Multiparadigm programming with Ruby

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
cc@cse.concordia.ca

August 23, 2018

Imperative programming and pure OOP

- In computer science, imperative programming is a programming paradigm that describes computation in terms of statements that change a program state.
- The term *pure object-oriented programming* implies that all of the data types in the language are objects and all operations on those objects can be invoked by message passing.
- Sending a message to an object invokes a method by the receiver object. item A message contains the method's name along with any parameters.

Examples: Message passing

```
puts "The Ruby language".length #=> 17
puts "Ruby".index("y")          #=> 3
puts -7.abs                     #=> 7
puts 10.49.round                #=> 10
puts 10.51.round                #=> 11
puts 2.next                     #=> 3
puts 97.chr                     #=> "a"
```

Variables and aliasing

- Multiple variables referencing the same object is called *aliasing*. Consider the following example:

```
person1 = "Tony"
person2 = person1
```

- The assignment of `person1` to `person2` does not create an object.
- It assigns the object reference of `person1` to `person2`, so that both variables now would refer to the same object.
- We can avoid aliasing with `dup`, which creates a new object with identical contents.

```
person3 = person1.dup
person1[0] = "R"
puts person1 #=> Rony
puts person2 #=> Rony
puts person3 #=> Tony
```

Chaining assignment statements

- An assignment statement sets the value of a variable on its left hand side (*lvalue*) to the value of the expression on its right hand side (*rvalue*).
- Ruby supports chaining of assignments. It also allows one to perform assignments in some unexpected places. Consider the example below:

```
a = b = 1 + 2 + 3
puts a #=> 6
puts b #=> 6
a = (b = 1 + 2) + 3
puts a #=> 6
puts b #=> 3
```

Parallel assignment statements

```
a = 1
b = 2
a, b = b, a
puts a #=> 2
puts b #=> 1
x = 0
a, b, c = x, (x += 1), (x += 1)
puts a #=> 0
puts b #=> 1
puts c #=> 2
puts x #=> 2
```

Arrays

- An array is an ordered collection of elements, where each element is identified by an integer index.
- We can create arrays using literals. A literal array is simply a list of objects between square brackets. As everything is an object, this implies that an array can hold objects of different types, as in the example below:

```
a = [ "number", 1, 2, 3.14 ] # Array with four elements.

puts a[0]       # Access and display the first element.
                #=> number

a[3] = nil      # Set the last element to nil.

puts a          # Access and display entire array.
                #=> number 1 2 nil
```

Arrays /cont.

- We can also create an array by explicitly creating an Array object. Ruby allows us to specify array ranges, as in the example below:

```
myarray = [ 1, 2, 3, 4, 5, 6 ]

     index  0  1  2  3  4  5

puts myarray[0]       #=> 1

                      # [i...j] from index i to j
                      # excluding j
puts myarray[1...3]   # Exclusive range. => 2 3.

                      # [i..j] from index i to j
                      # including j
puts myarray[1..3]    # Inclusive range. => 2 3 4.

puts myarray[1,3]     # Range between 1st up to 3rd
                      # consecutive, inclusive.
                      #=> 2 3 4.
```

Arrays /cont.

- Ruby allows a negative index, forcing the array to count from the end.

```
a = [ "pi", 3.14, "prime", 17 ]
puts a.class                    #=> Array
puts a.length                   #=> 4
puts a[0]                       #=> pi
puts a[-1]                      #=> 17
puts a[1]                       #=> 3.14
puts a[2]                       #=> prime
puts a[3]                       #=> 17
puts a[4]                       #=> nil
b = Array.new
puts b.class                    #=> Array
puts b.length                   #=> 0
b[0] = "a"
b[1] = "new"
b[2] = "array"
puts b                          #=> a new array
```

Associative arrays

- An *associative array* (or *hash*) is an unordered collection of elements.
- An element is a pair of two objects: a *value* and a *key* through which the value can be retrieved. The value can be an object of any type.
- To store an element in an associative array, we must supply both objects:

$$\begin{aligned} hashName = \ & \{ \text{"}key\text{"} \Rightarrow \text{"}value\text{"}, \\ & \ldots \\ & \} \end{aligned}$$

- We can subsequently retrieve the value by supplying the appropriate key:

$$hashName[\text{"}key\text{"}] \Rightarrow value$$

Example: Associative arrays

```
biblio = { "nabokov89a" => "Pnin",
           "bulgakov96" => "The master and margarita",
           "nabokov89b" => "Invitation to a Beheading",
            "nabokov90" => "The defense",
              "kafka95" => "The trial" }

puts biblio.length          #=> 5
```

Example: Associative arrays /cont.

- We can access the collection to obtain the value associated with a given key:

```
puts biblio["bulgakov96"] #=> The master and margarita
```

- We can also access the collection in order to modify the value associated with a given key:

```
biblio["nabokov89a"] = "Lolita"
puts biblio["nabokov89a"] #=> Lolita
```

Example: Associative arrays /cont.

- We can also add to the collection:

```
biblio["nietzsche97"] = "Beyond good and evil"
puts biblio["nietzsche97"] #=> Beyond good and evil
puts biblio.length         #=> 6
```

- We can delete an element from the collection by supplying the appropriate key:

```
biblio.delete_if {|key, value| key == "kafka95"}
puts biblio.length         #=> 5
```

Iterating over an associative array

- We can iterate over the entire collection. In the example below we display all key-value pairs:

```
biblio.each_pair do |key, value|
  puts "#{key} : #{value}"
end
```

- The above will display:

```
nabokov90 : The defense
nietzsche97 : Beyond good and evil
nabokov89a : Lolita
nabokov89b : Invitation to a Beheading
bulgakov96 : The master and margarita
```

Iterating over an associative array /cont.

- One of the strengths (and perhaps weaknesses) of Ruby is that it allows us to do the same thing using different ways. We can perform the above iteration as follows:

```
biblio.each do |key, value|
  puts "#{key} : #{value}"
end
```

- There is yet another way to do that:

```
biblio.each {|key, value| puts key + " : " + value}
```

Iterating over an associative array /cont.

- We can iterate over the collection and access and display each key individually:

```
biblio.each_key {|key| puts key}
```

- The above will display:

```
nabokov90
nietzsche97
nabokov89a
nabokov89b
bulgakov96
```

Defining classes and features: Naming

- Ruby uses a convention to help it distinguish the usage of a name: the first characters of a name indicate how the name is used.
- Class names, module names, and constants should start with an uppercase letter.
- Class variables start with two "at" signs (`@@`).
- Local variables, method parameters, and method names should all start with a lowercase letter or with an underscore (`_`).
- Global variables are prefixed with a dollar sign (`$`), while instance variables begin with a single "at" sign.

Defining classes and features: Naming /cont.

```
local_variable
CONSTANT_NAME / ConstantName / Constant_Name
:symbol_name
@instance_variable
@@class_variable
$global_variable
ClassName
method_name
ModuleName
```

Objects

- Instances of classes (objects) contain state and behavior.
- Each object contains it own unique state.
- Behavior on the other hand is shared among objects.
- The state of the object is composed of a set of attributes (or fields), and their current values.

Example: Class Coordinate

```
class Coordinate
  @@total = 0
  def initialize (x, y)
    @@total += 1
    @x = x
    @y = y
  end
  def to_s
    return "(#@x, #@y)"
  end
  def Coordinate.total
    return "Number of coordinates:  #@@total"
  end
  ...
end
```

Example: Class Coordinate /cont.

```
def setx (x)
 @x = x
end
def sety (y)
  @y = y
end
def getx
  @x
end
def gety
 @y
end
```

Example: Class Coordinate /cont.

- The `class` keyword defines a class.
- By defining a method inside this class, we are associating it with this class.
- The `initialize` method is what actually constructs the data structure. Every class must contain an initialize method.
- `@x` and `@y` are instance (object) variables.
- `puts` and `print` write each of their arguments. `puts` adds a new line, whereas `print` does not add a new line.

Example: Class Coordinate /cont.

- A class can be instantiated with `new` as in

  `p1 = Coordinate.new(0, 0)`

  which defines an instance `p1` whose coordinates are $(0, 0)$.
- We can now use `p1`:

```
puts p1.to_s              #=> (0, 0)
p1.setx(2)
puts p1.getx              #=> 2
p1.sety(3)
puts p1.gety              #=> 3
puts p1.to_s              #=> (2, 3)
p2 = Coordinate.new(1, 1)
puts Coordinate.total  #=> Number of coordinates: 2
```

Example: Class Coordinate refined

```
class Coordinate
  attr_accessor :x, :y
  @@total = 0
  def initialize (x, y)
    @@total += 1
    @x = x
    @y = y
  end
  def to_s
     return "(#@x, #@y)"
  end
  def Coordinate.total
    return "Number of coordinates:  #@@total"
  end
end
```

Example: Class Coordinate refined /cont.

```
p1 = Coordinate.new(0,0)
puts p1.to_s      #=> (0, 0)
p1.x = 2
puts p1.x         #=> 2
p1.y = 3
puts p1.y         #=> 3
puts p1.to_s      #=> (2, 3)
```

Example: Subclassifying class Coordinate

- Consider class `XYZCoordinate` which defines a three-dimensional coordinate.

```
require "CoordinateV2.rb"
class XYZCoordinate < Coordinate
  attr_accessor :z
  @@newtotal = 0
  def initialize (x, y, z)
    super(x, y)
    @z = z
    @@newtotal += 1
  end
  def to_s
    return "(#{@x}, #{@y}, #{@z})"
  end
  def XYZCoordinate.total
    return "Number of 3D-coordinates: #@@newtotal"
  end
end
```

Example: Subclassifying class coordinate /cont.

```
p1 = XYZCoordinate.new(0,0,0)
puts p1.to_s                  #=> (0, 0, 0)
p2 = XYZCoordinate.new(1,5,5)
puts p2.to_s                  #=> (1, 5, 5)
puts XYZCoordinate.total      #=> Number of 3D-coordinates: 2
```

Object extensions

- Ruby allows us to extend specific instances with new behavior. Consider the example below:

```
def p1.whatIam
  return "The origin on the 3D system."
end

puts p1.whatIam #=> The origin on the 3D system.
puts p2.whatIam #=> Will cause an error.
```

Control flow: Single selection

- Ruby provides a rich set of control flow constructs to support *selection* and *repetition*.
- Consider the sentence "If you are a Computer Science student, then you must take this course."
- In other words, an action must be taken provided a certain condition holds.
- To support selection, the `if` statement is perhaps the simplest and it comes in three variations.
- Initially to support single selection with the optional alternative to execute a statement if the condition evaluates to false.

```
if boolean-expression-1 [then]
   if-body
[else boolean-expression-2 [then]
   else-body]
end
```

Control flow: Single selection /cont.

- The `if` statement also works as a statement modifier which evaluates expression if boolean-expression is true:

$$expression \texttt{ if } boolean - expression$$

- Finally, the `if` statement can be used as a ternary operator:

$$boolean - expression \texttt{ ? } expression_1 \texttt{ : } expression_2$$

which returns $expression_1$ if $boolean - expression$ is true and $expression_2$ otherwise.

Single selection with `unless`

- In Ruby, a negated form of the `if` statement is also available:
- Consider the sentence: "You must take this course, *unless* you have already taken an equivalent one."
- In other words, you have to take an action only if a certain condition does *not* hold.
- The term *unless* works as a negated *if*.

```
unless boolean-expression [then]
    unless-body
[else
    else-body]
end
```

- The `unless` statement can also work as a statement modifier:

$$\textit{expression}\ \texttt{unless}\ \textit{boolean} - \textit{expression}$$

which evaluates `expression` only if `boolean-expression` is false.

Multiple selection with `extended if`

- To support *multiple selection*, we can use an extended version of the `if` statement:

```
if boolean-expression-1 [then]
   if-body
elseif boolean-expression-2 [then]
   elseif-body
   ...
[else boolean-expression-n [then]
   else body]
end
```

Multiple selection: `case`

- We can also use the `case` statement: When a comparison returns true, the search stops and the body associated with the comparison is executed.
- The statement then returns the value of the last expression executed.
- If no comparison matches and an `else` clause is present, its body will be executed; otherwise, the statement returns `nil`.

```
case target
    when comparison [, comparison ] ... [ then ]
        body
    when comparison [, comparison ] ... [ then ]
        body
    ...
      [ else
          body ]
end
```

Example: Multiple selection with case

```
number = 11
case number
  when 1, 3, 5, 7, 9
    puts "Odd."
  when 0, 2, 4, 6, 8, 10
    puts "Even."
  else
    puts "Number is out of range."
end
```

Repetition with `while`

- The `while` loop executes its body zero or more times as long as its condition is true:

```
while boolean-expression [ do ]
   body
end
```

- The `while` loop can also operate as a statement modifier:
`expression while boolean-expression`

Repetition with `until`

- There is also a negated form that executes the body as long as `boolean-expression` is false (or: until the boolean-expression becomes true):

```
until boolean-expression [ do ]
   body
end
```

- The `while` can also work as a statement modifier:

```
expression until boolean-expression
```

Repetition with do

- Ruby also provides the do statement:

```
loop do
   body
   next if boolean-expression  # skip iteration
   break if boolean-expression # exit loop
   redo if boolean-expression  # do it again
end
```

Iterator-based loops

```
3.times { |count| puts count}          #=> 0 1 2

1.upto(10) { |count| puts count }      #=> 1 2 3 4 5 6 7 8 9 10

10.downto(1) { |count| puts count }    #=> 10 9 8 7 6 5 4 3 2 1

0.step(10,2) { |count| puts count }    #=> 0 2 4 6 8 10

for element in ['a', 'b', 'c']
  puts element                         #=> a b c
end
```

Iterators

- The keyword `each` returns successive elements of its collection:

```
a = [ "3.14", "number", "pi" ]
a.each { |el| print el + " " }          #=> 3.14 number pi
```

- The keyword `collect` takes each element from a collection and passes it to a block. The code below takes each element from the collection and displays its successor.

```
print ["H", "A", "L"].collect { |x| x.succ }      #=> IBM
```

- The keyword `find` returns the first element from a collection which meets a condition. Otherwise it returns nil. The code below displays the first even number from a collection.

```
print [1, 3, 7, 8, 9, 10].find { |x| x % 2 == 0 } #=> 8
```

Regular expressions

- A regular expression is a way of specifying a pattern of characters to be matched in a string.
- In Ruby this is done with `/pattern/`.
- In Ruby, regular expressions are objects and can thus be manipulated as such. Some common pattern descriptions are shown below:

Pattern	Description
`/Lisp|Lava/`	Matches a string containing *Lisp*, or *Lava*.
`/L(isp|ava)/`	As above.
`/ab+c/`	Matches a string containing an *a*, followed by one or more *b*s, followed by a *c*.
`/ab*c/`	matches a string containing an *a*, followed by zero or more *b*s, followed by a *c*.
`.`	Matches any character.
`/[Colloqui[um|a]/`	Matches *Colloquium*, or *Colloquia*.

Example: Reading a song by Pink Floyd

- Consider the song "Welcome to the machine" by Pink Floyd. In this example we are looking to extract and display lines which contain the word "punish."

```
File.open("welcome-to-the-machine.txt").each { |line|
    puts line if line =~ /punish/
}
```

- This will display:

```
You bought a guitar to punish your ma,
```

Access control: Defining access rights for features

- Public methods can be called by anyone. Methods are public by default (except for `initialize`, which is always private, see below).
- Protected methods can be invoked only by objects of the defining class and its subclasses.
- Private methods can be called only in the defining class.

Specifying access control

```
class MyClass
  def method1   # default is 'public'
    ...
  end
  protected     # subsequent methods will be protected'
  def method2
    ...
  end
  private       # subsequent methods will be 'private'
  def method3
    ...
  end
  public        # subsequent methods will be 'public'
  def method4
    ...
  end
end
```

Alternative specification of access control

```
class MyClass
  def method1
    ...
  end
  ...
  public :method1, :method4
  protected :method2
  private :method3
end
```

Introspection

- *Introspection* is the ability of a computational system to consult (but not modify) its own structure.
- In Ruby, we can obtain the following type of knowledge about a program:
  - What objects it contains.
  - The contents and behaviors of objects.
  - The current class hierarchy.

Example: Introspection

- For the instantiations below:

```
require "CoordinateV2.rb"
require "XYZCoordinate.rb"
p1 = Coordinate.new(0, 0)
p2 = XYZCoordinate.new(0,0,0)
def p2.whatIam
  return "The origin on the 3D system."
end
```

  we can execute reflective queries to obtain knowledge about the system.

Example: Introspection /cont. - Looking for objects

- What objects does the system contain?
- We can iterate over all instances of `Coordinate` in the system, posing a reflective query about each one.
- Let us inspect the system for objects of type `Coordinate`:

```
ObjectSpace.each_object(Coordinate) { |p|
  puts p.inspect
}
```

- We obtain the following:

```
#<XYZCoordinate:0x28455d8 @y=0, @z=0, @x=0>
#<Coordinate:0x2846028 @y=0, @x=0>
```

- Note that an instance of `XYZCoordinate` *is_a* `Coordinate`, hence the listing of `p2` in the output.

Example: Introspection /cont. - Content and behavior of objects

- We can check whether or not a particular object may respond to a message:

```
puts p1.respond_to?("setX")          #=> false
puts p2.respond_to?("whatIam")       #=> true
```

- We can also determine the class and unique id of objects, and test their relationship to classes:

```
puts p1.id                           #=> 21113660
puts p1.class                        #=> Coordinate
puts p2.class                        #=> XYZCoordinate
puts p2.instance_variables           #=> @y @z @x
puts p2.kind_of? Coordinate          #=> true
puts p2.kind_of? XYZCoordinate       #=> true
puts p1.kind_of? XYZCoordinate       #=> false
puts p2.instance_of? Coordinate      #=> false
puts p2.instance_of? XYZCoordinate #=> true
```

Example: Introspection /cont. - The current class hierarchy

- We can inquire about the direct superclass of a given class:

```
puts XYZCoordinate.superclass                #=> Coordinate
```

- We can also inquire about class features:

```
puts XYZCoordinate.private_instance_methods #=>
puts XYZCoordinate.public_instance_methods  #=>
puts XYZCoordinate.class_variables          #=> @@total
```