# Ruby: variables & objects

- all vars are references to objects
- every obj has an id

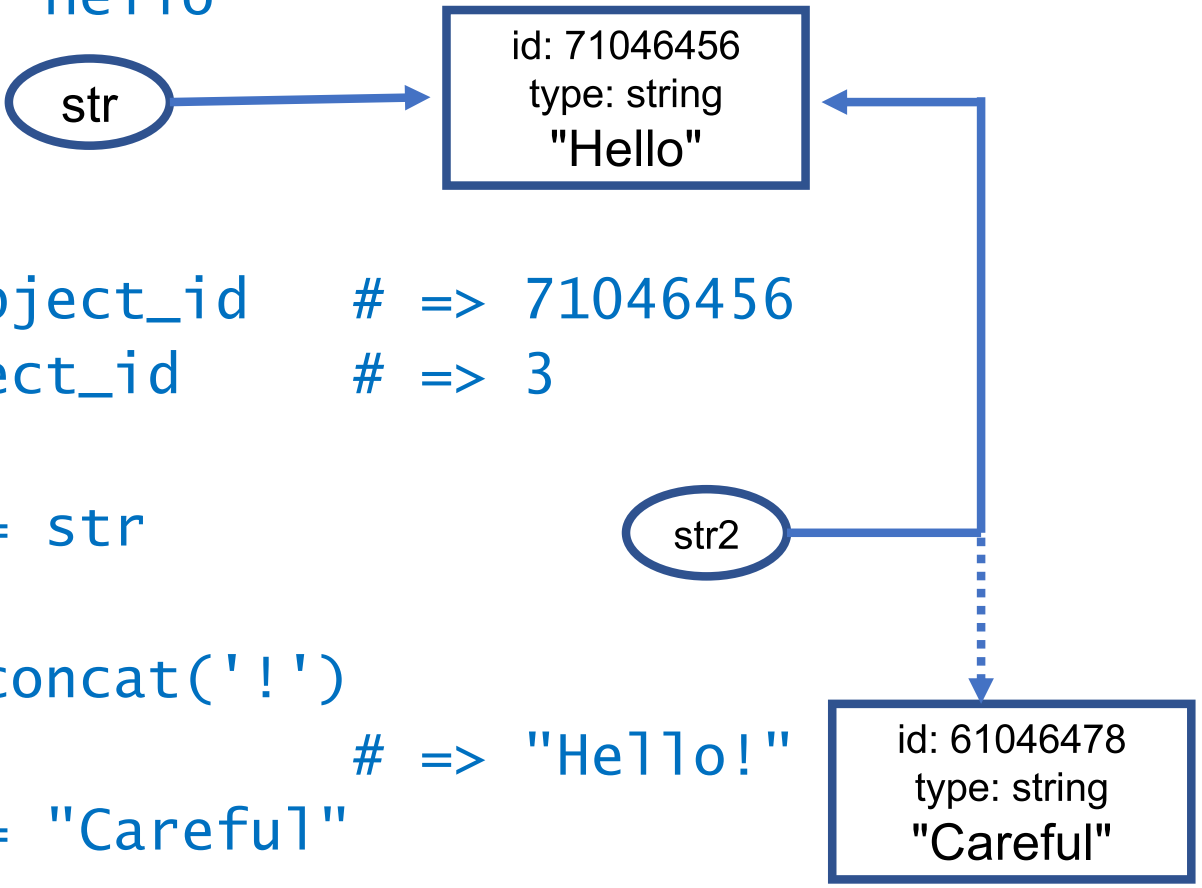
```
str = "Hello"



str.object_id   # => 71046456
3.object_id     # => 3

str2 = str

str2.concat('!')
str             # => "Hello!"
str2 = "Careful"
```



# Recall: Defining a class

```
class Teen < Person   # superclass is Person
   @@no_teens = 0     # class var
   # default constructor
   def initialize( name, age )
      @name = name
      @age = age
      @@no_teens++
   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

## Recall: attr_accessor helper

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

```
attr_accessor   :name :age
```

method def'd in library

args

- 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.

method def'ed for each collection: "iterator"

block parameter

```
teens.each    do |t|
    puts "#{t.name} : #{t.age}"
end
```

Block

generates output

`#{expr}` used to substitute `expr` value within string

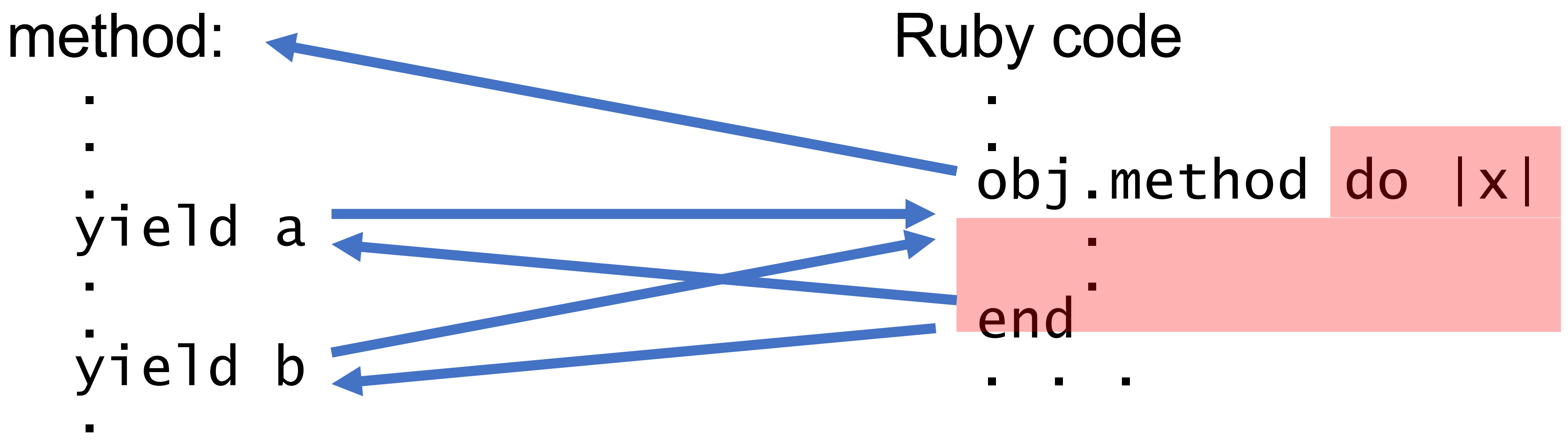
  here block is passed as an arg to method each
- '{','}' used for "do","end" if it fits on 1 line

# Blocks (cont.)

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

```
      yield  <args>
```

E.g.



```
method:
  .
  .
  .
  yield a
  .
  .
  yield b
  .
```

Ruby code

```
.
.
obj.method do |x|
   .
   .
end
. . .
```

- 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, " }
```

# Iterating with an index: unRuby-like

Iterators let objects manage their own traversal

- range traversals:
```
(1..10).each do |x| … end
(1..10).each { |x| … }
1.upto(10) do |x| … end
```

- array traversals:
```
my_array.each do |elt| … end
my_array.each_with_index do |elt,index| …end
```

- Hash traversals:
```
hsh.each_key do |key| … end
hsh.each_pair do |key, val| … end
```

- simple iteration with no index:
```
10.times { … }   # iterator of arity zero
```

# 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.

## Remember..

- `a.b` means: call method `b` on object `a`
- not `b` is an instance variable of `a`
- not `a` is some data structure that has `b` as a member

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

# Selftest... (tricky?)

Which line of code produces the same result as

```
arg = ["cool", "classy", "class"]
res = []
for i in ( 0 .. arg.length-1 ) do
   res << arg[i].capitalize
end
```

- `arg.each { |s| s.capitalize }`
- `res = arg.each { |s| s.capitalize }`
- `res = arg.map { |s| s.capitalize }`
- The above code won't run due to syntax errors

# Hashes and poetry mode

- you may omit parens around function args
- you may omit hash braces when last arg to fct is a hash

```
link_to("Edit",{:controller=>"teens", :action=>'edit'})
link_to("Edit", :controller=>"teens", :action=>'edit')
link_to "Edit", :controller=>"teens", :action=>'edit'
link_to "Edit", controller: "teens", action: 'edit'
```

All good and equivalent!

- Given: `def foo( arg, hash1, hash2 ) … end`

  Which is not a legal call to `foo()`?

```
foo a, {x:1, y:2}, x:3
foo a, x:1, y:2, x:3
foo( a, {x:1, y:2}, {x:3})
foo a, {x:1, y:2}, {x:3}
```

# Modules and Mix-ins

- Modules: collection of methods that aren't a class
- But you can "*mix its methods into*" a class

```
class A
    include Enumerable
    …
end
```

➔ if you implement method `each`, it provides
`all?, any?, collect, find, include?, inject, map, partition, sort, etc.`

Similarly, if class incudes `Comparable` and implement `<=>` then it provides: `< <= => > == between?` for free, as well as `sort` without needing a comparator block

## "Duck Typing"

If it responds to the same methods as a duck…
it might as well be a duck.