CSE M226: Programming Languages

Lecture 5:Ruby – Advanced Features

Rokan Uddin Faruqui

Dept. of Computer Science and Engineering
University of Chittagong, Bangladesh
Email: rufaruqui@cu.ac.bd

Outline

- Regular Expressions
- Yield
- Inheritance, Module and **Mixins**
- Duck Typing
- Metaprogramming
- Dynamic Typing

Regular Expressions

- **String literals:** find a particular piece of text

Regular Expressions

- **String literals:** find a particular piece of text
- **Anchors:** the beginning and the end of the string, or a word

Regular Expressions

- **String literals:** find a particular piece of text
- **Anchors:** the beginning and the end of the string, or a word
- **Character classes:** define a set of allowed characters

Regular Expressions

- **String literals:** find a particular piece of text
- **Anchors:** the beginning and the end of the string, or a word
- **Character classes:** define a set of allowed characters
- **Quantifiers:** define how often a character is expected to occur

Regular Expressions

- **String literals:** find a particular piece of text
- **Anchors:** the beginning and the end of the string, or a word
- **Character classes:** define a set of allowed characters
- **Quantifiers:** define how often a character is expected to occur
- **Captures:** once found, capture a particular part of the text, so we can use it

Regular Expressions

- **String literals:** find a particular piece of text
- **Anchors:** the beginning and the end of the string, or a word
- **Character classes:** define a set of allowed characters
- **Quantifiers:** define how often a character is expected to occur
- **Captures:** once found, capture a particular part of the text, so we can use it

RE - String literals

```
1 text = "A regular expression is a sequence of
      characters that define a search pattern."
  matches = text.match(/sentence/)
3
```

RE - Anchors

```
text = "A regular expression is a sequence of
    characters that define a search pattern."

puts 'Found "A" at the beginning of the string.' if
    text.match(/^A/)
puts 'Found "O" at the beginning of the string.' if
    text.match(/^O/)

puts 'Found the string "character".' if text.match
    (/character/)
puts 'Found the word "character".' if text.match(/
    character\b/)
```

RE - Character classes

```
text = "A regular expression is a sequence of characters that define a search pattern."
p text.scan(/\b[aeiou][a-z]*\b/)
```

The output will be

```
["expression", "is", "a", "of", "a"]
```

RE - Quantifiers

```
1   p text.scan(/\b[AEIOUaeiou][a-z]+\b/)

3   p text.scan(/\b[AEIOUaeiou][a-z]?\b/)

5   p text.scan(/\b[AEIOUaeiou][a-z]\b/)

7
```

RE - Captures

```
    p text.scan(/\b[A-Za-z]+\b +\b[AEIOUaeiou][a-
z]*\b/)
2
```

RE - Character Classes - shorthand

```
  * `\d` is the same as `[0-9]` (any digit)
  * `\D` is the same as `[^0-9]` (everything except digits)
  * `\w` is the same as `[A-Za-z_\-]`, called *word character* (i.e. this
    allows all lowercase and uppercase latin letters, as well as underscores and
    dashes)
  * `\W` is the same as `[^A-Za-z_\-]` (everything that is not a word character)
  * `\s` means "any whitespace", including spaces, tabs, and linebreaks
  * `\S` everything that is not whitespace
```

Yield

```ruby
def call_block
  puts "Start of method"
  yield
  yield
  puts "End of method"
end

call_block { puts "In the block" }
```

produce

```
Start of method
In the block
In the block
End of method
```

Yield

```
def who_says_what
  yield("Dave", "hello")
  yield("Andy", "goodbye")
end

who_says_what {|person, phrase| puts "#{person}
    says #{phrase}"}
```

produce

```
Dave says hello
Andy says goodbye
```

Yield

```
def who_says_what
  yield("Dave", "hello")
  yield("Andy", "goodbye")
end

who_says_what {|person, phrase| puts "#{person}
    says #{phrase}"}
```

produce

```
Dave says hello
Andy says goodbye
```

Inheritance

```
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end

p = Parent.new
p.say_hello

# Subclass the parent...
class Child < Parent
end

c = Child.new
c.say_hello
```

Inheritance

```
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end

p = Parent.new
p.say_hello

# Subclass the parent...
class Child < Parent
end

c = Child.new
c.say_hello
```

Module

- Modules provide a namespace and prevent name clashes.
- Modules support the mixin facility.

```
module Trig
  PI = 3.141592654
  def Trig.sin(x)
   # ..
  end

  def Trig.cos(x)
  # ..
  end
end
```

Mixins

```
module Debug
  def who_am_i?
"      #{self.class.name} (id: #{self.
    object_id}): #{self.name}"
  end
end

class Phonograph
  include Debug
   attr_reader :name
  def initialize(name)
    @name = name
  end
```

Mixins

```
class EightTrack
  include Debug
  attr_reader :name
 def initialize(name)
   @name = name
 end
end
ph = Phonograph.new("West End Blues")
et = EightTrack.new("Surrealistic Pillow")

```

```
ph.who_am_i? # => "Phonograph (id:
70266478767560): West End Blues"
et.who_am_i? # => "EightTrack (id:
70266478767520): Surrealistic Pillow"
```

Iterators and the Enumerable Module

- The Ruby collection classes (*Array*, *Hash*, and so on) support a large number of operations that do various things with the collection: *traverse it*, *sort it*, and so on.‘

Iterators and the Enumerable Module

- The Ruby collection classes (*Array*, *Hash*, and so on) support a large number of operations that do various things with the collection: *traverse it*, *sort it*, and so on.‘
- Your classes can support all these neat-o features, thanks to the magic of mixins and module Enumerable.

Iterators and the Enumerable Module

- The Ruby collection classes (*Array*, *Hash*, and so on) support a large number of operations that do various things with the collection: *traverse it*, *sort it*, and so on.‘
- Your classes can support all these neat-o features, thanks to the magic of mixins and module Enumerable.
- All you have to do is write an iterator called each, which returns the elements of your collection in turn.

Iterators and the Enumerable Module

- The Ruby collection classes (*Array*, *Hash*, and so on) support a large number of operations that do various things with the collection: *traverse it*, *sort it*, and so on.‘
- Your classes can support all these neat-o features, thanks to the magic of mixins and module Enumerable.
- All you have to do is write an iterator called each, which returns the elements of your collection in turn.

Mixins

```
module Summable
  def sum
    inject(:+)
  end
end

class Array
  include Summable
end

class Range
  include Summable
end
```

Mixins

```
require_relative "vowel_finder"

class VowelFinder
    include Summable
end

[ 1, 2, 3, 4, 5 ].sum # => 15
('a'..'m').sum # => "abcdefghijklm"

vf = VowelFinder.new("the quick brown fox jumped")
vf.sum # => "euiooue"

```

Inheritance, Mixins, and Design

So, when do you use each?

Inheritance and **mixins** both allow you to write code in one place and effectively inject that code into multiple classes.

- First let's look at subclassing. Look out for a "is-a" relationship
- Identify whether it is just for code sharing or "subclass" relationship
- In the past, we?ve tended to gloss over that fact when programming. Because inheritance was the only scheme available for sharing code, we got lazy and said things like "My *Person* class is a subclass of my *DatabaseWrapper* class."

Inheritance, Mixins, and Design

```
class Person
    include Persistable
    # ...
end

```

instead of this:

```
class Person < DataWrapper
# ...
end

```