Software Engineering
ECE444

Ruby

Ruby is…

- a scripting language like Perl/PHP/JavaScript/etc.
- interpreted
- minimalist language, but with rich libraries

Homework:

1. get Ruby on your computer:
   preinstalled on Mac;
   otherwise download & install it

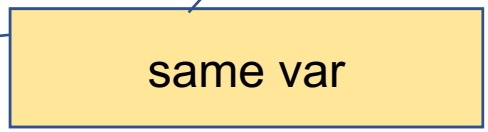
2. run irb # interactive Ruby
   alternatively: https://ruby.github.io/TryRuby/

3. play and learn ☺

Some differences to other languages

- no var declarations ; vars not typed

```
s = 3 ;  s = "three"        # perfectly valid
```



- statements separated by newlines (mostly)
  or ';' if on the same line

- statement can split across lines if unambiguous

```
total = x +
          y          # OK
total = x
        + y          # something different
```

- indentation insignificant

Variable naming conventions and rules

- ClassNames use UpperCamelCase
  ```
  class FriendFinder … end
  ```
- methods & variables use snake_case
  ```
  def learn_conventions … end
  def faculty_member? … end
  def charge_credit_card! … end
  some_variable
  ```
  --- no declarations!
- CONSTANTS: `TEST_MODE = true`
- `$global_vars`
- `@instance_var`
- `@@class_var`

Symbols

`:symbol` (note the ':')

- play an important role in Ruby
- like immutable string whose value is itself but not a string type
- not a reference to an object like CONSTANT

```
favorite_framework = :rails
:rails.to_s() == "rails"
"rails".to_sym() == :rails
:rails == "rails"  # => false
```

- can also write `symbol:` which is the same symbol

Some types

- vars have no types, but objects do!
- boolean: 2 values:
  - false, nil
  - true, anything_else including ""
- string:
  - "string"
  - 'string'
  - %Q{like double quotes}
  - %q{like single quotes}
  - double quotes can include expressions that get evaluated

```
@name="John"
 "name is @name"  # gets evaluated to "name is John"
```

Variables, arrays and hashes

- There are no declarations!
  - local variables must be assigned before use
  - instance & class variables ==`nil` until assigned
- OK: `x = 3; x = 'foo'`
- Wrong: `Integer x=3`
- Array: `x = [1,'two',:three]`
  `x[1] == 'two' ; x.length==3`
- Hash: collection of key-value pairs

```
h = {'a'=>1, :b=>[2, 3]}
```

or

```
h = {'a':1, b:[2,3]}

h[:b][0] == 2
h.keys == ['a', :b]
```

Control flow

- if <cond> (or unless <cond>)
  stmts
  [elsif <condi>
  stmts]
  . . .
  [else
  stmts]
  end

- case <expr>
  when <comp>
  stmts
  when <compi>
  stmts
  else
  stmts
  end

- while <cond> (or until)
  stmts
  end

- do
  stmts
  while <cond> (or until)

- for i in 0..9 do
  stmts
  end

- 1.upto(10) do |i|
  stmts
  end

- 10.times do stmts; end

Methods

```
def foo(x,y)
  return [x,y+1]
end

def foo(x,y=0)  # y is optional, 0 if omitted
  [x,y+1]       # last exp returned as result
end

def foo(x,y=0) ; [x,y+1] ; end
```

- every expression has a value: e.g., x=5 # 5
  every method returns value of last expression evaluated in method

Calling methods

- Call with `a,b = foo(x,y) # a is target object`
  or `a,b = foo(x) # when optional arg used`
  or `a,b = foo x # ()'s optional`
- Everything (except fixnums) is pass-by-reference
- Careful with spaces!!!
  f(45) # OK
  f 45 # OK
  f(3+2)+1 ! ≡ f (3+2)+1
- underneath: sends to object:

```
1 + 2                    1.send(:+, 2)
my_array[4]              my_array.send(:[], 4)
my_array[3] = "foo"      my_array.send(:[]=, 3,"foo")
if (x == 3)  ....        if (x.send(:==, 3)) ...
my_func(z)               self.send(:my_func, z)
```

More method call examples

```
y = [1,2]
y = y + ["foo",:bar] # => [1,2,"foo",:bar]
y << 5               # => [1,2,"foo",:bar,5]
y << [6,7]           # => [1,2,"foo",:bar,5,[6,7]]
```

- "<<" *destructively modifies* its receiver, "+" does not
  - destructive methods often have names ending in "!"
  - but most are nondestructive, returning a new copy
- Remember! These are nearly all *instance methods* of Array—*not* language operators!
- So `5+3`, `"a"+"b"`, and `[a,b]+[b,c]` are all *different* methods named '+'
  - `Numeric#+`, `String#+`, and `Array#+`, to be specific

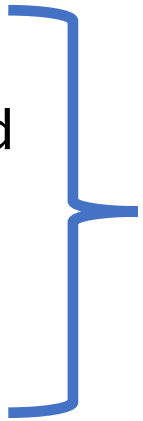
Defining a class

```
class Teen
    # default constructor
    def initialize( name, age )
        @name = name
        @age = age
    end

    # setter methods
    def name=( new_name ); @name=new_name; end
    def age=( new_age ); @age=new_age; end

    # getter methods
    def name?; @name; end
    def age?; @age; end
end


joe = Teen.new( "Joe", 16 )
joe.name= "Joey"
joe.age?          #  => 16
```

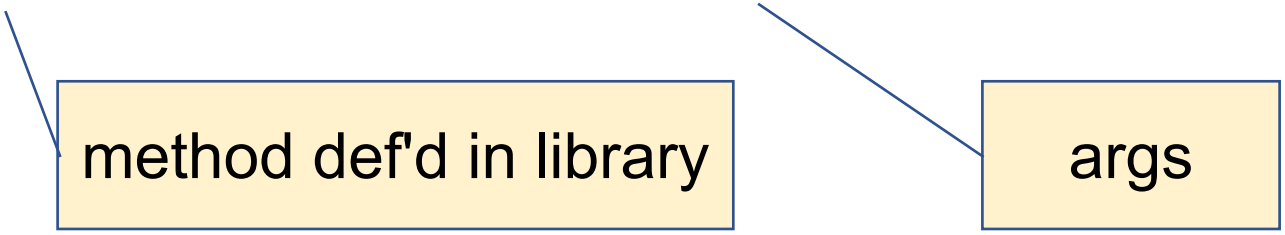


Only way to access instant vars from outside the class

attr_accessor helper

- writing setter and getter methods can be tedious
- generate them automatically with `attr_accessor`:

```
attr_accessor   :name :age
```



- also:
  - `attr_reader`
  - `attr_writer`

Ruby: differentiating principles

1. Everything is an object. No exceptions.
2. Every operation is a method call on some object (or more precisely, the specification of a method + optional args are sent to a specific object.)

```
a.b       # call method b on obj a
5.+(3)    # 5 is obj; + is method; 3 is arg
```

- it is the receiving objs responsibility to deal with the args, regardless of their types
- there is no typecasting (in most cases) and no operator overloading
- if receiver cannot handle the call, it automatically passes it to its superclass

Ruby: differentiating principles (cont.)

3. Supports reflection: the ability to ask objects about themselves.

```
3.class      # => Fixnum

[1, "a", :b].length    # => 3
```

4. All programming is meta-programming: classes & methods can be added or changed at any time, even by the program to which they belong.

   e.g., `attr_accessor` method described earlier

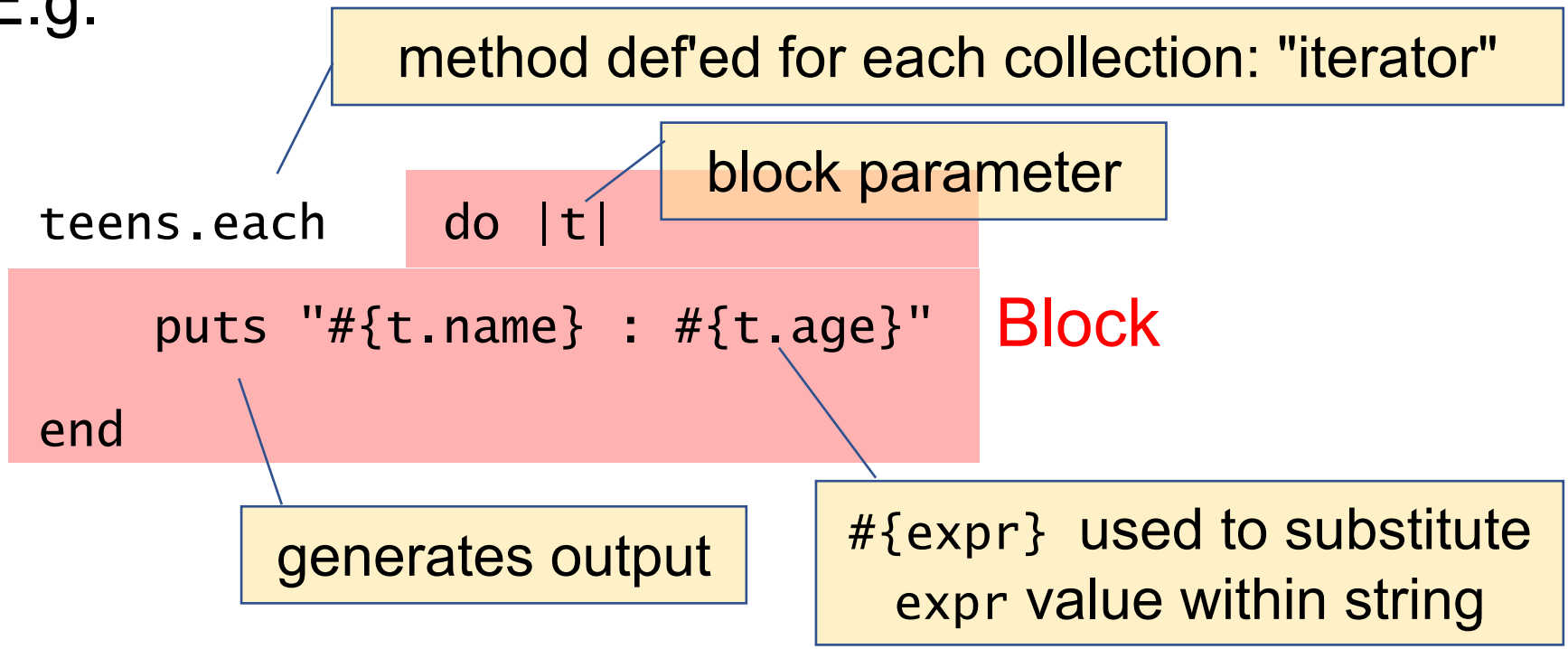
Ruby: differentiating principles (cont.)

e.g., adding a method to an existing class

```
class Integer
    def fib
        if self.zero?
            0
        elsif self == 1
            1
        else
            (self-1).fib + (self-2).fib
        end
    end
end
```

Blocks

- method with no name, which can be called with args (like lambda expressions in Lisp)
  E.g.



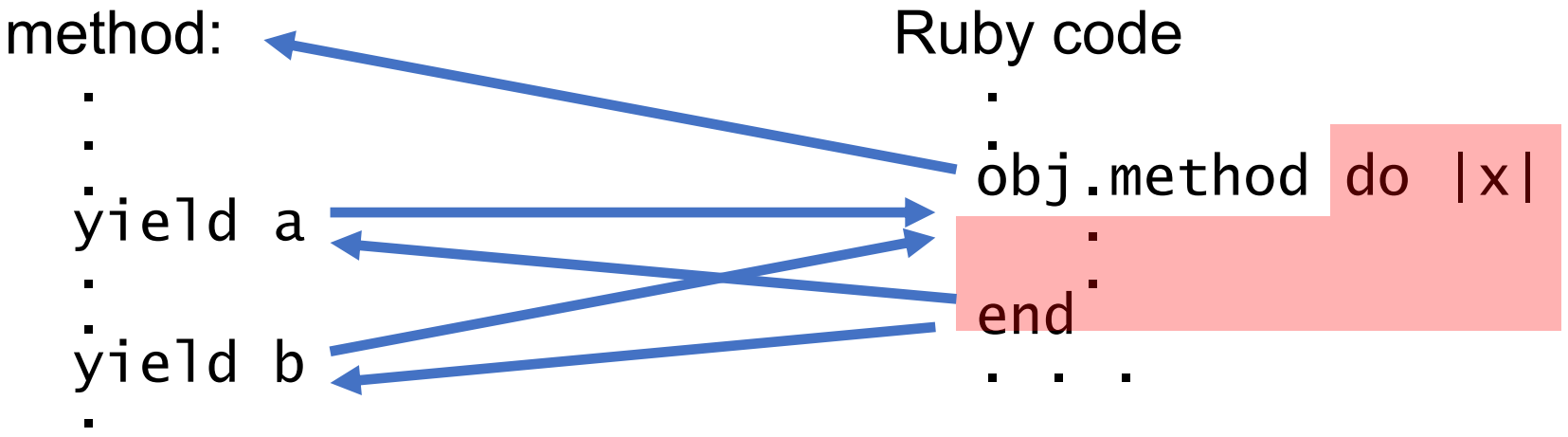
  here block is passed as an arg to method each
- '{','}' used for "do","end" if it fits on 1 line

Blocks (cont.)

- one block can be passed as arg to any method
- the block is invoked whenever the method executes

```
yield  <args>
```

E.g.



- Block returns value of last expression executed
- If method doesn't execute 'yield', block not executed

Blocks: example

```
def sequence( n, m, c )
    # generate n values: m*i+c
    i=0
    while( i<n )
        yield m*i+c
        i += 1
    end
end


sequence( 3, 5, 1 ) {|y| puts y}
     # => 1  6  11
```

- careful with `return` in block: does not apply to block instead use `next`

More block examples

- implement loops, but don't think of them that way

```
["apple","banana","cherry"].each do |string|
  puts string
end

for i in (1..10) do
  puts i
end

1.upto 10 do |num|
  puts num
end

3.times {  print "Rah, " }
```

Example: Web-page generation

- You can

```
def make_page( contents )
  page = ""
  page << make_header
  page << contents
  page << make_footer
end
contents = make_contents
make_page contents
```

- More Ruby-like

```
def make_page
  page = ""
  page << make_header
  page << yield
  page << make_footer
end
make_page { make_contents }
```

Iterators

- methods that invoke yield
- typically used to operate on collections (e.g., arrays, hashes, ranges, etc.)
- "each": method on a collection that takes a single argument: a block
- Used in many places; e.g.,

```
File.open( filename ) do |f|
  f.each { |line| print line }
end
```

Many operations on collections

- `c.map <block>`
  applies block to each element of c
  returns array of block-returned values
  e.g.,
  ```
  (1..3).map {|x| x*x } # => [1, 4, 9]
  ```
- `c.select <block>` or `c.reject <block>`
  subset of c for which block returns true/false
- `c.unique`
- `c.sort <block>`
  c sorted according to sorting criteria defined by blk

These functions can be applied to any object that supports "each" method, whether collection or not.

What does this due?

```
words = IO.read( "filename" ).
        split( /\w+/ ).
        select { |s| s.length==5 }.
        map { |s| s.downcase }.
        uniq.
        sort
```