Programming Paradigms

Unit 4 — Ruby Advanced

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

1. Functions, Code Blocks and Procs
2. Classes
3. Modules and Mixins

Outline

1. **Functions, Code Blocks and Procs**
2. Classes
3. Modules and Mixins

Defining Functions

- Functions are used to bundle one or more statements into a single unit
- They can be defined in the console, and without defining a class first
- Functions are also referred to as methods

```
>> def tell_the_truth
>>     true
>> end
=> nil

>> tell_the_truth
=> true
```

- Every function returns something: if no explicit `return expr1, expr2, ...` statement is used, the value of the last processed expression is returned
  - More than one values are returned in an array
- Like everything else, functions in Ruby are considered objects
  - You can call `tell_the_truth.class` or `tell_the_truth.methods`

Positional Parameters to Functions

- Functions accept parameters
- The default are positional parameters, i.e., the order of the parameters matters

```
>> def method_name (var1, var2)
>>     expr..
>> end
```

- You can set default values for the parameters, which will be used if method is called without passing parameters

```
>> def method_name (var1=value1, var2=value2)
>>     expr ...
>> end
```

- Function call

```
>> method_name param1, param2
```

Positional Parameters to Functions: Example

- Function with two parameters that are initialized

```
>> def test(a1='Ruby', a2='Perl')
>>     puts "The programming language is #{a1}"
>>     puts "The programming language is #{a2}"
>> end
```

- Call function with two parameters

```
>> test 'C', 'C++'
>> The programming language is C
>> The programming language is C++
=> nil
```

- Call function with one parameter (which is the first one)

```
>> test 'C'
>> The programming language is C
>> The programming language is Perl
=> nil
```

Named Parameters to Functions

- Before Ruby 2.0, there was no support for named parameters, which are specified/referred to by a name in the function call
- For example, in Objective-C a function call can look like this:

```
[window addNew:@"Title"
       xPosition:20
       yPosition:50
       width:100
       height:50
       drawingNow:YES];
```

- Named parameters simplify, e.g.,
  - the handling of optional parameters
  - the parameters can be passed in any order

Named Parameters Using Hashes

- A single hash can be used to emulate named parameters (only way before Ruby 2.0)

```
>> def tell_the_truth(options = {})
>>     case options[:profession]
>>     when :lawyer
>>         'almost certainly not false'
>>     when :doctor
>>         true
>>     else
>>         'yep'
>>     end
>> end

>> tell_the_truth( :profession => :lawyer )
=> "almost certainly not false"

>> tell_the_truth
=> "yep"
```

Named Parameters as Keyword Arguments

- Ruby 2.0 introduced keyword arguments, which are named parameters
- They are followed by a colon (:) and an optional default value
- When calling a method, the order of the arguments can be in any order without affecting the behavior (not the case for positional arguments)
- Parameters that are not initialized have to be specified

```
>> def total(subtotal:, tax:10, discount:5)
>> subtotal + tax - discount
>> end

>> total(subtotal:100)
=> 105

>> total(subtotal:100, discount:20)
=> 90

>> total(discount:20, subtotal:100)
=> 90

>> total()
ArgumentError: missing keyword: subtotal
```

Code Blocks/1

- A code block is basically a function without a name
- It can be passed as a parameter to a function or method
- It is delimited by
  - curly braces `{...}` (inline or single-line block) or
  - `do...end` (multi-line block)

```
>> 3.times { puts 'hello' }
hello
hello
hello
=> 3
```

- `times` is an iterator (method) for the class `Fixnum` that does something a certain number of times

Code Blocks/2

- Let's write our own version of `times` called `log2times`
- `x.log2times` does something $\log_2(x)$ times

```
class Fixnum
   def log2times
      i = self
      while i > 1
         i = i / 2
         yield
      end
   end
end
```

```
>> 2.log2times { puts 'Hello world!'}
Hello world!
=> nil
>> 5.log2times { puts 'Hello world!'}
Hello world!
Hello world!
=> nil
```

- `self` gives you access to the current object – the object that is receiving the current message
- `yield` in the method `log2times` calls the passed code block
- Ruby has open classes: write an existing class definition, specify/define something and it will be added to the class
  - The code extends the class `Fixnum` by adding a method `log2times`

Parameters to Code Blocks

- It is also possible to pass parameters to the code block
- In the block, a variable is placed in vertical lines | | to accept parameters

```
>> (0..3).each { |x| puts x }
0
1
2
3
=> 0..3
```

Equivalent expression:

```
(0..3).each do |x|
   puts x
end
```

- |x| assumes the values 0 . . . 3
- What is the result of the following expression?

```
>> ['2', 'plus', '3', 'is', "#{2+3}"].each { |x| puts x }
```

Procs/1

- Code blocks are not first-class citizens of Ruby
  - For example, you cannot assign them to a variable

```
>> y = { |x| puts x }
syntax error,...
```

- If you want to do something other than yielding them, you have to convert them to a `Proc`, which is a class in Ruby, and you can create objects of this type

```
>> y = Proc.new { |x| puts x }
=> #<Proc:0xb7367ac4>
>> y.call(3)
3
=> nil
>> y.class
=> Proc
```

Procs/2

- Any code block can be turned into a `Proc` object if it is passed as a parameter and the parameter is preceded by an ampersand
- The code block (object) is then executed with the `call` method (similar to `yield`)
- This allows to pass around executible code

```
>> def call_block(&block)
>>    block.call
>> end
=> nil
```

```
>> def pass_block(&block)
>>    call_block(&block)
>> end
=> nil
```

```
>> pass_block { puts 'Hello block!'  }
Hello block!
=> nil
```

Outline

1 Functions, Code Blocks and Procs

2 Classes

3 Modules and Mixins

Class Hierarchy

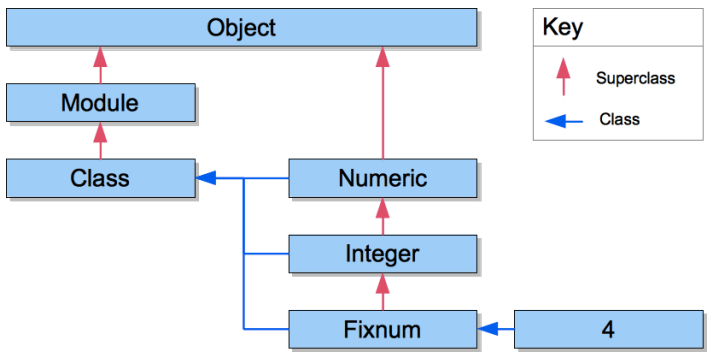
- Ruby supports single inheritance, creating a hierarchy of classes
- The methods `class` and `superclass` can be used to obtain, respectively, the class of an object and the parent of a class

```
>> 4.class
=> Fixnum

>> Fixnum.superclass
=> Integer

>> Integer.superclass
=> Numeric

>> Numeric.superclass
=> Object

>> Object.superclass
=> nil
```

Ruby Metamodel

- As in Ruby everything is an object, classes are themselves instances of the class `Class`

```
>> 4.class.class
=> Class
>> 4.class.superclass
=> Integer
>> Numeric.superclass
=> Object
>> 4.class.superclass.superclass.class
=> Class
```



Defining a Class/1

- A class is made up of a collection of
  - variables representing the internal state and
  - methods providing behaviours that operate on that state
- Class names must begin with a capital letter
  - By convention, names that contain more than one word are run together with each word capitalized, e.g., CamelCase
- Let's start building a class `Customer`

```
>> class Customer
>> end
=> nil
```

- This is the simplest possible class: an empty class (doing nothing)

Defining a Class/2

- Initializing the class and creating variables

```
>> class Customer
>>     @@no_of_customers=0
>>     def initialize(name, addr)
>>        @name=name
>>        @addr=addr
>>        @@no_of_customers = @@no_of_customers + 1
>>     end
>> end
```

- Class variables are prepended with `@@`
  - Belong to the class and have one value per class
- Instance variables are prepended with `@`
  - Belong to the instances/objects and have different values for each instance
  - Need not to be declared, but are dynamically appended to an object when they are first assigned
- The `initialize` method is executed when a new object is created

Accessor Methods for a Class/1

- By default, variables are private and can only be directly accessed within an instance method
- To provide access from outside, accessor methods are needed
- Accessor methods can have the same name as variables
  - ... and 'addr=' is a valid method name
  - When you use them, it looks like you are accessing directly the variables

```
>> class Customer
>>    def set_name( name )
>>       @name = name
>>    end
>>    def get_name
>>       return @name
>>    end
>>    def addr=( addr )
>>       @addr = addr
>>    end
>>    def addr
>>       return @addr
>>    end
>>    def get_no_of_customers
>>       return @@no_of_customers
>>    end
>> end
```

Accessor Methods for a Class/2

- Now we can use our class

```
>> c1 = Customer.new( 'max', 'meran' )
=> #<Customer:0x0000000084fd00 @name="max", @addr="meran">

>> c2 = Customer.new( 'moritz', 'meran' )
=> #<Customer:0x000000008cc4b8 @name="moritz", @addr="meran">

>> c1.addr
=> "meran"

>> c1.addr='bozen'
=> "bozen"

>> c1.addr
=> "bozen"

>> c1.get_no_of_customers
=> 2
```

Accessor Methods for a Class/3

- Since getters and setters are so common, they can be autogenerated

```
>> class Customer
>>     @@no_of_customers=0
>>     attr_accessor :name, :addr
>> end
```

- `attr_accessor` is a method, which is run when Ruby constructs the class object, and it generates the setter and getter methods for you
  - `addr=(addr)`, `addr`, `name=(name)`, and `name`
- If instead `attr :name, :addr` is used, only the getter methods are created

Example Tree Class

- Let's build a class Tree, which allows to create a tree and to traverse it

```
class Tree
   attr_accessor :children, :node_name

   def initialize(name, children=[])
      @node_name = name
      @children = children
   end

   def traverse(&block)
      process &block
      children.each {|c| c.traverse &block}
   end

   def process(&block)
      block.call self
   end
end
```

Using the Tree Class

- Let's create a tree with root node 'Ruby' and children 'Reia' and 'MacRuby'

```
rubytree = Tree.new('Ruby',
                    [Tree.new('Reia'),Tree.new('MacRuby')])
```

- Processing the root node by outputting its name

```
rubytree.process {|node| puts node.node_name}
Ruby
```

- Traversing the whole tree (printing each node name)

```
rubytree.traverse {|node| puts node.node_name}
Ruby
Reia
MacRuby
```

Inheritance

- If we want to create a red-black tree based on our Tree class, we need to add a color and a method for balancing the tree

```
class RedBlackTree < Tree
   attr_accessor :color

   def initialize(name, color, children=[])
      super(name, children)
      @color = color
   end

   def balance()
      ...
   end
end
```

- `<` creates a subclass of an existing class
- `super` calls the `initialize` method of the superclass `Tree`
- In Ruby, a class can only inherit from a single other class

Outline

1 Functions, Code Blocks and Procs

2 Classes

3 **Modules and Mixins**

Modules/1

- A module is a collection of (related) classes, methods/functions and constants

```
module Identifier
    statement1
    statement2
    ...........
end
```

- A module comes together with its own namespace (to avoid name clashes)
- Similar to classes, but
  - there are no instances (objects) of modules
  - there is no inheritance

Modules/2

- An example is the `Math` module

```
>> module Math
>>     PI = 3.141592654
>>     def sqrt( v )
>>         # ...
>>     end
>>     # ...
>> end
```

- It defines various constants, e.g., `PI`, and functions, e.g., `sqrt`
- Constants begin with an uppercase letter

Using Modules

- To use modules, you can use either qualified names

```
>> Math.sqrt(2)
=> 1.4142135623731
>> Math::PI
=> 3.14159265358979
```

- or include the module

```
>> include Math
=> Object
>> sqrt(2)
=> 1.4142135623731
>> PI
=> 3.14159265358979
```

Mixins: Including Modules in Classes

- Modules become really interesting when used in combination with classes
- Ruby does not support multiple inheritance
- However, it does support a mechanism called a mixin
- If we include a module in a class definition, the module's methods are appended to the class
  - Effectively, the module is "mixed in" with the class

Mixins/1

- Unfortunately, it is not that easy to create a mixin
- In order to use the full power of a module in a class, your class may need to implement certain methods
- Let's try to get the tree node names sorted in alphabetical order
- We'll try to do this mixing the module `Enumerable` into our class
  - `Enumerable` provides a method called `sort`

Mixins/2

- Just including the module `Enumerable` in our `Tree` class won't give us the full functionality:

```
>> class Tree
>>     include Enumerable
>>     attr_accessor :children, :node_name
>> ...
>> end
=> nil

>> ruby_tree = Tree.new(...)
=> ...

>> ruby_tree.sort
NoMethodError:  undefined method 'each' for ...
```

- A class wanting to be `enumerable` must implement the method `each` to go through all elements
  - In our `Tree` class, the method needs to go through all nodes

Mixins/3

- Let's add the method each, which needs an object and a code block as parameter

```
>> class Tree
>>    include Enumerable
>>    attr_accessor :children, :node_name
>>    ...
>>    def each(&block)
>>       block.call self
>>       children.each {|c| c.each &block}
>>    end
>> end
```

```
>> ruby_tree.sort
NoMethodError:  undefined method `<=>' for ...
```

- We are missing yet another operator

Mixins/4

- A class wanting to be `comparable` must implement the "spaceship operator" '<=>'
- This operator is used for comparing two objects:

$$a \texttt{ <=> } b = \begin{cases} -1 & \text{if } a < b \\ 1 & \text{if } a > b \\ 0 & \text{if } a = b \end{cases}$$

- Let's add this operator to our `Tree` class

Mixins/5

```
class Tree
   ...
   def <=>(t)
      return -1 if self.node_name < t.node_name
      return 1 if self.node_name > t.node_name
      return 0 if self.node_name == t.node_name
      return nil
   end
end
```

```
>> ruby_tree.sort
=> [#<Tree:0xb73bfa6c @node_name="McRuby",...
```

- The output doesn't look very nice, `sort` returns an array of trees

Success!

- However, an array supports the method `each` as well
- So we can pass a code block to the array, printing the names of the nodes:

```
>> (ruby_tree.sort).each {|n| puts n.node_name}
McRuby
Reia
Ruby
```

- This small example already hints at the flexibility provided by modules, mixins, and code blocks

Success?

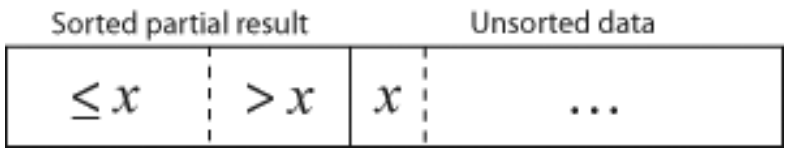
- Well, looking at the code for our `Tree` class you'll notice that some of the code is redundant
- Now that we've implemented the `each` method, we don't need the `traverse` method anymore (which does essentially the same thing)
- So we can refactor the code to make it slimmer and better

Final Tree Class

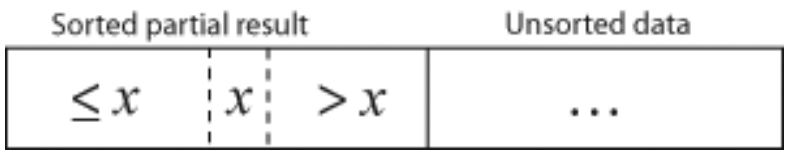
```
class Tree
   include Enumerable
   attr_accessor :children, :node_name

   def initialize(name, children=[])
      @children = children
      @node_name = name
   end

   def each(&block)
      block.call self
      children.each {|c| c.each &block}
   end

   def <=>(tree)
      ...
   end
end
```

Example Insertion Sort/1

- Insertion sort iterates through a list of data, consuming one input element each repetition, and growing a sorted output list
- In each iteration, it removes one element from the input data, finds the correct position in the sorted output list, and inserts it there
- Sorting is typically done in-place, by iterating through the input array from the beginning, and growing the sorted list behind it
- In the array after k iterations the first $k + 1$ entries are sorted
- If the current element is x



- it becomes after this iteration



Example Insertion Sort/2

- Insertion sort in Java

```
class InsertionSort{

   static void sort(int[] a){
      for (int i = 1; i < a.length; i++){
         int val = a[i];
         int j = i-1;

         while (j >= 0 && a[j] > val){
            a[j+1] = a[j];
            j--;
         }
         a[j+1] = val;
      }
   }

   public static void main(String[] args){
      int[] a = {2,43,24,100,3};
      sort(a);
      for (int e :  a)
         System.out.println(e);
   }
}
```

Example Insertion Sort/3

- Insertion sort in Ruby

```
def insertionsort( a )
   a.each_index do |i|
      val = a[i]
      j = i - 1
      while j >= 0 and a[j] > val
         a[j+1], a[j] = a[j], a[j+1]
         j -= 1
      end
   end
end

a = [2,43,24,100,3]
insertionsort(a)
puts a
```

Example Insertion Sort/4

- You can do it even shorter

```
def insertionsort3( a )
   a.each_with_index do |val, i|
      while i > 0 and a[i-1] > val
         a[i], a[i-1] = a[i-1], a[i]
         i -= 1
      end
   end
end
```

- ... or like this:

```
def insertionsort4( a )
   a.each_with_index do |val, i|
      i.downto(1) do |j|
         a[j], a[j-1] = a[j-1], a[j] if a[j-1] > val
      end
   end
end
```

Example Reverse a List

- Reverse a list: `[1,2,3,4]` $\rightarrow$ `[4,3,2,1]`

```
def rev(a)
   return a if a == []
   rev(a.drop(1)).push(a[0])
end
```

Summary

- Strengths of Ruby
  - Ruby is a pure object-oriented language, treating objects in a consistent way
  - Ruby is a strongly typed language, but applies dynamic type checking
  - Supports Duck typing, and is therefore very flexible when it comes to substitutability
  - Some nice features not present in other languages: rich methods on arrays, code blocks, modules and mixins
  - Programmers can be very productive using Ruby, can be used like a scripting language
  - Comes with a very successful web development framework: Ruby on Rails
    - The original Twitter implementation was done in Ruby
- Weaknesses of Ruby
  - Performance: Ruby is not the most efficient language
    - All the flexibility makes it difficult to compile programs
  - Concurrent programming is difficult to do with a state-based language
  - Type Safety: duck typing makes it harder to debug code that has type errors in it