y| y % 2 == 0 }
[5,4,3].collect { |x| -x }
```

- n.times runs code block n times
- n.upto(m) runs code block for integers n..m
- a.find returns first element x of array such that the block returns true for x
- a.collect applies block to each element of array and returns new array

Another Example of Code Blocks

```
File.open("test.txt", "r") do |f|
  f.readlines.each { |line| puts line }
end
```

- open method takes code block with file argument
  - file automatically closed after block executed
- readlines reads all lines from a file and returns an array of the lines read; use each to iterate