CSc 372

Comparative Programming Languages

28 : Ruby — Classes

Department of Computer Science
University of Arizona

collberg@gmail.com

Christian Collberg

Inheritance

- Let's start with this class `Bird`, with two instance variables `name` and `age`:

```
class Bird
   def initialize(name,age)
      @name = name
      @age = age
   end
   def to_s
      "#{@name} : #{@age}"
   end
end

puts Bird.new("donald",45)
```

Inheritance

- We can can create a new class, `Duck`, as an extension of `Bird`:

```
class Duck < Bird
   def initialize(name,age,kind)
      @name = name
      @age = age
      @kind = kind
   end
end
puts Duck.new("huey",8,"cartoon")
```

Overriding Methods

- Now, the `to_s` doesn't print the new attribute `kind` — but we can override it with a new definition.
- Note that both `to_s` methods now exist, one in `Bird` and one in `Duck`.

```
class Duck < Bird
  def initialize(name,age,kind)
    @name = name
    @age = age
    @kind = kind
  end
  def to_s
    "#{@name} : #{@age} : #{@kind}"
  end
end
```

Overriding methods

- We can call the method in the super class using the `super` keyword — it sends the same message (with the same arguments) to the parent class.

```
class Duck < Bird
  def initialize(name,age,kind)
    @name = name
    @age = age
    @kind = kind
  end
  def to_s
    super + " : #{@kind}"
  end
end
```

Defining getters

- We can define getters by hand, like this:

```
class Duck
   def initialize(name,age,kind)
      @name = name; @age = age; @kind = kind
   end
   def name
      @name
   end
   def age
      @age
   end
end
d = Duck.new("huey",8,"cartoon")
puts d.name()
```

Defining getters

- The attr_reader method, does this for us.
- attr_reader is actually a method (!) defined in module Module that generates these methods automatically

```
class Duck
   def initialize(name,age,kind)
      @name = name
      @age = age
      @kind = kind
   end
   attr_reader :name, :age
end
d = Duck.new("huey",8,"cartoon")
puts d.name()
```

Defining setters

- We can define setters too, by creating a method `"attr="` for an attribute `attr`

```
class Duck
   def initialize(name,age,kind)
      @name = name; @age = age; @kind = kind
   end
   attr_reader :name, :age
   def age=(new_age)
      @age = new_age
   end
end
d = Duck.new("huey",8,"cartoon")
d.age = 9
```

Defining setters

- Or, we can use `attr_writer` to generate the setters automatically:

```
class Duck
   def initialize(name,age,kind)
      @name = name; @age = age; @kind = kind
   end
   attr_reader :name, :age
   attr_writer :age
end
d = Duck.new("huey",8,"cartoon")
d.age = d.age + 1
d.age += 1
puts d
```

Class variables

- Class variables start with `@@`. They should be initialized inside the class.

```
class Duck < Bird
   @@number = 0
   def initialize(name,age,kind)
      @name = namel @age = age; @kind = kind
      @@number += 1
      @number = @@number
   end
   attr_reader :name, :age
   attr_writer :age
   def to_s
      super + ":#{@kind}[bird ##{@number}:of #{@@number}]"
   end
end
```

Defining class methods

- Class (static) methods are defined by prefixing the name with the classname:

```
class Bird
   @@flock = []
   def initialize(name,age)
      @name = name; @age = age
      @@flock << self
   end
   def Bird.flock
      return @@flock
   end
end
Bird.new("huey",8); Bird.new("dewey",8); ...
puts Bird.flock
```

Access control

- `public`, `protected`, `private` mean roughly the same as in Java.
- Of course, access control is dynamic — everything happens at runtime. There are no errors until you try to execute a method you don't have access to.

```
class Bird
  def roast; end
  def steam; end
  def fry; end
  def deepfry; end
  public :roast, :steam
  protected :fry
  private :deepfry
end
```

Freezing objects

- You can freeze an object to prevent someone from modifying it.

```
class Bird
   def initialize(name,age)
      @name = name; @age = age
   end
   attr_writer :age
end
h = Bird.new("huey",8)
h.age = 9
h.freeze
h.age = 10
puts h
```

Freezing classes

- As we've seen, class definitions are executable code, they essentially build the class at runtime, as they're encountered.
- So, since classes are objects, too, it makes sense that we can freeze them:

```
Bird.freeze

class Bird
  def newmethod
  end
end
```

Exercise: Factorial

- Write the factorial program in Ruby.
- Note that there's no need to put the function in a class.
- Extend the program to take input from the command line, i.e. if your file is called `fact`, you should be able to do

  ```
  > fact 10
  3628800
  ```

  HINT: `ARGV` holds the input arguments, the method `to_i` converts from string to integer.

Exercise: Reading

- Write a program which reads a string from the user and prints `true` if its `y` or `Y`, `false` if it's `n` or `N` or an empty line, and loops otherwise. Ignore leading or trailing blanks. Examples:

```
> ./yes
Are you sure? [y/n]: y
true
> ./yes
Are you sure? [y/n]:      n
false
> ./yes
Are you sure? [y/n]: asdfsdf
Are you sure? [y/n]: dsfsdfs
Are you sure? [y/n]:
false
```

HINT: `gets()` reads a string from the command line.

Exercise: Complex Class

- Write a class `Complex` that implements complex numbers. Given these statements

  ```
  a = Complex.new(10,20)
  puts a
  b = a.add(Complex.new(5,6))
  puts b
  ```

  the program should print

  ```
  > ruby Complex.rb
  10+i20
  15+i26
  ```

  HINT: Use string interpolation in `to_s`.

Exercise: Operator overloading

- Extend `Complex` from the previous problem so that `add` can be called using the + operator instead. Given these statements

  ```
  a = Complex.new(10,20)
  b = Complex.new(5,6)
  c = a + b
  puts c
  ```

  the program should print

  ```
  > ruby Complex.rb
  15+i26
  ```

  HINT: An operator is defined like this:

  ```
  def * (a)
     ...
  end
  ```

Exercise: Complex Arrays

- Write a class `ComplexArray` to implement arrays of complex numbers. Given these statements:

  ```
  a = Complex.new(10,20)
  b = Complex.new(5,6)
  x1 = ComplexArray.new([a,b])
  puts x1
  ```

  the program should print

  ```
  > ruby Complex.rb
  [10+i20,5+i6]
  ```

Exercise: Polymorphic functions

- Extend `Complex` by overriding the `add` method so that it now can take both a `Complex` number and an integer as argument. These statements

  ```
  a = Complex.new(10,20)
  puts a.add(Complex.new(5,6))
  puts a.add(5)
  puts a + 5
  ```

  should produce

  ```
  > ruby Complex.rb
  15+i26
  15+i20
  15+i20
  ```

  HINT: To do the type test you use: `b.kind_of?(Fixnum)`.

Readings

- Read Chapter 3, page 25–41, in *Programming Ruby — The Pragmatic Programmers Guide*, by Dave Thomas.
- Read page 394–395, in *Programming Ruby*, about freezing objects.

The three of us are twins!