

CMSC 330: Organization of Programming Languages

Inheritance, Mixins, Code Blocks,
Equality

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

No Outside Access To Internal State

- ▶ 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 1: What is the output?

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

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

Quiz 1: What is the output?

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

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

Quiz 2: What is the output?

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

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

Quiz 2: What is the output?

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

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

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
```

Quiz 6: 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
- B. Dirty, no good Billy the kid
- C. Billy the Kid
- D. *Error*

Quiz 6: 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
- B. Dirty, no good Billy the kid**
- 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 7: What is the output?

```
class Animal
  def initialize(h, w)
    @@h = h
    @w = w
  end
  def measure()
    return @@h + @w
  end
end
giraffe = Animal.new(1,2)
elephant = Animal.new(3,4)
puts giraffe.measure()
```

- A. 0
- B. 3
- C. 5
- D. 7

Quiz 7: What is the output?

```
class Animal
  def initialize(h, w)
    @@h = h
    @w = w
  end
  def measure()
    return @@h + @w
  end
end
End
giraffe = Animal.new(1,2)
elephant = Animal.new(3,4)
puts giraffe.measure()
```

- A. 0
- B. 3
- C. 5**
- D. 7

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
- ▶ To define a mixin, use `module`, not `class`

```
module Doubler
  def double
    self +self
  end
end
```

```
#include the mixin in a class
class Integer
  include Doubler
end
```

```
10.double => 20
```

Mixins

```
module Doubler
  def double
    self +self
  end
end
```

```
class Integer
  include Doubler
end
```

```
10.double => 20
```

```
class String
  include Doubler
end
```

```
"hello".double => "hellohello"
```


Mixin method lookup rules:

- ▶ When you call method **m** of class **C**
 1. look if **class C** has method **m**
 2. **mixin** in class **C**
 3. if multiple mixins included, **later mixin** shadows early mixin
 4. **C's superclass**
 5. **C's superclass mixin**
 6. **C's superclass's superclass**
 7. ...

Mixin example 1

```
module M1
  def hello
    "M1 hello"
  end
end

module M2
  def hello
    "M2 hello"
  end
end
```

```
class A
  include M1
  include M2
  def hello
    "A hello"
  end
end
```

```
a = A.new
a.hello #class A has a method hello and it is called for
a.hello # returns A hello
a.class.ancestors
=> [A, M2, M1, Object, Kernel, BasicObject]
```

Mixin example 2

```
module M1
  def hello
    "M1 hello"
  end
end
```

```
module M2
  def hello
    "M2 hello"
  end
end
```

```
class A
  include M1
  include M2
end
```

- class A does not have a method hello. look for the method hello from mixin.
- Both M1 and M2 have a method hello. M2's hello shadows M1's hello method.

```
a = A.new
a.hello    => returns M2 hello
a.class.ancestors
=> [A, M2, M1, Object, Kernel, BasicObject]
```

Mixin example 3

```
module M1
  def hello
    "m1 says hello, " + super
  end
  def what
    "Mary"
  end
end
class A
  def hello
    "A says hello, " + what
  end
  def what
    "Alice"
  end
end
class B < A
  include M1
  def hello
    "B says hello, " + super
  end
  def what
    "Bob"
  end
end
```

```
b = B.new
B.ancestors=>
[B, M1, A, Object, Kernel,
BasicObject]

b.hello=>
B says hello, m1 says hello, A says
hello, Bob
```

B's hello is called. super called B's superclass M1's hello. super in M1's hello called hello in superclass A. At the end, the "what" method of the current object "b" is called.

Mixins: Comparable

```
class OneDPoint
  attr_accessor :x
  include Comparable
  def <=>(other) # used by Comparable
    if @x < other.x then return -1
    elsif @x > other.x then return 1
    else return 0
    end
  end
end
```

```
p = OneDPoint.new
p.x = 1
q = OneDPoint.new
q.x = 2
x < y # true
puts [y,x].sort
# prints x, then y
```

Code Blocks

- ▶ A **code block** is a piece of code that is invoked by another piece of code
- ▶ Code blocks are useful for encapsulating repetitive computations

Array Iteration with Code Blocks

- ▶ The `Array` class has an `each` method
 - Takes a code block as an argument

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

code block delimited by
{ }'s or do...end

parameter name
(optional)

body

More Examples of Code Block Usage

- ▶ Sum up the elements of an array

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

- ▶ Print out each segment of the string as divided up by commas (commas are printed trailing each segment)

- Can use any delimiter

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

(“delimiter” = symbol used to denote boundaries)

Yet 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 (`a.collect!` modifies the original)

Still Another Example of Code Blocks

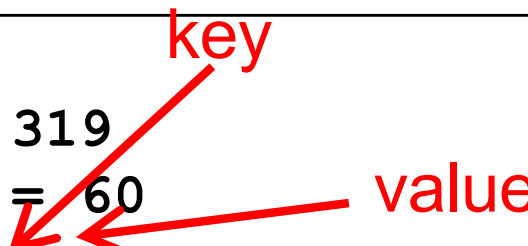
```
File.open("test.txt", "r") do |f|  
  f.readlines.each { |line| puts line }  
end
```

alternative syntax: `do ... end` instead of `{ ... }`

- `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
- Can do something similar on strings directly:
- `"r1\nr2\n\nr4".each_line { |rec| puts rec }`
 - Apply `code block` to each newline-separated substring

Code Blocks for Hashes

```
population = {}
population["USA"] = 319
population["Italy"] = 60
population.each { |c,p|
  puts "population of #{c} is #{p} million"
}
```



- ▶ Can iterate over keys and values separately

```
population.keys.each { |k|
  print "key: ", k, " value: ", population[k]
}

population.values.each { |v|
  print "value: ", v
}
```

Using Yield To Call Code Blocks

- ▶ Any method can be called with a code block
 - 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
```

Using Yield to Call Code Blocks

Ruby methods receive an implicit code block

```
def do_it_twice
  return "No block" unless block_given?
  yield
  yield
end
```

```
do_it_twice {puts "hello"}
=>
hello
hello
```

Code Block is not an Object

- ▶ Proc makes an object out of code blocks
 - `t = Proc.new{|x| x+2}`
- ▶ a method that receives a proc object

```
def say(p)
  p.call 10
end
```

```
say(t) => return 12
```

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 a parameter
 - 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
 - As we saw, your methods can use code blocks too!

Mixins: Enumerable

```
class MyRange
  include Enumerable #map,select,
inject, collect, find
  def initialize(low,high)
    @low = low      #(2,8)
    @high = high
  end
  def each
    i=@low
    while i <= @high
      yield i
      i=i+1
    end
  end
end
```


Quiz 4: What is the output

```
a = [5,10,15,20]
a.each { |x| x = x*x }
puts a[1]
```

- A. 10
- B. 100
- C. (Nothing)
- D. *Error*

Quiz 4: What is the output

```
a = [5,10,15,20]
a.each { |x| x = x*x }
puts a[1]
```

- A. 10
- B. 100
- C. (Nothing)
- D. *Error*

Quiz 5: What is the output

```
def myFun(x)
  yield x
end
myFun(3) { |v| puts "#{v} #{v*v}" }
```

- A. 3
- B. 3 9
- C. 9 81
- D. 9 nil

Quiz 5: What is the output

```
def myFun(x)
  yield x
end
myFun(3) { |v| puts "#{v} #{v*v}" }
```

- A. 3
- B. 3 9
- C. 9 81
- D. 9 nil

Ranges

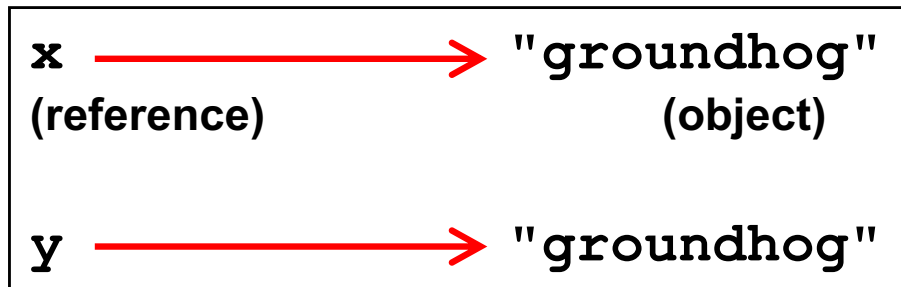
- ▶ `1..3` is an object of class `Range`
 - Integers between 1 and 3 inclusively
- ▶ `1...3` also has class `Range`
 - Integers between 1 and 3 *but not including 3 itself*.
- ▶ Not just for integers
 - `'a'..'z'` represents the range of letters 'a' to 'z'
 - `1.3...2.7` is the *continuous* range `[1.3,2.7)`
 - `(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`

Object Copy vs. Reference Copy

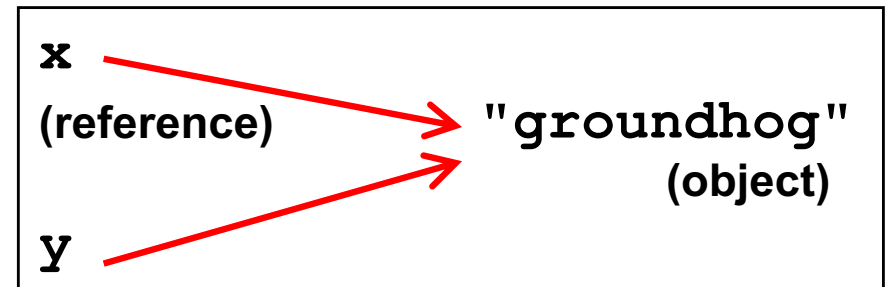
- ▶ Consider the following code
 - Assume an object/reference model like Java or Ruby
 - Or even two pointers pointing to the same structure

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

- ▶ Which of these occur?



Object copy



Reference copy

Object Copy vs. Reference Copy (cont.)

▶ For

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

- Ruby and Java would both do a reference copy

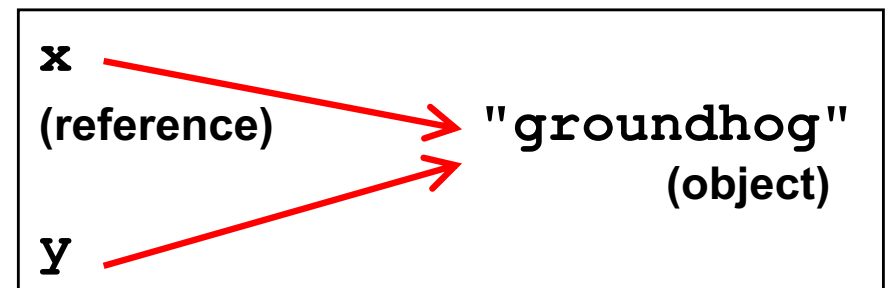
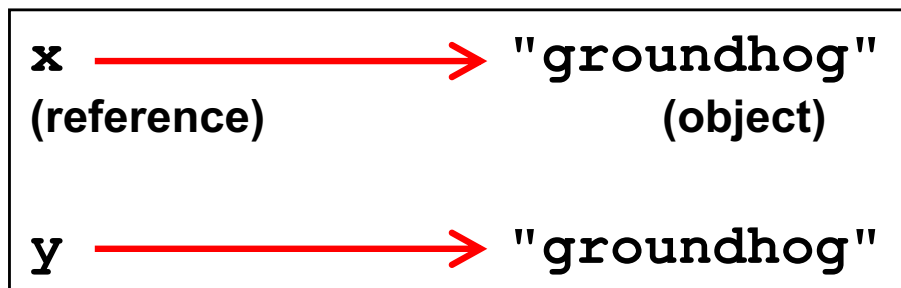
▶ But for

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

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

Physical vs. Structural Equality

- ▶ Consider these cases again:



- ▶ 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

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

Quiz 6: Which is true?

- a) Structural equality implies physical equality
- b) Physical equality implies structural equality
- c) Physical equality does not work for cyclic data structures
- d) `==` always means physical equality

Quiz 6: Which is true?

- a) Structural equality implies physical equality
- b) **Physical equality implies structural equality**
- c) Physical equality does not work for cyclic data structures
- d) `==` always means physical equality