Ruby Issues 2013...

Mark Keane...CSI...UCD

Pausing for breath...

A: Spatt !!!!

B: Hash...any Hash...

C: symbols...again...

D: rescue me...

E: Regexp

F: Dotty....

G: Script sequences

H: Super

I: Privacy

J: Iterators

Part A:

Splat...

Splats – what are they and where are they typically useful?

*Splats...

- are most useful for allowing a method to take varying numbers of arguments...I want this method to take 2 arguments but sometimes there may be 3 or 6
- or where you are unsure what the argument input might be...

```ruby
def add_numbers(*nos)
    p nos
    nos.reduce(:+)
end


def arith(fn, *nos)
    p [fn] + nos
    nos.reduce(fn)
end

p add_numbers(1)
p add_numbers(1,3,5,7)
p add_numbers(1, 4)

p arith(:+, 1, 4, 5, 6)
p arith(:*, 1, 4, 5, 6)
```

```
$ ruby stuff.rb
[1]
1
[1, 3, 5, 7]
16
[1, 4]
5
[:+, 1, 4, 5, 6]
16
[:*, 1, 4, 5, 6]
120
```

Part B:

Hash...

Hashes – hashes vs. arrays, why/when is one preferred over the other?

Hashes – what are they and how can one easily access keys and values?

Dweadful Hash I...

- Most of the time it makes sense to be dealing with arrays of objects, especially if you are accessing many different attributes of them regularly
- Eg. I might have an array of songs and different methods that compute album times, count songs by artists, build libraries of albums etc...
- But, the cost is that you always need to iterate over that array to do something to its elements (with **each, for, select, collect, while**....)
- So, processing time and memory will increase as array embiggens...and arrays are copied on use

Dweadful Hash II...

- A hash table allows you to access an item directly, at a relatively constant processing time cost and is memory efficient
- It is often an optimisation step
- It is appropriate where you want a look-up table to find a specific (attribute) value (n.b., set hash to global variable **$Matches[34444]**)
- Eg in similiarity program, I store commonly used attribute-values for certain items in a hash; I also have a hash to access match-objects in the match space; though most processing is done with the arrays of keys from that hash-table

```ruby
foo = Hash.new
foo[:a] = 1
foo[:b] = 2
foo[:c] = 3

p foo

p foo[:c]

foo[:c] = "hi"

p foo

p foo[:a] + foo[:b] + foo[:b]

p foo.keys

p foo.values
```

stuff2.rb

```
{:a=>1, :b=>2, :c=>3}
3
{:a=>1, :b=>2, :c=>"hi"}
5
[:a, :b, :c]
[1, 2, "hi"]
```

Part C:

Previous stuff we saw on :symbols...

Could you cover symbols again in the class? For example when best to use them and when to avoid them. I'm not too sure about what happens if I define a symbol in a ruby script that is already defined somewhere else, for example, in some gem program that the ruby script is using

Symbols...

- There should not be any mystery about symbols
- They are immutable, single stored entities
- As such, they are often used as keys in hashes, to name methods or as keywords in pre-defined kernel methods ; e.g., **:+** or **:attr_accessor**
- Apart from their usefulness in taking up less space, it is important that they look much different in your code; if you see a symbol you know it is something in your program with will never change

:hello not “hello”

- a symbol is a good way to pass around string information if (i) that string does not have to be changed at runtime (symbols are immutable) and (ii) you do not need to apply string-methods to it
- symbols are more efficient (internally represented as a number), whereas string comparison invites a walking comparison of each char, symbol comparison is just no-identical?
- symbols use less memory as once created, the same one is used throughout; identical strings are each separately stored entities

Eg

```
$ irb
irb(main):001:0> :hello.object_id
=> 484168
irb(main):002:0> :hello.object_id
=> 484168
irb(main):003:0> "hello".object_id
=> 70269428749760
irb(main):004:0> "hello".object_id
=> 70269428718300
irb(main):005:0>
```

main uses...flags and keys in hash tables...in yaml

Bad Interactions

- Hard to say...
- If a symbol is used in a gem then it will, by definition, be identical to one that you create
- But, its context in the gem should help avoid most clashes

  attr_accessor :name, :id, :time

- ...is part of the syntax of class-attribute definition

```ruby
class Big
   def initialize()
     @arr1 = [:fi, :fie, :foe]
   end

   def get_arr
      @arr1
   end
end


class Small
   def initialize()
     @arr2 = [:fi, :fie1, :foe]
   end

   def get_arr1
      @arr2
   end
end

a = Big.new
b = Small.new

p a.get_arr
p b.get_arr1

p (a.get_arr)[0] == (b.get_arr1)[0]
p (a.get_arr)[1] == (b.get_arr1)[1]
p (a.get_arr)[2] == (b.get_arr1)[2]
```

```
$ ruby symbol.rb
[:fi, :fie, :foe]
[:fi, :fie1, :foe]
true
false
true
```

symbol.rb

```
class Big
   def initialize()
     @arr1 = Hash.new
   end

   def get_arr
      @arr1
   end

   def update1
     @arr1[:fi] = 1
     @arr1[:fo] = 2
 p @arr1
   end
end

class Small
   def initialize()
     @arr2 = Hash.new
   end

   def get_arr1
      @arr2
   end

   def update2
     @arr2[:fi] = 3
     @arr2[:fum] = 4
 p @arr2
   end
end
```

hash_symbol.rb

```
class Big
   def initialize()
     @arr1 = Hash.new
   end

   def get_arr
      @arr1
   end

   def update1
     @arr1[:fi] = 1
     @arr1[:fo] = 2
 p @arr1
   end
end

class Small
   def initialize()
     @arr2 = Hash.new
   end

   def get_arr1
      @arr2
   end

   def update2
     @arr2[:fi] = 3
     @arr2[:fum] = 4
 p @arr2
   end
end
```

```
a = Big.new; a.update1
b = Small.new; b.update2
p (a.get_arr).merge(b.get_arr1)
```

```
$ ruby hash_symbol.rb
{:fi=>1, :fo=>2}
{:fi=>3, :fum=>4}
{:fi=>3, :fo=>2, :fum=>4}
```

hash_symbol.rb

Troubleshooters.Com, Code Corner and Ruby Revival Present

The Ruby_Newbie Guide to Symbols

Copyright (C) 2005 by Steve Litt

The following statements are handy in using (or not using) symbols:

- A Ruby symbol looks like a colon followed by characters. (`:mysymbol`)
- A Ruby symbol is a *thing* that has both a number (integer) and a string.
- The value of a Ruby symbol's string part is the name of the symbol, minus the leading colon.
- A Ruby symbol cannot be changed at runtime.
- Neither its string representation nor its integer representation can be changed at runtime.
- Ruby symbols are useful in *preventing* modification.
- Like most other things in Ruby, a symbol is an object.
- When designing a program, you can usually use a string instead of a symbol.
    - Except when you must guarantee that the string isn't modified.
- Symbol objects do not have the rich set of instance methods that String objects do.
- After the first usage of `:mysymbol` all further useages of `:mysymbol` take no further memory -- they're all the same object.
- Ruby symbols save memory over large numbers of identical literal strings.
- Ruby symbols enhance runtime speed to at least some degree.

```
$ irb

irb(main):001:0> :this-is-a
NameError: undefined local variable or method `is' for main:Object
        from (irb):1
        from /opt/local/bin/irb:12:in `<main>'

002:0> :this_is_a_really_long_symbol_but_I_would_not_encourage_you_to_use_them
=> :this_is_a_really_long_symbol_but_I_would_not_encourage_you_to_use_them
```

Part D:

Catching...Throwing...Rescue & Exceptions

I would like to know more about the error catching thing you mentioned last week i.e."begin...rescue"- is there a big set of predefined errors/exception classes like in Java?

Catch...Rescue and Errors...

- We did not look at **catch** and **throw**, those they exist to break out of loops and blocks; you can also **raise StandardError.new("yuk")**
- We did look at break and rescue which have definite roles in handling exceptions
- Yes, there are loads of error classes, all are sub-classes of the **Exception Class**

```
begin

# Any number of ruby statements here
# Usually, they are executed without exceptions and
# execution continues after the end statement

rescue

# This is the rescue clause; exception-handling code goes here
# If an exception is raised by the code above, or propagates up
# from one of the methods called above, then the execution jumps here

end
```

stuff3.rb

```
puts "BAD PLUS TEST*******"
def bad_plus(x, y)
  begin
      sum = x + y
  rescue
    p $!
    puts "Have Fixed Crappy Args"
    sum = x + y[0]
  end
  puts “Sum is: #{sum}”
end

bad_plus(3, 4)
bad_plus(3,[5, 6, 7])
```

stuff3.rb

```
puts "BAD PLUS TEST*******"
def bad_plus(x, y)
  begin
      sum = x + y
  rescue
    p $!
    puts "Have Fixed Crappy Args"
    sum = x + y[0]
  end
  puts “Sum is: #{sum}”
end

bad_plus(3, 4)
bad_plus(3,[5, 6, 7])
```

```
$ ruby stuff3.rb
BAD PLUS TEST*******
Sum is: 7
#<TypeError: Array can't be coerced into Fixnum>
Have Fixed Crappy Args
Sum is: 8
```

```
puts "BAD PLUS EXTRA TEST*******"
def bad_plus_extra(x, y)
  begin
     sum = x + y
  rescue => exceptcase
     p exceptcase
     puts "Have Fixed Crappy Args"
     sum = x + y[1]
  end
  puts "Sum is: #{sum}"
  p exceptcase
end

bad_plus_extra(3, 4)
bad_plus_extra(3,[5, 6, 7])
```

stuff3.rb

```ruby
puts "BAD PLUS EXTRA TEST*******"
def bad_plus_extra(x, y)
  begin
     sum = x + y
  rescue => exceptcase
    p exceptcase
     puts "Have Fixed Crappy Args"
     sum = x + y[1]
  end
  puts "Sum is: #{sum}"
  p exceptcase
end

bad_plus_extra(3, 4)
bad_plus_extra(3,[5, 6, 7])
```

```
$ ruby stuff3.rb

BAD PLUS EXTRA TEST*******
Sum is: 7
nil
#<TypeError: Array can't be coerced into Fixnum>
Have Fixed Crappy Args
Sum is: 9
#<TypeError: Array can't be coerced into Fixnum>
```

```
puts "BAD SUM TEST*******"

def bad_sum(x, y, z)
  begin
     if z == 0 then mysum(x, y) else mysum(x,y,z) end
  rescue TypeError => exceptcase
     puts "Crappy type detected: #{exceptcase}"
  rescue ArgumentError => exceptcase
     puts "Crappy type detected: #{exceptcase}"
  end
end

def mysum(x, y, z)
   p (x + (y + z))
end

bad_sum(2, 3, 4)
bad_sum(2, 3, 0)
bad_sum(2, 3, [4])
```

stuff3.rb

```ruby
puts "BAD SUM TEST*******"

def bad_sum(x, y, z)
  begin
    if z == 0 then mysum(x, y) else mysum(x,y,z) end
  rescue TypeError => exceptcase
    puts "Crappy type detected: #{exceptcase}"
  rescue ArgumentError => exceptcase
    puts "Crappy arg detected: #{exceptcase}"
  end
end

def mysum(x, y, z)
    p (x + (y + z))
end

bad_sum(2, 3, 4)
bad_sum(2, 3, 0)
bad_sum(2, 3, [4])
```

```
$ ruby stuff3.rb
BAD SUM TEST*******

9

Crappy arg detected: wrong number of
arguments (2 for 3)

Crappy type detected: Array can't be coerced
into Fixnum
```

Ruby Exception Class Hierarchy...

```
+--Exception
    +--NoMemoryError
    +--ScriptError
    |   +--LoadError
    |   +--NotImplementedError
    |   +--SyntaxError
    +--SecurityError          # Was a StandardError in 1.8
    +--SignalException
    |   +--Interrupt
    +--SystemExit
    +--SystemStackError       # Was a StandardError in 1.8
    +--StandardError
        +--ArgumentError
        +--FiberError         # New in 1.9
        +--IOError
        |   +--EOFError
        +--IndexError
        |   +--KeyError       # New in 1.9
        |   +--StopIteration # New in 1.9
        +--LocalJumpError
        +--NameError
        |   +--NoMethodError
        +--RangeError
        |   +--FloatDomainError
        +--RegexpError
        +--RuntimeError
        +--SystemCallError
        +--ThreadError
        +--TypeError
        +--ZeroDivisionError
```

What Normal Errors Do...

- When an Exception object is generated by code, it becomes the thing returned by the block or method concerned
- As it rises out of the block/method it looks for a rescue statement; if it finds one it checks itself against (if it specifies a particular Exception)
- If there is no rescue statement then it exits all the way out to the top-level stopping execution and generating the error at the terminal

Part E:

Regexp Class

I also don't really understand the point of Regexp's - why were these methods not just added to the String class, why is there a whole separate class for them?

The History Argument...

Regular expression

From Wikipedia, the free encyclopedia

In computing, a **regular expression** provides a concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters. Common abbreviations for "regular expression" include **regex** and **regexp**.

The concept of regular expressions was first popularized by utilities provided by Unix distributions, in particular the editor ed and the filter grep.[*citation needed*] A regular expression is written in a formal language that can be interpreted by a regular expression processor, which is a program that either serves as a parser generator or examines text and identifies parts that match the provided specification. Historically, the concept of regular expressions is associated with Kleene's formalism of regular sets, introduced in the 1950s.

The Basics Argument...

Basic concepts

[edit]

A regular expression, often called a pattern, is an expression that specifies a set of strings. To specify such sets of strings, rules are often more concise than lists of a set's members. For example, the set containing the three strings "Handel", "Händel", and "Haendel" can be specified by the pattern `H(ä|ae?)ndel` (or alternatively, it is said that the pattern *matches* each of the three strings). In most formalisms, if there exists at least one regex that matches a particular set then there exist an infinite number of such expressions. Most formalisms provide the following operations to construct regular expressions.

Boolean "or"

A vertical bar separates alternatives. For example, `gray|grey` can match "gray" or "grey".

Grouping

Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|grey` and `gr(a|e)y` are equivalent patterns which both describe the set of "gray" and "grey".

Quantification

A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark `?`, the asterisk `*` (derived from the Kleene star), and the plus sign `+` (Kleene cross).

- `?` The question mark indicates there is *zero or one* of the preceding element. For example, `colou?r` matches both "color" and "colour".
- `*` The asterisk indicates there is *zero or more* of the preceding element. For example, `ab*c` matches "ac", "abc", "abbc", "abbbc", and so on.
- `+` The plus sign indicates there is *one or more* of the preceding element. For example, `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac".

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations +, −, ×, and ÷. For example, `H(ae?|ä)ndel` and `H(a|ae|ä)ndel` are both valid patterns which match the same strings as the earlier example, `H(ä|ae?)ndel`.

The precise syntax for regular expressions varies among tools and with context; more detail is given in the *Syntax* section.

The Formal Argument...

Formal definition

[edit]

Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory.[5][6] Given a finite alphabet Σ, the following constants are defined as regular expressions:

- (*empty set*) ∅ denoting the set ∅.
- (*empty string*) ε denoting the set containing only the "empty" string, which has no characters at all.
- (*literal character*) *a* in Σ denoting the set containing only the character *a*.

Given regular expressions R and S, the following operations over them are defined to produce regular expressions:

- (*concatenation*) *RS* denoting the set { αβ | α in set described by expression *R* and β in set described by *S* }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- (*alternation*) *R* | *S* denoting the set union of sets described by *R* and *S*. For example, if R describes {"ab", "c"} and S describes {"ab", "d", "ef"}, expression *R* | *S* describes {"ab", "c", "d", "ef"}.
- (*Kleene star*) R^* denoting the smallest superset of set described by *R* that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from set described by *R*. For example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then alternation. If there is no ambiguity then parentheses may be omitted. For example, `(ab)c` can be written as `abc`, and `a|(b(c*))` can be written as `a|bc*`. Many textbooks use the symbols ∪, +, or ∨ for alternation instead of the vertical bar.

Examples:

- `a|b*` denotes {ε, "a", "b", "bb", "bbb", ...}
- `(a|b)*` denotes the set of all strings with no symbols other than "a" and "b", including the empty string: {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", ...}
- `ab*(c|ε)` denotes the set of strings starting with "a", then zero or more "b"s and finally optionally a "c": {"a", "ac", "ab", "abc", "abb", "abbc", ...}

Expressive power and compactness

[edit]

The formal definition of regular expressions is purposely parsimonious and avoids defining the redundant quantifiers `?` and `+`, which can be expressed as follows: `a+` = `aa*`, and `a?` = `(a|ε)`. Sometimes the complement operator is added, to give a *generalized regular expression*; here R^C matches all strings over Σ^* that do not match *R*. In principle, the complement operator is redundant, as it can always be circumscribed by using the other operators. However, the process for computing such a representation is complex, and the result may require expressions of a size that is double exponentially larger.[7][8]

Regular expressions in this sense can express the regular languages, exactly the class of languages accepted by deterministic finite automata. There is, however, a significant difference in compactness. Some classes of regular languages can only be described by deterministic finite automata whose size grows exponentially in the size of the shortest equivalent regular expressions. The standard

Why Bother...

- They are not stings
- As objects they are a very different type of thing that captures sets of strings rather than one string
- As pattern-objects they are used to search strings; if the pattern was always a string then you might not need them, but when it is something more general than a string, it is...well...different

Part F:

Dotty...

Could you clarify the difference between the dot operator (.) and the double colon operator (::) ?

NO

```
class Foo
    def check
        p "foo"
    end

    def self.checko
        p "foo class method"
    end
end

x = Foo.new
x.check
Foo::checko
Foo.checko

#Eeeeeeeeeekk
Foo::check
```

foo.rb

```
class Foo
    def check
        p "foo"
    end

    def self.checko
        p "foo class method"
    end
end

x = Foo.new
x.check
Foo::checko
Foo.checko

#Eeeeeeeeeekk
Foo::check
```

foo.rb

```
$ ruby foo.rb
"foo"
"foo class method"
"foo class method"
foo.rb:17:in `<main>': undefined method `check' for
Foo:Class (NoMethodError)
```

```
module Freaky
  class Basic
    def do_it
        p "do it"
    end

    def self.do_again
      p "do again class method"
    end
  end
end
b = Freaky::Basic.new
b.do_it
Freaky::Basic.do_again
Freaky::Basic::do_again
#Badddddd
Freaky.Basic.do_again
b = Basic.new
b.do_it
```

foo.rb

```
module Freaky
  class Basic
    def do_it
        p "do it"
    end

    def self.do_again
      p "do again class method"
    end
  end
end
b = Freaky::Basic.new
b.do_it
Freaky::Basic.do_again
Freaky::Basic::do_again
#Badddddd
Freaky.Basic.do_again
b = Basic.new
b.do_it
```

```
$ ruby foo.rb
"do it"
"do again class method"
"do again class method"
```

foo.rb

```
module Freaky
  class Basic
    def do_it
        p "do it"
    end

    def self.do_again
      p "do again class method"
    end
  end
end
b = Freaky::Basic.new
b.do_it
Freaky::Basic.do_again
Freaky::Basic::do_again
#Badddddd
Freaky.Basic.do_again
b = Basic.new
b.do_it
```

```
$ ruby foo.rb
"do it"
"do again class method"
"do again class method"
```

```
foo.rb:42:in `<main>': undefined method `Basic' for
Freaky:Module (NoMethodError)
```

foo.rb

```
module Freaky
  class Basic
    def do_it
        p "do it"
    end

    def self.do_again
      p "do again class method"
    end
  end
end
b = Freaky::Basic.new
b.do_it
Freaky::Basic.do_again
Freaky::Basic::do_again
#Badddddd
Freaky.Basic.do_again
b = Basic.new
b.do_it
```

```
$ ruby foo.rb
"do it"
"do again class method"
"do again class method"
```

```
foo.rb:42:in `<main>': undefined method `Basic' for
Freaky:Module (NoMethodError)
```

```
foo.rb:46:in `<main>': uninitialized constant
Object::Basic (NameError)  -- at Basic.new
```

Part G:

Script sequences

If Ruby is a scripting language, essentially a sequence of commands for the Ruby interpreter, then does the order of your code matter? For example, if I define a class in my code AFTER I have called it, will this cause trouble? Will order of statements ever cause trouble?

YES

```
Bloggie.please

class Bloggie
    def self.please
      p "please"
    end
end
```

bloggie.rb

```
$ ruby bloggie.rb
bloggie.rb:2:in `<main>': uninitialized constant Object::Bloggie (NameError)
```

```
class Bloggie
    def self.please
      p "please"
    end
end

Bloggie.please
```

bloggie.rb

```
$ ruby bloggie.rb
"please"
```

Order Matters I...

- So, Ruby will do everything in the sequence in which it is written
- If you call a class or method or module before you have defined it then it will throw the head...
- One mysterion, is what **require** does; rem. require loads the named file and executes it (so all the definitions in this other file are before other statements in the toplevel file)

Order Matters II...

- Of course, Michael, the flow of control from one statement to the next can be modified by **conditionals** (if x is the case, do this condition not that) or **throws/catches** or breaks
- But that, baby puppy, is a story for another day

Part H:

Super...

What about using **super** in inheritace?

```
class Foo
  def foo_it
    puts "Do foo first..."
  end
end

class FooLet
  def foo_it
    super
    puts "Then do the rest..."
  end
end

a =  Foo.new
b = FooLet.new
a.foo_it
puts "\n"
b.foo_it
```

super.rb

```
class Foo
  def foo_it
    puts "Do foo first..."
  end
end

class FooLet
  def foo_it
    super
    puts "Then do the rest..."
  end
end

a =  Foo.new
b = FooLet.new
a.foo_it
puts "\n"
b.foo_it
```

super.rb

```
$ ruby super.rb
"Do foo first...

super:10:in `foo_it': super: no superclass
method `foo_it' for #<FooLet:0x007fe9220e3478>
(NoMethodError)
```

```ruby
class Foo
  def foo_it
    puts "Do foo first..."
  end
end

class FooLet < Foo
  def foo_it
    super
    puts "Then do the rest..."
  end
end

a =  Foo.new
b = FooLet.new
a.foo_it
puts "\n"
b.foo_it
```

super.rb

```
$ ruby super.rb
Do foo first...

Do foo first...
Then do the rest...
$
```

```
class Foo
  def initialize(a, b)
    @a = a
    @b = b
    @count = 2
  end
end

class FooLet < Foo
  def initialize(a1, b1)
    super
    @count = 4
  end
end

a =  Foo.new(:mark, :keane)
p a
b = FooLet.new(:donald, :duck)
p b
```

super_init.rb

```ruby
class Foo
  def initialize(a, b)
    @a = a
    @b = b
    @count = 2
  end
end

class FooLet < Foo
  def initialize(a1, b1)
    super
    @count = 4
  end
end

a =  Foo.new(:mark, :keane)
p a
b = FooLet.new(:donald, :duck)
p b
```

super_init.rb

```
$ ruby super_init.rb
#<Foo:0x007fbb7d8bf0c0 @a=:mark, @b=:keane, @count=2>
#<FooLet:0x007fbb7d8bef30 @a=:donald, @b=:duck, @count=4>
```

Part I:

Privacy again...

Privacy in Classes II

- *public methods* can be invoked from anywhere, no access control is enforced; all methods are public by default (except **initialize** which is always private)
- *private methods* are internal to the implementation of a class, can only be invoked by other instance methods of the class (or its sub-classes); always invoked in *functional style*, not on ***object*** *or* ***self***
- *protected methods* are like private in that they are internal to the class, but can be invoked on **self** (but it is more complicated than this; see p.233 F&M)

Privacy in Classes II

- *public methods* can be called by anyone -- no access control is enforced; all methods are public by default (except **initialize** which is always private)

except "global" methods defined outside classes, like at the top: **REM**

Privacy in Classes II

- *public methods* can be invoked from anywhere, no access control is enforced; all methods are public by default (except **initialize** which is always private)
- *private methods* are internal to the implementation of a class, can only be invoked by other instance methods of the class (or its sub-classes); always invoked in *functional style*, not on ***object*** *or* ***self***
- *protected methods* are like private in that they are internal to the class, but can be invoked on **self** (but it is more complicated than this; se p.233 F&M)

Privacy Eg.1a

```
class Testo
 def initialize
     @foo = "heeeee"
 end

 def santa
     @foo + "   Hooooo"
 end

 def wicked_elf
     @foo + @foo + @foo
 end
 def snooty
     "I do not laugh"
 end
end
test = Testo.new
test.initialize
#privacy.rb:21: private method `initialize' called for #<Testo:0x27f38
#@foo="heeeee"> (NoMethodError)

p test.santa
p test.wicked_elf
p test.snooty
```

```
$ ruby privacy.rb
 "heeeee  Hooooo"
 "heeeeeheeeeeheeeee"
 "I do not laugh"
```

privacy.rb

Privacy Eg.1b

```
class Testo
 def initialize
     @foo = "heeeee"
 end

 def santa
     @foo + "  Hooooo"
 end

 def wicked_elf
     @foo + @foo + @foo
 end
 def snooty
     "I do not laugh"
 end
     public :santa
     protected :wicked_elf
     private :snooty
end
test = Testo.new
p test.santa
p test.wicked_elf
p test.snooty
```

```
$ ruby privacy.rb
=> "heeeee  Hooooo"
=> protected method `wicked_elf' called for
#<Testo:0x27aec @foo="heeeee">
(NoMethodError)
=> private method `snooty' called for #<Testo:
0x27b64 @foo="heeeee"> (NoMethodError)
```

privacy.rb

Privacy Eg.2a

```
lass Testo
 def initialize
     @foo = "heeeee"
 end

 def santa
     p @foo + "  Hooooo, " + snooty
     p self.wicked_elf
 end

 def wicked_elf
     @foo + @foo + @foo
 end

 def snooty
     "I do not laugh"
 end

 public :santa
 protected :wicked_elf
 private :snooty
end

test = Testo.new
test.santa
puts "**********"
test.wicked_elf
```

```
$ ruby privacy_a.rb
 "heeeee  Hooooo, I do not laugh"
 "heeeeeheeeeeheeeee"
**********
protected method `wicked_elf' called for <Testo:
0x27a10 @foo="heeeee"> (NoMethodError)
```

privacy_a.rb

Privacy Eg.2b

```
lass Testo
 def initialize
     @foo = "heeeee"
 end

 def santa
     puts "Hooooo " + self.wicked_elf
     puts self.snooty
 end

 def wicked_elf
     “I am a laughing elf”
 end

 def snooty
     "I do not laugh"
 end

 public :santa
 protected :wicked_elf
 private :snooty
end

test = Testo.new
test.santa
```

```
$ ruby privacy_b.rb
 "Hooooo I am a laughing elf"
...private method `snooty' called for #<Testo:
0x11a594> (NoMethodError)
```

privacy_b.rb

Part J:

Iterators...

```
my_array = [:tom, "and", :joe, :are, :twins, "and", :so]

p my_array.each{|item|  [item] + ["blibb"]}
```

```
[:tom, "and", :joe, :are, :twins, "and", :so]
```

```
$ ruby foo.rb
"do it"
"do again class method"
"do again class method"
```

iterators.rb

```
my_array = [:tom, "and", :joe, :are, :twins, "and", :so]

p my_array.each{|item|  [item] + ["blibb"]}
```

```
[:tom, "and", :joe, :are, :twins, "and", :so]
```

```
my_array.each{|item| p [item] + ["blibb"]}
```

```
[:tom, "blibb"]
["and", "blibb"]
[:joe, "blibb"]
[:are, "blibb"]
[:twins, "blibb"]
["and", "blibb"]
[:so, "blibb"]
```

iterators.rb

```
my_array = [:tom, "and", :joe, :are, :twins, "and", :so]

p my_array.each{|item|  [item] + ["blibb"]}

my_array.each{|item| p [item] + ["blibb"]}

p my_array.select{|item| item.is_a?(Symbol)}
```

```
[:tom, "and", :joe, :are, :twins, "and", :so]
```

```
[:tom, "blibb"]
["and", "blibb"]
[:joe, "blibb"]
[:are, "blibb"]
[:twins, "blibb"]
["and", "blibb"]
[:so, "blibb"]
```

```
[:tom, :joe, :are, :twins, :so]
```

iterators.rb

```
my_array = [:tom, "and", :joe, :are, :twins, "and", :so]

p my_array.each{|item|  [item] + ["blibb"]}

my_array.each{|item| p [item] + ["blibb"]}

p my_array.select{|item| item.is_a?(Symbol)}

p my_array.collect{|item| item.to_s[0..2]}
```

```
[:tom, "and", :joe, :are, :twins, "and", :so]
```

```
[:tom, "blibb"]
["and", "blibb"]
[:joe, "blibb"]
[:are, "blibb"]
[:twins, "blibb"]
["and", "blibb"]
[:so, "blibb"]
```

```
[:tom, :joe, :are, :twins, :so]
```

```
["tom", "and", "joe", "are", "twi", "and", "so"]
```

iterators.rb

```
my_array = [:tom, "and", :joe, :are, :twins, "and", :so]

p my_array.each{|item|  [item] + ["blibb"]}

my_array.each{|item| p [item] + ["blibb"]}

p my_array.select{|item| item.is_a?(Symbol)}

p my_array.collect{|item| item.to_s[0..2]}

p my_array.map{|item| item.to_s[0..2]}
```

```
[:tom, "and", :joe, :are, :twins, "and", :so]
```

```
[:tom, "blibb"]
["and", "blibb"]
[:joe, "blibb"]
[:are, "blibb"]
[:twins, "blibb"]
["and", "blibb"]
[:so, "blibb"]
```

```
[:tom, :joe, :are, :twins, :so]
```

```
["tom", "and", "joe", "are", "twi", "and", "so"]
```

iterators.rb

```
my_array = [:tom, "and", :joe, :are, :twins, "and", :so]

p my_array.each{|item|  [item] + ["blibb"]}

my_array.each{|item| p [item] + ["blibb"]}

p my_array.select{|item| item.is_a?(Symbol)}

p my_array.collect{|item| item.to_s[0..2]}

p my_array.map{|item| item.to_s[0..1]}

p my_array.inject{|a, b| [a, b]}
```

iterators.rb

```
my_array = [:tom, "and", :joe, :are, :twins, "and", :so]

p my_array.inject{|a, b| [a, b]}
```

[:tom, “and”]

[[:tom, “and”], :joe]

[[[:tom, “and”], :joe], :twins]

[[[[:tom, “and”], :joe], :twins],”and”]

[[[[[[:tom, "and"], :joe], :are], :twins], "and"], :so]

iterators.rb