CS 360 Internet Programming

Ruby

Ruby Essentials

Daniel Zappala
Computer Science
Brigham Young University

Initialize Method

```
class Song
  def initialize(name, artist, duration)
    @name     = name
    @artist   = artist
    @duration = duration
  end
end
song = Song.new("New York, New York","Frank Sinatra",260)
song2 = song
```

- initialize method called when a new object is created
- variables *reference* the object, so song and song2 point to the same object in memory

Access Control

- public methods: can be called by any object
- private methods: can only be called by the instance
- protected methods: can be called by *any* object of the same class or a subclass

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

Inheritance

```
puts song.to_s
-> #<Song:0xb7cb9c1c>
class Song
  def to_s
    "Song: #@name--#@artist (#@duration)"
  end
end

song = Song.new("New York, New York","Frank Sinatra",260)
puts song.to_s
-> "Song: New York, New York--Frank Sinatra (260)"
```

- all objects inherit from class Object
- can override the methods of Object

Defining SubClasses

```
class KaraokeSong < Song
  def initialize(name, artist, duration, lyrics)
    super(name, artist, duration)
    @lyrics = lyrics
  end
  def to_s
    super + " [#@lyrics]"
  end
end

song = KaraokeSong.new("New York, New York","Frank Sinatra",260,"Start sp
puts song.to_s
-> "Song: New York, New York--Frank Sinatra (260) [Start spreading the ne
```

- KaraokeSong is a *subclass* of Song, Song is the *superclass*
- super calls the same method of the superclass

Attributes

```
class Song
  def name
    @name
  end
end
```

- instance variables are private unless you expose them via methods

```
class Song
  attr_reader :name, :artist, :duration
end
```

- attr_reader shortcut: the same as creating a method that returns the value of an instance variable
- automatically creates instance variables

Writable Attributes

```
class Song
  def duration=(new_duration)
    @duration = new_duration
  end
end
song.duration = 257
```

- if method name ends with an = symbol, Ruby lets you use = as an assignment operator for the attribute

```
class Song
  attr_writer :duration
end
```

- attr_writer shortcut
- attr_accessor shortcut declares attribute as both readable and writeable

Class Variables

```
class Song
  @@plays = 0
  def initialize(name, artist, duration)
    @name     = name
    @artist   = artist
    @duration = duration
    @plays    = 0
  end
  def play
    @plays  += 1
    @@plays += 1
    "This  song: #@plays plays. Total #@@plays plays."
  end
end
```

- shared among all instances

Class Methods

- needed when a class method must work without being tied to a particular instance
- examples: Song.new, File.delete
- prefaced by the class name and a period

```
class Song
  def Song.version
    "1.0-r1"
  end
end

puts Song.version
-> 1.0-r1
```

Containers

```
class SongList          # wrapper around an array
  def initialize
    @songs = Array.new
  end
  def append(song)
    @songs.push(song)
    self
  end
  def delete_first
    @songs.shift
  end
  def delete_last
    @songs.pop
  end
  def [](index)         # defines [] method
    @songs[index]
  end
```

Using Iterators

- for loop version

```
def with_title(title)
  for i in 0...@songs.length
    return @songs[i] if title == @songs[i].name
  end
  return nil
end
```

- iterator version
- requires less knowledge about array implementation

```
def with_title(title)
  @songs.find {|song| title == song.name }
end
```

Common Iterators

```
[1, 3, 5, 7, 9].find {|v| v*v > 30 } -> 7

[1, 3, 5, 7, 9].each {|i| print i } -> 13579

[1, 3, 5, 7, 9].collect {|x| x.succ } -> [2, 4, 6, 8, 10]

[1, 3, 5, 7, 9].inject {|sum, element| sum+element} -> 16
```

Creating an Iterator

```
class File
  def File.my_open(*args)
    result = file = File.new(*args)
    if block_given?
      result = yield file
      file.close
    end
    return result
  end
end
```

- use the iterator to define a block that must be run as a transaction
- example: a file open method that ensures the file closes itself when done
- returns an open file if no block given

Strings

- single-quoted
  - `\\` makes `\`
  - `\'` makes `'`
- double-quoted
  - many more escape sequences, e.g. `\n` and `\t`
  - value substitution with `#{expr}`

String Parsing

- file format:

```
/jazz/j00132.mp3   | 3:45 | Fats     Waller     | Ain't Misbehavin'
/jazz/j00319.mp3   | 2:58 | Louis    Armstrong  | Wonderful World
/bgrass/bg0732.mp3| 4:09 | Strength in Numbers | Texas Red
         :              :        :                     :
```

- parser:

```
File.open("songdata") do |song_file|
  songs = SongList.new
  song_file.each do |line|
    file, length, name, title = line.chomp.split(/\s*\|\s*/)
    songs.append(Song.new(title, name, length))
  end
  puts songs[1]
end
-> Song: Wonderful World--Louis    Armstrong (2:58)
```

Additional String Methods

- extra spaces in the artist name
  - use `name.squeeze!(" ")`
  - the ! modifies the name in place
- convert 2:58 into seconds
  - use `mins,secs = length.split(/:/)` and then convert
  - or use `mins,secs = length.scan(/\d+/)`, with regular expression
- see the library reference for more

Defining a Method

- special method names
  - trailing ?: acts as a query
  - trailing !: dangerous, or modifies the calling object
  - trailing =: may be used for assignment
- default arguments

```
def cool_dude(arg1="Miles", arg2="Coltrane", arg3="Roach")
  "#{arg1}, #{arg2}, #{arg3}."
end
cool_dude -> Miles, Coltrane, Roach
cool_dude("Bart") -> Bart, Coltrane, Roach
cool_dude("Bart", "Elwood") -> Bart, Elwood, Roach
cool_dude("Bart", "Elwood", "Linus") -> Bart, Elwood, Linus
```

Variable-Length Argument Lists

```
def varargs(arg1, *rest)
  "Got #{arg1} and #{rest.join(', ')}"
end
varargs("one") -> "Got one and "
varargs("one", "two") -> "Got one and two"
varargs "one", "two", "three" -> "Got one and two, three"
```

Return Values

- returns value of last statement executed
- can return more than one value in an array

```
def meth_three
 100.times do |num|
   square = num*num
   return num, square if square > 1000
 end
end
num, square = meth_three
```