==========

Iterators

supply a `for` loop with the members of a set
abstraction/generalization of the "`from A to B by C`" sorts of stuff you see built-in in older languages

pioneered by Clu:
`for i in <iterator> do ... end`
built-in iterators for `from_to`, `from_to_by`, etc.

wonderful for iterating over arbitrary user-defined sets
very good for abstraction; for loop doesn't have to know whether set is a linked list, hash table, dense array, etc.

may be ***true iterators*** (as in Clu, C#, Icon, Python, Ruby) or interface-based approximation ("***iterator objects***," as in Euclid, Java, and C++)

in Python:

```
def uptoby(lo, hi, step):
   while True:
      if (step > 0 and lo > hi) \
         or (step < 0 and lo < hi): return
      yield lo
      lo += step   # ignore overflow

for i in uptoby(1, 20, 2):
   print (i)
```

Iterator objects (Euclid, C++, Java)

Standard interface for abstraction to drive for loops.
Supported in Euclid and Java with special loop syntax, and in C++ through clever use of standard constructor and operator overload mechanisms.

In Java:

```
List<foo> myList = ...;

for (foo o : myList) {
   // use object o
}
```

requires that the to-be-iterated class (here, `List`) implements the `Iterable` interface, which exports a method

```
public Iterator<T> iterator()
```

where Iterator is an interface exporting methods

```
public boolean hasNext()
```

and

```
public T next()
```

The `for` loop is syntactic sugar for

```
for (Iterator<foo> i = myList.iterator(); i.hasNext();) {
   foo o = i.next();
   // use object o
}
```

C++ version looks like

```
list<foo> my_list;
...
for (list<foo>::const_iterator i = my_list.begin();
      i != my_list.end(); i++) {
   // make use of *i or i->field_name
}
```

Don't have to have an equivalent of the Iterator interface (it's just a convention), because C++ individually type-checks every use of a generic (template).

Note the different conceptual model:
Java has a special `for` loop syntax that uses methods of a special class
C++ standard library defines iterators as "pointer-like" objects with increment operations to drive ordinary `for` loops

All the standard library collection/container classes support iterators, in both languages.

True iterators (Clu, Icon, C#, Python)

iterator itself looks like a procedure, except it can include "`yield`" statements that produce intermediate values.
when the iterator returns, the loop terminates

C# for loop resembles that of Java:

```
foreach (foo o in myList) {
   // use object o
}
```

This is syntactic sugar for

```
for (IEnumerator<foo> i =
      myList.GetEnumerator(); i.MoveNext()) {
   foo o = i.Current;
   // use object o
}
```

`Current` is an ***accessor*** -- a special method supporting field-like access:

```
public object Current {
   get {
      return ...;
   }
   put {
      ... = value;
   }
}
```

In contrast to Java, you don't need to hand-create the `hasNext()` [`MoveNext()`] and `next()` [`Current`] methods.The compiler does this automatically when your class implements the `IEnumerable` interface and has an iterator -- a method containing "`yield return`" statements and "returning" an `IEnumerator`:

```
class List : IEnumerable {
   ...
   public IEnumerator GetEnumerator() {
      node n = head;
      while (n != null) {
         yield return n.content;
         n = n.next;
      }              // NB: no return statement
   }
}
```

If you want to be able to have multiple iteration orders, your class can have multiple methods that each return an `IEnumerator`.
Then you can say, e.g.

```
foreach (object o in myTree.InPreOrder) { ...

foreach (object o in myTree.InPostOrder) { ...
```

detail:
`IEnumerator` implements `MoveNext` and `Current` (also `Reset`)
`IEnumerable` implements `GetEnumerator`, which returns an `IEnumerator`

Loop body as lambda (Smalltalk, Scheme, ML, Ruby, ...)

OCaml:

```
open Printf;;
let show n = printf "%d\n" n;;

let upto lo hi =
  fun f -> let rec helper i =
           if i > hi then ()
           else (f i ; helper (i + 1)) in
  helper lo;;

upto 1 10 show;;      =>
   1
   2
   3
   4
   5
   6
   7
   8
   9
   10
   - : unit = ()
```

Ruby:

```
sum = 0                                  => 0
[ 1, 2, 3 ].each { |i| sum += i }        => [1, 2, 3]  # array itself
sum                                      => 6
```

Here the (parameterized) brace-enclosed block is passed to the `each` method as a parameter.

There's also more conventional-looking syntax:

```
sum = 0
for i in [1, 2, 3] do     # 'do' is optional
   sum += i
end
sum
```

The `for` loop is syntactic sugar for a call to `each`.

Here's a more object-oriented alternative:

```
sum = 0
1.upto 3 {|i| sum += i}
sum
```

or instead of using braces:

```
sum = 0
i.upto 3 do |i| sum += i end
sum
```

You can write your own iterators using '`yield`'.

```
class Array
def find
   for i in 0...size
      value = self[i]
      return value if yield(value)
   end
   return nil
end
end
...
[1, 3, 5, 7, 9].find {|v| v*v > 30 }     => 7
```

Think of `yield` as invoking the block that was juxtaposed ("associated") with the call to the iterator.

(FWIW, the array class already has a `find` method in Ruby, but we can redefine it, and it probably looks like this anyway.)

Blocks can also be turned into first-class closures, with unlimited extent:

```
def nTimes(aThing)
   # Ruby, like most scripting languages, is dynamically typed
   return proc { |n| aThing * n }
end
```

In recent Ruby, –> is a synonym for `proc`

```
p1 = nTimes(3)
p2 = nTimes("foo")
p1.call(4)             => 12
p2.call(4)             => "foofoofoofoo"
```

This lets us build higher-level functions. Here's reduction for arrays:

```
class Array
   def reduce(n)
      each { |value| n = yield n, value }     # that's self.each
         # yield invokes (just once) the block associated
         # with the call to reduce. Note the lack of parens:
         # "yield (n, value)" would pass a single tuple.
      n  # return value
   end
   def sum
      reduce(0) { |a, v| a + v }
   end
   def product
      reduce(1) { |a, v| a * v }
   end
 end

   [2, 4, 6].sum          => 12
   [2, 4, 6].product      => 48
```

All in all Ruby is pretty cool. Check it out.
(I do wish it let you associate more than one block with a call.)

Implementation of true iterators (section 9.5.3-CS)

coroutines or threads
overkill

single-stack
used in Clu
works, but would confuse a standard debugger, and not compatible with some conventions for argument passing

implicit iterator object
kinda cool; used in C# and Python

block as lambda expression (Ruby, functional languages)