Ruby II

CSc 372: Comparative Programming Languages

Inheritance

Ruby Inheritance

- like most OO languages, Ruby supports inheritance
- if class A inherits from class B, it will inherit its data and functions, which can be overridden
- single inheritance (i.e. one direct parent)
- every object has methods it inherits (through the ancestral line) from Object

```ruby
class Animal
  attr_accessor :name, :type, :age
  def initialize(name, type, age)
    @name = name
    @type = type
    @age = age
  end

  def grow
    @age += 1
  end

  def speak
    puts "Hi!"
  end
end


class Dog < Animal
  def initialize(name, age)
    super(name, 'canine', age)
  end

  def speak
    puts "Woof!"
  end
end
```

- The `Dog` class inherits from the `Animal` class.
- Syntax: `Subclass < Superclass`
- `super`: used to call parent's method
- `grow` will do the same thing for `Dog` as it will for `Animal`
- `speak` has been overridden so the result will depend on the actual type
- The variables and accessors did not have to be redefined for `Dog`.

```ruby
dog = Dog.new('Howard', 10)
dog.speak
puts dog.age
dog.grow
puts dog.age
```

Typing

Typing in Ruby

- dynamic typing–types are checked at runtime rather than before
- allows for flexibility without the nuisance of polymorphism
- inheritance is possible
- duck typing is possible

What is duck typing?

First…what is NOT duck typing?

First...what is NOT duck typing?

- could be called *nominative* typing where a value is associated with a specific *type* of object
- can still allow for polymorphism through inheritance and other means
- *what the object can do is determined by its set type*
- Example in Ruby: the `is_a?` method allows you to check the actual type of any Object

So what IS duck typing?

- “If it walks like a duck and quacks like a duck, treat it like a duck…”
- which is actually a little misleading.
- A more appropriate description would be: *We don’t really care if it’s a duck or not. We care if it can walk and quack.*
- Example in Ruby: The `respond_to?` function allows you to check if an object responds to (i.e. has an implementation of) a particular method.
- It’s a little bit like implementing interfaces–i.e. anything that implements `Comparable` in Java is expected to have an implementation of `compareTo`.

```ruby
class Animal
  attr_accessor :name, :type, :age
  def initialize(name, type, age)
    @name = name
    @type = type
    @age = age
  end

  def grow
    @age += 1
  end

  def speak
    puts "Hi!"
  end
end
```

```ruby
class Dog < Animal
  def initialize(name, age)
    super(name, 'canine', age)
  end

  def speak
    puts "Woof!"
  end

  def walk
    puts "The #{@type} is walking..."
  end
end
```

```ruby
class Duck < Animal
  def initialize(name, age)
    super(name, 'bird', age)
  end

  def speak
    puts "Quack!"
  end
  alias quack speak #quack is now a second name for the Duck's speak method
  def swim
    puts "The #{@type} is swimming..."
  end

  def walk
    puts "The #{@type} is walking..."
  end
end
```

```ruby
dog = Dog.new("Howard", 11)
duck = Duck.new("Howard", 27)
animal = Animal.new("Animal", "muppet", 47)
list = Array.new()
list.push(dog)
list.push(duck)
list.push(animal)
list.push("String")
list.push(3)
```

```ruby
#this is not duck typing because we are actually
#checking the type of the variable
puts "NOT DUCK TYPING"
list.each do |a|
  if a.is_a?(Animal)
    puts "#{a.name} is a #{a.type} and is #{a.age} years old."
    puts "#{a.name} says: "
    a.speak
  end
  if a.is_a?(Duck)
    a.quack
    a.speak
    a.walk
    a.swim
  elsif a.is_a?(Dog)
    a.walk
    a.speak
  end
end
```

```ruby
dog = Dog.new("Howard", 11)
duck = Duck.new("Howard", 27)
animal = Animal.new("Animal", "muppet", 47)
list = Array.new()
list.push(dog)
list.push(duck)
list.push(animal)
list.push("String")
list.push(3)
```

```ruby
puts "\nDUCK TYPING"
#this is duck typing because instead of checking types we check
#functionality
list.each do |a|
  if a.respond_to?(:speak)
    puts "#{a.name} is a #{a.type} and is #{a.age} years old."
    puts "#{a.name} says: "
    a.speak
  end
  if a.respond_to?(:walk)
    a.walk
  end
  if a.respond_to?(:quack)
    a.quack
    a.swim
  end
end
```

Aliasing

Aliasing

- The keyword `alias` can be used to create aliases for methods.
- One use for this may be if you use inheritance and want to both override a method AND preserve the original one.
- How would you do that in Java?
- Important thing to keep in mind: Since Ruby is interpreted, where you put the alias statement in the code may affect the results.

```ruby
class Animal
  attr_accessor :name, :type, :age
  def initialize(name, type, age)
    @name = name
    @type = type
    @age = age
  end

  def grow
    @age += 1
  end

  def speak
    puts "Hi!"
  end
end


class Dog < Animal
  def initialize(name, age)
    super(name, 'canine', age)
  end
  alias sayHi speak
  def speak
    puts "Woof!"
  end
  alias bark speak
end

dog = Dog.new('Howard', 10)
dog.sayHi#uses the Animal version of "speak"
dog.speak#uses the Dog version of "speak"
dog.bark#the Dog version of "speak"
```

Blocks

Blocks in Ruby

- Blocks are like functions that can be passed as arguments into another function.
- You've already seen some blocks–e.g. when iterating through an array or range.
- There are various ways to implement these.

```ruby
#basic block example
def saySomething
   if block_given?
      yield#run the block
   end
end

saySomething {puts "hello!"}
saySomething {puts "goodbye!"}
saySomething
```

```ruby
#method that takes an array and a value
#and multiplies the value to every element in the array
def multAll a, n
  a.each_index {|i| a[i] = a[i]*n}
end

array = Array.[](3, 1, 4, 1, 5, 9, 2)
print array
puts
multAll array, 5
print array
puts
```

```ruby
#a map function for an array
#this means we pass in a block that
#should be applied to every element in the array
def myMap(a, &block)
  ret = []
  a.each {|x| ret.push(block.call(x))}
  return ret
end

array = Array.new(10) {|i| i = i + 5}
print array
puts
isEven = proc {|x| x%2==0} #one way of defining a block
puts "applying isEven"
print myMap(array, &isEven)
puts
```

```ruby
#we can also pass in a method
def cube n
  n*n*n
end

puts "applying cube"
print myMap(array, &method(:cube))
puts

#you can also define the function inside the argument call
puts "dividing each item by 5"
print myMap(array, &Proc.new {|n| n/5})
puts
```

```ruby
#filter out the items in an array for which block returns true
def myFilter(a, &block)
  ret = []
  a.each {|x| ret << x if !block.call(x)}
  ret
end

puts "get the items that are not even"
print myFilter(array, &isEven)
puts
```

```ruby
#a method can take multiple blocks
#call a if x is even and b if it is odd
def test(x, a, b)
  if x%2==0
    a.(x)
  else
    b.(x)
  end
end

test(4, Proc.new {|x| puts x*x}, Proc.new {|x| puts x+x})#x is squared because it is even
test(5, Proc.new {|x| puts x*x}, Proc.new {|x| puts x+x})#x is doubled because it is odd

#another way of doing the same thing
def test(x, a, b)
  if x%2==0
    a.call(x)
  else
    b.call(x)
  end
end

test(4, Proc.new {|x| puts x*x}, Proc.new {|x| puts x+x})
test(5, Proc.new {|x| puts x*x}, Proc.new {|x| puts x+x})
```

```ruby
#take an array--apply one block to the elements if the item is negative and another block if it is not
def apply(a, b1, b2)
  ret = []
  a.each do |x|
    if x < 0
      ret << b1.(x)
    else
      ret << b2.(x)
    end
  end
  return ret
end

array = Array.[](-6, -8, 0, 1, 4, -2, 3, 7, -10)
print array
puts
print apply(array, Proc.new {|x| x = -1*x}, Proc.new {|x| x = x * 2})
puts
```

Exercises: ICA 14

1. Write the following items in a file called `ica14.rb` and submit them on lectura using the following command:

```
turnin csc372ica14 ica14.rb
```

2. Write a method called `divideList` that divides a list of elements into two lists where the first list contains the elements for which a block returns true and the second list contains the elements for which it returns false.
3. Write a method called `countTrue` that takes an array and two blocks and returns the number of elements for which each block returns true. That means you should be returning two values.
4. Write a class for a binary search tree called `BST` with a method called `traverseTree` that traverses each node in the tree and applies a block to each one.