Iterators and blocks

Using iterators and blocks

Iterate with each or use a for loop?

Creating iterators

Iterators and blocks

Some methods are *iterators*. An iterator that is implemented by the Array class is each. each iterates over the elements of the array. Example:

```
>> x = [10,20,30]
=> [10, 20, 30]

>> x.each { puts "element" }
element
element
element
=> [10, 20, 30]
```

The construct { puts "element" } is a *block*. Array#each invokes the block once for each of the elements of the array.

Because there are three values in x, the block is invoked three times and "element" is printed three times.

Speculate: What does (1..50).each { putc ?x } do?

Iterators and blocks, continued

Iterators can pass one or more values to a block as arguments. Array#each passes each array element in turn.

A block can access arguments by naming them with a parameter list, a comma-separated sequence of identifiers enclosed in vertical bars.

We might print the values in an array like this:

```
>> [10, "twenty", 30].each { |e| printf("element: %s\n", e) }
element: 10
element: twenty
element: 30
```

A note about the format %s: In C, the value of the corresponding parameter must be a pointer to a zero-terminated sequence of char values. Ruby is more flexible—%s causes to_s to be invoked on the corresponding value. The result of to_s is used.

Another possibility for the format is %p, which causes inspect to be invoked. However, the second line above would be element: "twenty".

Iterators and blocks, continued

For reference:

```
[10, "twenty", 30].each { |e| printf("element: %s\n", e) }
```

Problem: Using a block, compute the sum of the numbers in an array containing values of *any* type. (Use elem.is_a? Numeric to decide whether elem is a number of some sort.)

Examples:

```
>> sum = 0

>> [10, "twenty", 30].each { ??? }
>> sum
=> 40

>> sum = 0

>> (1..100).each { ??? }

>> sum
=> 5050
```

Sidebar: Iterate with each or use a for loop?

You may recall that the for loop requires the result of the "in" expression to have an each method. Thus, we always have a choice between a for loop,

```
for name in "x".methods do
   puts name if name.include? "!"
end
```

and iteration with each,

```
"x".methods.each {|name| puts name if name.include? "!" }
```

Which is better?

Iterators and blocks, continued

Array#each is typically used to create side effects of interest, like printing values or changing variables but in many cases it is the value returned by an iterator that is of principle interest.

See if you can describe what each of the following iterators is doing.

```
>> [10, "twenty", 30].collect { |v| v * 2 }
=> [20, "twentytwenty", 60]

>> [[1,2], "a", [3], "four"].select { |v| v.size == 1 }
=> ["a", [3]]

>> ["burger", "fries", "shake"].sort { |a,b| a[-1] <=> b[-1] }
=> ["shake", "burger", "fries"]

>> [10, 20, 30].inject(0) { |sum, i| sum + i }
=> 60

>> [10,20,30].inject([ ]) { |thusFar, element| thusFar << element << "---" }
=> [10, "---", 20, "---", 30, "---"]
```


Iterators and blocks, continued

We have yet to study inheritance in Ruby but we can query the ancestors of a class like this:

```
>> Array.ancestors
=> [Array, Enumerable, Object, Kernel]
```

Because an instance of Array is an Enumerable, we can apply iterators in Enumerable to arrays:

```
>> [2, 4, 5].any? { |n| n % 2 == 0 }
=> true

>> [2, 4, 5].all? { |n| n % 2 == 0 }
=> false

>> [1,10,17,25].detect { |n| n % 5 == 0 }
=> 10

>> ["apple", "banana", "grape"].max { |a,b| v = "aeiou";
                                          a.count(v) <=> b.count(v) }
=> "banana"
```


Iterators and blocks, continued

Many classes have iterators. Here are some examples:

```
>> 3.times { |i| puts i }
0
1
2
=> 3

>> "abc".each_byte { |b| puts b }
97
98
99

>> (1..50).inject(1) { |product, i| product  * i }
=>30414093201713378043612608166064768844377641568960512000000000000
```

To print each line in the file x.txt, we might do this:

```
IO.foreach("x.txt") { |line| puts line }
```

A quick way to find the iterators for a class is to search for "block" in the documentation.

Blocks and iterators, continued

As you'd expect, blocks can be nested. Here is a program that reads lines from standard input, assumes the lines consist of integers separated by spaces, and averages the values.

```
total = n = 0
STDIN.readlines().each {
   |line|
   line.split(" ").each {
      |word|
      total += word.to_i
      n += 1
      }
   }

printf("Total = %d, n = %d, Average = %g\n", total, n, total / n.to_f) if n != 0
```

```
% cat nums.dat
5 10 0 50

 200
1 2 3 4 5 6 7 8 9 10
% ruby sumnums.rb < nums.dat
Total = 320, n = 15, Average = 21.3333
```

Notes:

- STDIN represents "standard input". It is an instance of IO.
- STDIN.readlines reads standard input to EOF and returns an array of the lines read.
- The printf format specifier %g indicates to format the value as a floating point number and select the better of fixed point or exponential form based on the value.

Some details on blocks

An alternative to enclosing a block in braces is to use do/end:

```
a.each do
      |element|
      printf("element: %s\n", element)
   end
```

do/end has lower precedence than braces but that only becomes an issue if the iterator is supplied an argument that is not enclosed in parentheses. (Good practice: enclose iterator argument(s) in parentheses, as shown in these slides.)

Note that do, {, or a backslash (to indicate continuation) must appear on the same line as the iterator invocation. The following will produce an error

```
a.each
   do                      # "LocalJumpError: no block given"
      |element|
      printf("element: %s\n", element)
   end
```


Some details on blocks, continued

Blocks raise issues with the scope of variables. If a variable is created in a block, the scope of the variable is limited to the block:

```
>> x
NameError: undefined local variable or method `x' for main:Object

>> [1].each { x = 10 }     => [1]

>> x
NameError: undefined local variable or method `x' for main:Object
```

If a variable already exists, a reference in a block is resolved to that existing variable.

```
>> x = "test"              => "test"

>> [1].each { x = 10 }     => [1]

>> x                       => 10
```

Sometimes you want that, sometimes you don't. It's said that this behavior may change with Ruby 2.0.

Creating iterators with yield

In Ruby, an iterator is "a method that can invoke a block".

The yield expression invokes the block associated with the current method invocation.

Here is a simple iterator that yields two values, a 3 and a 7:

```
def simple()
   puts "simple: Starting up..."
   yield 3

   puts "simple: More computing..."
   yield 7

   puts "simple: Out of values..."
   "simple result"
end
```

```
Usage:

>> simple() { |x| printf("\tx = %d\n", x) }
simple: Starting up...
        x = 3
simple: More computing...
        x = 7
simple: Out of values...
=> "simple result"
```

The iterator (simple) prints a line of output, then calls the block with the value 3. The iterator prints another line and calls the block with 7. It prints one more line and then returns, producing "simple result" as the value of simple() { |x| printf("\tx = %d\n", x) }.

Notice how the flow of control alternates between the iterator and the block.

yield, continued

Problem: Write an iterator from_to(f, t, by) that yields the integers from f through t in steps of by, which defaults to 1.

```
>> from_to(1,10) { |i| puts i }
1
2
...
10
=> nil

>> from_to(0,100,25) { |i| puts i }
0
25
50
75
100
=> nil
```


yield, continued

If a block is to receive multiple arguments, just specify them as a comma-separated list for yield.

Here's an iterator that produces consecutive pairs of elements from an array:

```
def elem_pairs(a)
   for i in 0..(a.length-2)
      yield a[i], a[i+1]
   end
end
```

Usage:

```
>> elem_pairs([3,1,5,9]) { |x,y| printf("x = %s, y = %s\n", x, y) }
x = 3, y = 1
x = 1, y = 5
x = 5, y = 9
```

Speculate: What will be the result with yield [a[i], a[i+1]]? (Extra brackets.)

yield, continued

Recall that Array#select produces the elements for which the block returns true:

```
>> [[1,2], "a", [3], "four"].select { |v| v.size == 1 }
=> ["a", [3]]
```

Speculate: How is the code in select accessing the result of the block?

yield, continued

The last expression in a block becomes the value of the yield that invoked the block.

Here's how we might implement a function-like version of select:

```
def select(enumerable)
   result = [ ]
   enumerable.each do
      |element|
      if yield element then
         result << element
      end
   end
   return result
end
```

Usage:

```
>> select([[1,2], "a", [3], "four"]) { |v| v.size == 1 }
=> ["a", [3]]
```

Note the we pass the array as an argument instead of invoking the object's select method.

yield, continued

Problem: Implement in Ruby an analog for ML's foldr.

```
>> foldr([10,20,30], 0) { |e, thus_far| e + thus_far }
=> 60

>> foldr([10,20,30], 0) { |e, thus_far| 1 + thus_far }
=> 3

>> foldr([5, 1, 7, 2], 0) { |e, max| e > max ? e : max }
=> 7
```

Here's a weakness in the instructor's implementation:

```
>> foldr(1..10, [ ]) { |e,thus_far| thus_far + [e] }
NoMethodError: undefined method `reverse_each' for 1..10:Range
```

What can we learn from it?

A Batch of Odds and Ends

Constants

Symbols

The Hash class

Constants

A rule in Ruby is that if an identifier begins with a capital letter, it represents a constant.

Ruby allows a constant to be changed but a warning is generated:

```
>> A = 1
=> 1

>> A = 2; A
(irb): warning: already initialized constant A
=> 2
```

Modifying an object referenced by a constant does *not* produce a warning:

```
>> L = [10,20]
=> [10, 20]

>> L << 30; L
=> [10, 20, 30]

>> L = 1
(irb): warning: already initialized constant L
```

Constants, continued

You may have noticed that the names of all the standard classes are capitalized. That's not simply a convention; Ruby requires class names to be capitalized.

```
>> class b
>> end
SyntaxError: compile error
(irb): class/module name must be CONSTANT
```

If a method is given a name that begins with a capital letter, it can't be found:

```
>> def M; 10 end
=> nil
>> M
NameError: uninitialized constant M
```


Constants, continued

There are a number of predefined constants. Here are a few:

ARGV
: An array holding the command line arguments, like the argument to main in a Java program.

FALSE, TRUE, NIL
: Synonyms for false, true, and nil.

STDIN, STDOUT
: Instances of IO representing standard input and standard output (the keyboard and screen, by default).

Symbols

An identifier preceded by a colon creates a Symbol. A symbol is much like a string but a given identifier always produces the same symbol:

```
>> s = :length          => :length
>> s.object_id          => 42498
>> :length.object_id    => 42498
```

In contrast, two identical string literals produce two different String objects:

```
>> "length".object_id   => 23100890
>> "length".object_id   => 23096170
```

If you're familiar with Java's String.intern method, note that Ruby's String#to_sym is roughly equivalent:

```
>> "length".to_sym.object_id    => 42498
```

For the time being, it's sufficient to simply know that *:identifier* creates a Symbol.

The Hash class

Ruby's Hash class is similar to Hashtable and Map in Java. It can be thought of as an array that can be subscripted with values of any type, not just integers.

The expression { } (empty curly braces) creates a Hash:

```
>> numbers = { }        => { }
>> numbers.class        => Hash
```

Subscripting a hash with a "key" and assigning a value to it stores that key/value pair in the hash:

```
>> numbers["one"] = 1   => 1
>> numbers["two"] = 2   => 2
>> numbers              => {"two"=>2, "one"=>1}
>> numbers.size         => 2
```

Hash, continued

At hand:

```
>> numbers          => {"two"=>2, "one"=>1}
```

To fetch the value associated with a key, simply subscript the hash with the key. If the key is not found, nil is produced.

```
>> numbers["two"]          => 2

>> numbers["three"]        => nil
```

The value associated with a key can be changed via assignment. A key/value pair can be removed with Hash#delete.

```
>> numbers["two"] = "1 + 1"      => "1 + 1"

>> numbers.delete("one")         => 1      # The associated value, if any, is
                                           #  returned.

>> numbers                       => {"two"=>"1 + 1"}
```

Speculate: What is the net result of numbers["two"] = nil?

Hash, continued

There are no restrictions on the types that can be used for keys and values.

```
>> h = { }                      => { }

>> h[1000] = [1,2]              => [1, 2]

>> h[true] = { }                => { }

>> h[[1,2,3]] = [4]             => [4]

>> h                            => {true=>{ }, [1, 2, 3]=>[4], 1000=>[1, 2]}

>> h[h[1000] + [3]] << 40       => [4, 40]

>> h[!h[10]]["x"] = "ten"       => "ten"

>> h                            => {true=>{"x"=>"ten"}, [1, 2, 3]=>[4, 40], 1000=>[1, 2]}
```


Hash, continued

It was said earlier that if a key is not found, nil is returned. That was a simplification. In fact, the *default value* of the hash is returned if the key is not found.

The default value of a hash defaults to nil but an arbitrary default value can be specified when creating a hash with new:

```
>> h = Hash.new("Go Fish!")     => { }     # Example from ruby-doc.org

>> h["x"] = [1,2]               => [1, 2]

>> h["x"]                       => [1, 2]

>> h["y"]                       => "Go Fish!"

>> h.default                    => "Go Fish!"
```

It is not discussed here but there is also a form of Hash#new that uses a block to produce default values.

Hash example: tally.rb

Here is a program that reads lines from standard input and tallies the number of occurrences of each word. The final counts are dumped with inspect.

```
counts = Hash.new(0)     # Use default of zero so that ' += 1' works.

STDIN.readlines.each {
   |line|
   line.split(" ").each {
      |word|
      counts[word] += 1
      }
   }
puts counts.inspect              # Equivalent: p counts
```

Usage:

```
% ruby tally.rb
to be or
not to be
^D
{"or"=>1, "be"=>2, "to"=>2, "not"=>1}
```

tally.rb, continued

The output of puts counts.inspect is not very user-friendly:

```
{"or"=>1, "be"=>2, "to"=>2, "not"=>1}
```

Hash#sort produces a list of key/value lists ordered by the keys, in ascending order:

```
>> counts.sort
[["be", 2], ["not", 1], ["or", 1], ["to", 2]]
```

Problem: Produce nicely labeled output, like this:

```
Word            Count
be                  2
not                 1
or                  1
to                  2
```


tally.rb, continued

At hand:

```
>> counts.sort
[["be", 2], ["not", 1], ["or", 1], ["to", 2]]
```

Solution:

```
([["Word","Count"]] + counts.sort).each {
    |k,v| printf("%-10s\t%5s\n", k, v)          # %-10s left-justifies in a field of width 10
    }
```

As a shortcut for easy alignment, the column headers are put at the start of the list. Then, we use %5s instead of %5d to format the counts and accommodate "Count". (Recall that this works because %s causes to_s to be invoked on the value.)

Is the shortcut a "programming technique" or a hack?

tally.rb, continued

Hash#sort's default behavior of ordering by keys can be overridden by supplying a block.

The block is repeatedly invoked with two arguments: a pair of list elements.

```
>> counts.sort { |a,b| puts "a = #{a.inspect}, b = #{b.inspect}"; 1}
a = ["or", 1], b = ["to", 2]
a = ["to", 2], b = ["not", 1]
a = ["be", 2], b = ["to", 2]
a = ["be", 2], b = ["or", 1]
```

The block is to return -1, 0, or 1 depending on whether a is considered to be less than, equal to, or greater than b.

Here's a block that sorts by descending count: (the second element of the two-element lists)

```
>> counts.sort { |a,b| b[1] <=> a[1] }
[["to", 2], ["be", 2], ["or", 1], ["not", 1]]
```

How could we put ties on the count in ascending order by the words? Example:

```
[["be", 2], ["to", 2], ["not", 1], ["or", 1]]
```


Hash initialization

It is tedious to initialize a hash with a series of assignments:

```
numbers = { }
numbers["one"] = 1
numbers["two"] = 2
...
```

Ruby provides a shortcut:

```
>> numbers = { "one", 1, "two", 2, "three", 3 }
=> {"three"=>3, "two"=>2, "one"=>1}
```

There's a more verbose variant, too:

```
>> numbers = { "one" => 1, "two" => 2, "three" => 3 }
=> {"three"=>3, "two"=>2, "one"=>1}
```

One more option: (but note that both keys and values are strings)

```
>> Hash[ * %w/a 1 b 2 c 3 d 4 e 5/ ]
=> {"a"=>"1", "b"=>"2", "c"=>"3", "d"=>"4", "e"=>"5"}
```