CMSC 330: Organization of Programming Languages

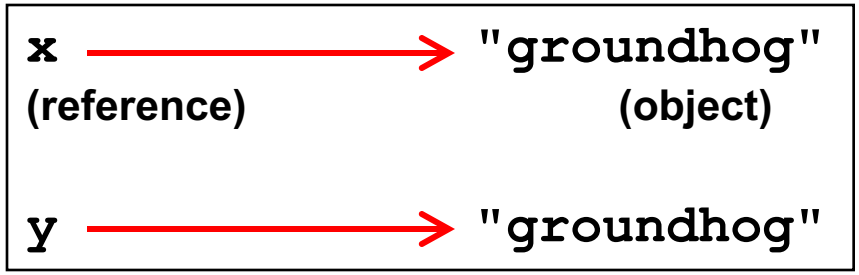
Equality, Mixin Inheritance, Miscellany

Object Copy vs. Reference Copy

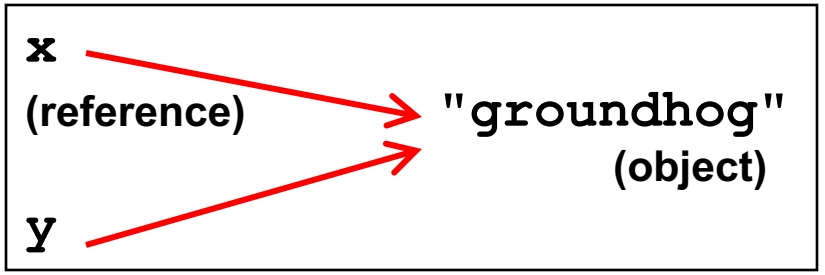
- Consider the following code
  - Assume an object/reference model like Java or Ruby
    - Or even two pointers pointing to the same structure

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

- Which of these occur?



Object copy



Reference copy

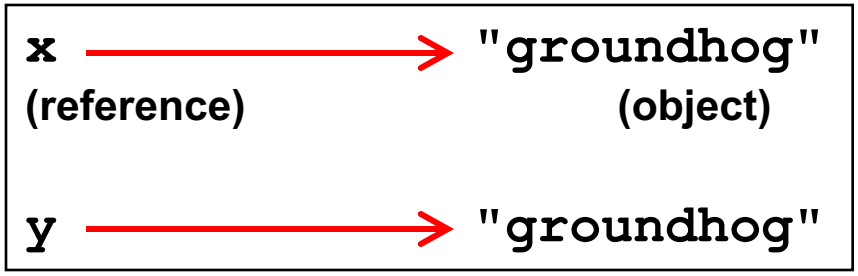
Object Copy vs. Reference Copy (cont.)

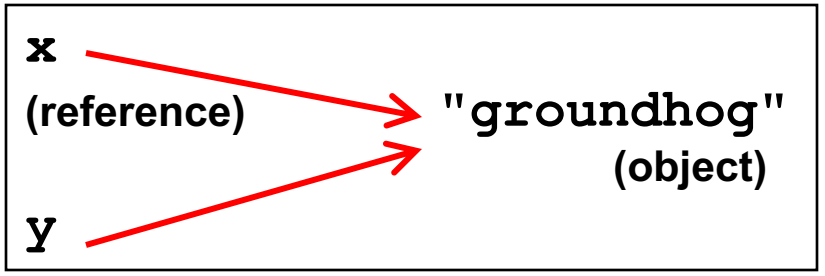
- For
```
x = "groundhog" ; y = x
```
  - Ruby and Java would both do a reference copy

- 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 1: 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 1: 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

Comparisons

- Sorting requires ability to compare two values
- Ruby comparison method <=>
  - -1 = less
  - 0 = equals
  - +1 = greater
- Examples
  - 3 <=> 4 returns -1
  - 4 <=> 3 returns +1
  - 3 <=> 3 returns 0

Sorting

- Two ways to sort an Array
  - Default sort (puts values in ascending order)
    - [2,5,1,3,4].sort # returns [1,2,3,4,5]
  - Custom sort (based on value returned by code block}
    - [2,5,1,3,4].sort { |x,y| y <=> x } # returns [5,4,3,2,1]
    - Where -1 = less, 0 = equals, +1 = greater
    - Code block return value used for comparisons

Quiz 2: What is the output?

```
print
  [1,4,7,3,2].sort { |x,y| (x % 2) <=> (y % 2) }
```

- Recall that % is the modulus operator
- And <=> is the built in comparison operator

a) `[1, 2, 3, 4, 7]`

b) `[4, 2, 1, 7, 3]`

c) `[1, 7, 3, 4, 2]`

d) `[7, 4, 3, 2, 1]`

Quiz 2: What is the output?

```
print
  [1,4,7,3,2].sort { |x,y| (x % 2) <=> (y % 2) }
```

- Recall that % is the modulus operator
- And <=> is the built in comparison operator

a) `[1, 2, 3, 4, 7]`

b) `[4, 2, 1, 7, 3]` – evens, then odds, in original order

c) `[1, 7, 3, 4, 2]`

d) `[7, 4, 3, 2, 1]`

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

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

```
gets     # implicitly reads input line into $_
print    # implicitly prints out $_
```

- Using $_ leads to shorter programs
  - And confusion
  - We suggest you avoid using it

Mixins

- Inheritance is one form of code reuse
- 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 a module for A, not class

Ruby Modules

- A module is a collection of methods and constants
  - Module methods can be called directly
    - So module defines a namespace for those methods
  - Instance methods are "mixed in" to another class

```
module Doubler
  def Doubler.base  # module method
    2
  end
  def double        # instance method
    self + self
  end
end
```

```
Doubler.base
  => 2
Doubler.class
  => Module
Doubler.new
  #err, no method
Doubler.double
  #err, no method
Doubler.instance
_methods
  => [:double]
```

Mixin Modules

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

```
class Integer # extends Integer
  include Doubler
end

10.double => 20
```

Inserts instance methods from **Doubler** into the class **Integer**

```
class String   # extends String
  include Doubler
end

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

Mixin Method Lookup

- When you call method m of class C, look for m
  1. in class C …
  2. in mixin in class C …
     - if multiple mixins included, start in latest mixin, then try earlier (shadowed) ones …
  3. in C's superclass …
  4. in C's superclass mixin …
  5. in C's superclass's superclass …
  6. …

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

Quiz 3: What is the output?

```
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.
- Both M1 and M2 have a method hello. M2's hello shadows M1's.

```
a = A.new
puts a.hello
```

a. "A hello"
b. "M1 hello"
c. "M2 hello"
d. (nothing)

Quiz 3: What is the output?

```
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.
- Both M1 and M2 have a method hello. M2's hello shadows M1's.

```
a = A.new
puts a.hello
```

a. "A hello"
b. "M1 hello"
c. "M2 hello"
d. (nothing)

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.class.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
```

Mixins: Enumerable

```
class MyRange
  include Enumerable  #map,select, inject, collect, find

  def initialize(low,high)
    @low = low    #(2,8)
    @high = high
  end
  def each #used by Enumerable
    i=@low
    while i <= @high
      yield i
      i=i+1
    end
  end
end
```

Ruby Summary

- Interpreted
- Implicit declarations
- Dynamically typed
- Built-in regular expressions
- Easy string manipulation
- Object-oriented
  - Everything (!) is an object
- Code blocks
  - Easy higher-order programming!
  - Get ready for a lot more of this...

Makes it quick to write small programs

Hallmark of scripting languages

Other Scripting Languages

- Perl and Python are also popular scripting languages
  - Also are interpreted, use implicit declarations and dynamic typing, have easy string manipulation
  - Both include optional "compilation" for speed of loading/execution
- Will look fairly familiar to you after Ruby
  - Lots of the same core ideas
  - All three have their proponents and detractors
  - Use whichever language you personally prefer

Example Perl Program

```
#!/usr/bin/perl
foreach (split//, $ARGV[0])) {
  if ($G{$_}) {
    $RE .= "\\" . $G{$_};
  } else {
    $RE .= $N ? "(?!\\" .
  join("|\\",values(%G)) . ')(\w)' : '(\w)';
    $G{$_} = ++$N;
}  }
```

Example Python Program

```
#!/usr/bin/python
import re
list = ("deep", "deer", "duck")
x = re.compile("^\S{3,5}.[aeiou]")
for i in list:
  if re.match(x, i):
    print I
  else:
    print
```