1 Introduction

You will explore some aspects of and write some simple classes and methods in the μSmalltalk language.

2 Description

• (Adapted from Exercise 12 of Chapter 10 from Programming Languages: Build, Prove, and Compare (p. 804).)
  Implement a class Rand having the following protocol.
  
  - Class method fromSeed: creates a new random-number generator in which the argument is the seed (an integer).
  
  - Class method new creates a new random-number generator with a default (but unspecified) seed.
  
  - Instance method next answers the next random number and updates the seed. By default, the sequence of random numbers should be generated as follows:

    \[ s_{i+1} = 9 \times s_i + 5 \mod 1024 \]

    where \( s_0 \) is the initial seed.

  - (extra) Class method fromSeed:withAPRNG: creates a new random-generator in which the first argument is the seed (an integer) and the second argument is an applicative pseudorandom number generator (a block of one argument), which determines the sequence of random numbers to be generated.

• (Adapted from Exercise 15 of Chapter 10 from Programming Languages: Build, Prove, and Compare (p. 805).)
  Add to class Array a class method from: aCollection that makes an array out of the elements of another collection.
  (Note: Simply give the implementation of the class method; there should be no need to otherwise change the implementation of class Array.)

The important methods from the class protocol of Array are:

<table>
<thead>
<tr>
<th>new: anInteger</th>
<th>Create and answer an array of size anInteger in which each element is nil.</th>
</tr>
</thead>
</table>

and the important methods from instance protocols of the Collection hierarchy are:

<table>
<thead>
<tr>
<th>at:put: key value</th>
<th>Modify the receiver by associating key with value. May add a new value of replace an existing value. Answer the receiver. (From the instance protocol for KeyedCollection.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>do: aBlock</td>
<td>For each element ( x ) of the collection, evaluate (value aBlock x). (From the instance protocol for Collection.)</td>
</tr>
<tr>
<td>size</td>
<td>Answer how many element the receiver contains. (From the instance protocol for Collection.)</td>
</tr>
</tbody>
</table>

\[ \text{addAll: aCollection} \]
\[ \text{add: anObject X not supported by Array} \]
1 Introduction

You will explore some aspects of and write some simple classes and methods in the \(\mu\)Smalltalk language.

2 Description

- (Adapted from Exercise 12 of Chapter 10 from *Programming Languages: Build, Prove, and Compare* (p. 804).)

  Implement a class `Rand` having the following protocol.

  - Class method `fromSeed`: creates a new random-number generator in which the argument is the seed (an integer).

  - Class method `new` creates a new random-number generator with a default (but unspecified) seed.

  - Instance method `next` answers the next random number and updates the seed. By default, the sequence of random numbers should be generated as follows:

    \[ s_{i+1} = 9 \times s_i + 5 \mod 1024 \]

    where \(s_0\) is the initial seed.

  - (extra) Class method `fromSeed:withAPRNG`: creates a new random-generator in which the first argument is the seed (an integer) and the second argument is an applicative pseudorandom number generator (a block of one argument), which determines the sequence of random numbers to be generated.

- (Adapted from Exercise 15 of Chapter 10 from *Programming Languages: Build, Prove, and Compare* (p. 805).)

  Add to class `Array` a class method `from: aCollection` that makes an array out of the elements of another collection.

  (Note: Simply give the implementation of the class method; there should be no need to otherwise change the implementation of class `Array`.)

The important methods from the class protocol of `Array` are:

| new: anInteger | Create and answer an array of size `anInteger` in which each element is `nil`. |

and the important methods from instance protocols of the `Collection` hierarchy are:

<table>
<thead>
<tr>
<th>at:put: key value</th>
<th>Modify the receiver by associating <code>key</code> with <code>value</code>. May add a new value of replace an existing value. Answer the receiver. (From the instance protocol for <code>KeyedCollection</code>.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>do: aBlock</td>
<td>For each element <code>x</code> of the collection, evaluate <code>(value aBlock x)</code>. (From the instance protocol for <code>Collection</code>.)</td>
</tr>
<tr>
<td>size</td>
<td>Answer how many element the receiver contains. (From the instance protocol for <code>Collection</code>.)</td>
</tr>
</tbody>
</table>

```smalltalk
addAll: aCollection from Collection
add: anObject < abstract
```
• Recall the implementations of the instance methods size and addAll: of the class Collection:

(method size () (locals cnt)
  (set cnt 0)
  (do: self (block (x) (set cnt (+ cnt 1))))
  cnt)
(method addAll: (aCollection)
  (do: aCollection (block (x) (add: self x)))
  self)

  – Implement the instance method size without using a block literal.
    (Hint: Although one typically uses a block literal to construct the argument of do:, do: simply requires that its argument is an object that “walks like a block and swims like a block and quacks like a block.”)
  – Implement the instance method addAll: aCollection without using a block literal.
  – Comment on the feasibility and/or utility of abolishing block literals from μSmalltalk, from the perspective of a language designer/implementer and from the perspective of a language user.

• (Adapted from Exercise 18.5 of Chapter 10 from Programming Languages: Build, Prove, and Compare (p. 806).)

Recall the implementation of the instance method detect:ifNone: of the class Collection:

(method detect:ifNone: (aBlock exnBlock) (locals answer searching)
  (set searching true)
  (do: self (block (x)
    (ifTrue: (and: searching {((value aBlock x)})
      {((set searching false)
        (set answer x)})}))
    (if searching exnBlock {answer})))

Implement the instance method detect:ifNone: so as to avoid the final if.

3 Requirements and Submission

You may use the reference interpreter (see Appendix A), but there may only be one active laptop in each group.

At the end of class, submit the group’s solutions as hard-copy; be sure to include the names of all group members in the submission.

A Interpreter

A reference μSmalltalk interpreter is available on the CS Department Linux systems (e.g., glados.cs.rit.edu and queeg.cs.rit.edu and ICLs 1 and 2) at:

/usr/local/pub/mtf/plc/bin/usmalltalk

Use the reference interpreter to check your code.
Solutions

- class Rand
  (class Rand Object
     (seed)
     (class-method new ()
       (withSeed: self 42))
     (class-method withSeed: (seed)
       (withSeed: (new super) seed))
     (method withSeed: (thatSeed)
       (set seed thatSeed) self)
     (method next ()
       (set seed (mod: (+ (* 9 seed) 5) 1024)))
  )

(new rand value will be a block (suspended code))

Note the use of (new super), rather than (new Rand), to create the new instance of the Rand class. This avoids an infinite loop, as the Rand class has a class-method new.

- class method Array.from: aCollection
  (class Array
     Array
     ()
     (class-method from: (aCollection) (locals a i)
       (set a (new: self (size aCollection)))
       (set i 0)
       (do: aCollection (block (x)
         (at:put: a i x)
         (set i (+ i 1))))
     )

[Image of an array notation]
Solutions

- class Rand

  (class Rand Object
    (seed)
    (class-method new ()
      (withSeed: self 42))
    (class-method withSeed: (seed)
      (withSeed: (new super) seed))
    (method withSeed: (thatSeed)
      (set seed thatSeed) self)
    (method next ()
      (set seed (mod: (+ (* 9 seed) 5) 1024)))
  )

  (class Rand Object
    (seed aprng)
    (class-method new ()
      (withSeed: self 42))
    (class-method withSeed: (seed)
      (withSeed::withAPRNG: self seed (block (x) (mod: (+ (* 9 x) 5) 1024)))
    )
    (class-method withSeed::withAPRNG: (seed aprng)
      (withSeed::withAPRNG: (new super) seed aprng))
    )
    (method withSeed::withAPRNG: (thatSeed thatAPRNG)
      (set seed thatSeed) (set aprng thatAPRNG) self)
    )
    (method next ()
      (set seed (value aprng seed)))
  )

Note the use of (new super), rather that (new Rand), to create the new instance of the Rand class. This avoids an infinite loop, as the Rand class has a class-method new.

- class method Array.from: aCollection

  (class Array Array
    ()
    (class-method from: (aCollection) (locals a i)
      (set a (new: self (size aCollection)))
      (set i 0)
      (do: aCollection [block (x)
        at:put: a i x]
        set i (+ i 1)])
  )
• living without block literals

- instance method Collection.size

(class CntBlock Block
  (cnt)
  (class-method new () (locals cnt)
    (withCnt: (new super) 0))
  (method withCnt: (thatCnt)
    (set cnt thatCnt) self)
  (method value (_) (set cnt (+ cnt 1)))
  (method get () cnt)
)
(class Collection Collection
()
 (method size () (locals cnt)
    (set cnt (new CntBlock))
    (do: self cnt)
    (get cnt))
)

- instance method Collection.addAll: aCollection

(class AddAllBlock Block
  (coll)
  (class-method withColl: (coll)
    (withColl: (new super) coll))
  (method withColl: (thatColl)
    (set coll thatColl) self)
  (method value (x) (add: coll x))
)
(class Collection Collection
()
 (method addAll: (aCollection)
   (do: aCollection (withColl: AddAllBlock self))
   self)
)

- For the language designer/implementer, the \( \mu \text{Smalltalk} \) semantics and implementation would be slightly simplified if block literals were abolished. It would simplify the semantics, by eliminating the CLOSURE primitive representation for blocks, the rule for evaluating a block literal, and the special-case rules for sending the message value to an object. (Indeed, the operational semantics of \( \mu \text{Smalltalk} \) as presented in (the current draft of) Programming Languages: Build, Prove, Compare does not admit the above implementations, since the operational semantics does not allow the value message to be answered by a user-defined method, though the interpreter does.) It would similarly simplify the implementation, by eliminating the parsing of block literals and various cases in the evaluator; however, the value message would remain a special case when checking the arity of method definitions and send expressions.

For the language user, programming in \( \mu \text{Smalltalk} \) would be awkward and cumbersome if block literals were abolished. The above implementations demonstrate a general pattern: for each eliminated block literal, introduce a new class with instance variables that correspond to the free variables of the block literal and initialize with the free variables of the block literal. Of course, there are some subtleties. Note that the last use of cnt in the original implementation of size is translated to a get cnt; after the execution of an eliminated block literal, the values of any instance variables set by the value method of the proxy object should be reflected back to the corresponding free variables via updates. The major difficulty with automating this elimination is that where and when a block object is executed is distinct from where and when a block object is created (cf. control-flow analysis in functional programming languages and points-to analysis is object-oriented programming languages); hence, it may be difficult or impossible to know where to insert such updates. (The implementations above “work” because we know
that the created block object is executed by the do: method and is not stored away to be executed later; note, however, that this required knowing something about the do: method in addition to the knowing the block literal to be eliminated.) The nuclear option would be to modify all uses and sets of an any variable captured by a block literal to go through a wrapper object that introduces a level of indirection.

In any case, manually building closures is awkward and cumbersome. Witness the addition of anonymous classes, inner classes, and lambdas to Java to simplify the definition and use of such one-off classes.

In addition to the issues described above, there are two additional features of Smalltalk that complicate the elimination of block literals. In the μSmalltalk and Smalltalk-80 languages, it is not clear that the use of the super receiver in a block literal can be (easily) handled. Similarly, in the Smalltalk-80 language, the use of a non-local return in a block literal can not be (easily) handled.

- **detect:ifNone:** without if

```
(method detect:ifNone: (aBlock exnBlock) (locals answer searching)
  (set searching true)
  (do: self (block (x)
    (ifTrue: (and: searching {((value aBlock x)})
      {{(set searching false)
        (set exnBlock {x})})}})
    (value exnBlock))
```


1 Introduction

In this programming assignment, you will implement a number of classes and methods in μSmalltalk in order to gain familiarity with the language and to practice object-oriented programming.

Download prog06.smt, prog06_tests.smt, and prog06_tests.soln.out (or copy from /usr/local/pub/mtf/plc/programming/ prog06-smalltalk on the CS Department Linux systems). The first is a template for your submission and also includes a number of supporting classes. The second is a test suite for the assignment and the third is reference solution’s output on the test suite.

2 Description

This assignment investigates writing μSmalltalk classes that represent immutable, space-efficient vectors, which we call “xvectors”. Complete the definitions of the abstract class XVector and its concrete sub-classes ArrayXVector, ConcatXVector, RepeatXVector, ReverseXVector, SwizzleXVector, and BlockXVector to provide the protocols specified in Figures 1, 2, and 3. (Note: These classes represent space-efficient vectors. Hence, they should not unnecessarily allocate new data. The trade-off is that the at: method on xvectors may not be O(1).)

See Requirements and Submissions for important restrictions.
XVector instance protocol

**display methods**

print

Print the receiver on standard output; an xvector is printed as <<, the list of elements separated by spaces, followed by >>.

debug

Print a representation of the xvector on standard output; the representation is constructed from the name of the receiver’s class, an open parenthesis, the arguments used to construct the receiver (separated by commas), and a close parenthesis; any xvector arguments used to construct the receiver are printed using debug; non-xvector arguments used to construct the receiver are printed using print. (subclass responsibility) (10pts)

Note: The initial basis of the ūSmalltalk interpreter includes global variables space, newline, semicolon, quote, left-paren, right-paren, left-square, right-square, left-curly, and right-curly, which are bound to objects of class Char that represent the space character, the new line character, the semicolon character “;”, the quote character “"”, the left parenthesis character “(”, the right parenthesis character “)”, the left square bracket character “[”, the right square bracket character “]”, the left curly brace character “{”, and the right curly brace character “}”). Such characters are useful for printing (send them the print message), but cannot be expressed using ūSmalltalk’s literal symbol notation.

**observer methods**

isEmpty

Answer whether the receiver contains any elements. (like the corresponding Collection method)

size

Answer how many elements the receiver contains. (like the corresponding Collection method) (subclass responsibility) (10pts)

at: anIndex

Answer the element at position anIndex, or report the error index-out-of-bounds if the position anIndex is out of bounds. A non-negative position counts forward from the start of the xvector (i.e., (at: xvector 0) returns the first element). A negative position counts backward from the end of the xvector (i.e., (at: xvector -1) returns the last element).

at:ifAbsent: anIndex exnBlock

Answer the element at position anIndex, or answer (value exnBlock) if the position anIndex is out of bounds. (see at: method comments) (10pts)

includes: anObject

Answer whether the receiver contains anObject; uses = to compare anObject to elements. (like the corresponding Collection method)

occurrencesOf: anObject

Answer how many of the receiver’s elements are equal to anObject; uses = to compare anObject to elements. (like the corresponding Collection method)

detect: aBlock

Answer the first element x in the receiver for which (value aBlock x) is true, or report the error no-object-detected if none. (like the corresponding Collection method)

detect:ifNone: aBlock exnBlock

Answer the first element x in the receiver for which (value aBlock x) is true, or answer (value exnBlock) if none.

sum

Answer the sum of the elements in the receiver; assumes all elements are members of the same Number subclass and answers an Integer if the receiver is empty. (5pts)

product

Answer the product of the elements in the receiver; assumes all elements are members of the same Number subclass and answers an Integer if the receiver is empty. (5pts)

min

Answer the minimum element in the receiver, or report the error min-of-empty if the receiver is empty; assumes all elements answer messages of the Magnitude instance protocol. (5pts)

max

Answer the maximum element in the receiver, or report the error max-of-empty if the receiver is empty; assumes all elements answer messages of the Magnitude instance protocol. (5pts)

Figure 1: XVector instance protocol
**XVector instance protocol**

<table>
<thead>
<tr>
<th><strong>display methods</strong></th>
<th><strong>Print the receiver on standard output; an xvector is printed as &lt;&lt;, the list of elements separated by spaces, followed by &gt;&gt;.</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>print</strong></td>
<td>Print a representation of the xvector on standard output; the representation is constructed from the name of the receiver’s class, an open parenthesis, the arguments used to construct the receiver (separated by commas), and a close parenthesis; any xvector arguments used to construct the receiver are printed using <strong>debug</strong>; non-xvector arguments used to construct the receiver are printed using <strong>print</strong>. (subclass responsibility) (10pts)</td>
</tr>
<tr>
<td><strong>debug</strong></td>
<td>Note: The initial basis of the μSmalltalk interpreter includes global variables space, newline, semicolon, quote, left-paren, right-paren, left-square, right-square, left-curly, and right-curly, which are bound to objects of class <strong>Char</strong> that represent the space character, the new line character, the semicolon character “;”, the quote character “&quot;”, the left parenthesis character “(”, the right parenthesis character “)””, the left square bracket character “[”, the right square bracket character “]”, the left curly brace character “{”, and the right curly brace character “}”. Such characters are useful for printing (send them the <strong>print</strong> message), but cannot be expressed using μSmalltalk’s literal symbol notation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>observer methods</strong></th>
<th><strong>Answer whether the receiver contains any elements. (like the corresponding Collection method)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>isEmpty</strong></td>
<td>Answer how many elements the receiver contains. (like the corresponding Collection method) (subclass responsibility) (10pts)</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>Answer the element at position <strong>anIndex</strong>, or report the error index-out-of-bounds if the position <strong>anIndex</strong> is out of bounds. A non-negative position counts forward from the start of the xvector (i.e., (at: xvector 0) answers the first element); a negative position counts backward from the end of the xvector (i.e., (at: xvector -1) answers the last element).</td>
</tr>
<tr>
<td><strong>at:</strong> <strong>anIndex</strong></td>
<td>Answer the element at position <strong>anIndex</strong>, or answer (value <strong>exmBlock</strong>) if the position <strong>anIndex</strong> is out of bounds. (see at: method comments) (10pts)</td>
</tr>
<tr>
<td><strong>at:ifAbsent:</strong> <strong>anIndex</strong> <strong>exmBlock</strong></td>
<td>Answer whether the receiver contains <strong>anObject</strong>: uses = to compare <strong>anObject</strong> to elements. (like the corresponding Collection method)</td>
</tr>
<tr>
<td><strong>includes:</strong> <strong>anObject</strong></td>
<td>Answer how many of the receiver’s elements are equal to <strong>anObject</strong>: uses = to compare <strong>anObject</strong> to elements. (like the corresponding Collection method)</td>
</tr>
<tr>
<td><strong>occurrencesOf:</strong> <strong>anObject</strong></td>
<td>Answer the first element <strong>x</strong> in the receiver for which (value <strong>aBlock</strong>) is true, or report the error no-object-detected if none. (like the corresponding Collection method)</td>
</tr>
<tr>
<td><strong>detect:</strong> <strong>aBlock</strong></td>
<td>Answer the first element <strong>x</strong> in the receiver for which (value <strong>aBlock</strong>) is true, or answer (value <strong>exmBlock</strong>) if none.</td>
</tr>
<tr>
<td><strong>detect:ifNone:</strong> <strong>aBlock</strong> <strong>exmBlock</strong></td>
<td>Answer the sum of the elements in the receiver; assumes all elements are members of the same <strong>Number</strong> subclass and answers an <strong>Integer</strong> if the receiver is empty. (5pts)</td>
</tr>
<tr>
<td><strong>sum</strong></td>
<td>Answer the product of the elements in the receiver; assumes all elements are members of the same <strong>Number</strong> subclass and answers an <strong>Integer</strong> if the receiver is empty. (5pts)</td>
</tr>
<tr>
<td><strong>product</strong></td>
<td>Answer the minimum element in the receiver, or report the error min-of-empty if the receiver is empty; assumes all elements answer messages of the Magnitude instance protocol. (5pts)</td>
</tr>
<tr>
<td><strong>min</strong></td>
<td>Answer the maximum element in the receiver, or report the error max-of-empty if the receiver is empty; assumes all elements answer messages of the Magnitude instance protocol. (5pts)</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: XVector instance protocol
For each element \( x \) in the receiver (in order of increasing position), evaluate
(value \( ab\text{Block} \cdot x \)). (like the corresponding Collection method) (10pts)

Evaluates \( ab\text{Block} \) once for each element in the receiver. The first argument
of the block is an element from the receiver; the second argument is the
result of the previous evaluation of the block, starting with \( \text{thisValue} \). Answer
the final value of the block. (like the corresponding Collection method)

### Comparison methods

**similar anObject**

Answers whether the receiver is similar to anObject; an xvector is not similar
to an object that is not an instance of XVector and two xvectors are similar if
they have the same size and elements of corresponding positions are similar.
(like the corresponding Collection method) (10pts)

**< anXVector**

Answers whether the receiver is less than anXVector; xvectors are compared
via lexicographic order; assumes all elements answer messages of the Magnitude
instance protocol. (like the corresponding Magnitude method) (10pts)

**> anXVector**

Answers whether the receiver is greater than anXVector. (see \(<\) method comments; like the corresponding Magnitude method)

**<= anXVector**

Answers whether the receiver is no greater than anXVector. (see \(<\) method comments; like the corresponding Magnitude method)

**>= anXVector**

Answers whether the receiver is no less than anXVector. (see \(<\) method comments; like the corresponding Magnitude method)

**min: anXVector**

Answer the lesser of the receiver and anXVector. (see \(<\) method comments; like the corresponding Magnitude method)

**max: anXVector**

Answer the greater of the receiver and anXVector. (see \(<\) method comments; like the corresponding Magnitude method)

### Producer methods

**+ anXVector**

Answer an xvector that represents the concatenation of the receiver and
anXVector.

If \( \text{anInteger} \) is non-negative, answer an xvector that rep-
resents \( \text{anInteger} \) concatenations of the receiver. If
\( \text{anInteger} \) is negative, report the error \text{negative-repeat}.
(There may be opportunities to override this method in a subclass; explain
your reasoning in a comment at the overriding method implementation.
Note: Remember that these classes represent space-efficient vectors. An
overriding implementation should not allocate more data than the abstract
superclass implementation and should make the answered xvector more
efficient for (some) operations than the xvector answered by the abstract
superclass implementation. (bonus 3pts))

**reverse**

Answer an xvector that represents the reversal of the receiver.
(There may be opportunities to override this method in a subclass; explain
your reasoning in a comment at the overriding method implementation.
(see * method comments) (bonus 3pts))

**fromIndex:toIndex: aStartIndex anEndIndex**

Answer an xvector that represents the elements of the receiver from posi-
tion \( \text{aStartIndex} \) to position \( \text{anEndIndex} \) (inclusive). If position \( \text{aStartIndex} \)
comes after position \( \text{anEndIndex} \) in the receiver, then the answered xvec-
tor has elements from the end of the receiver followed by elements
from the start of the receiver (i.e., the slice “wraps around”). If ei-
ther position \( \text{aStartIndex} \) or position \( \text{anEndIndex} \) are out of bounds,
then report the error report the error \text{index-out-of-bounds}. (10pts)
(There may be opportunities to override this method in a subclass; explain
your reasoning in a comment at the overriding method implementation. (see
* method comments) (bonus 3pts))

### Private methods (internal to XVector classes)

**elem: anIndex**

Answer the element at position \( \text{anIndex} \); assumes that the position \( \text{anIndex} \) is
positive and within bounds. (subclass responsibility) (10pts)

---

Figure 2: XVector instance protocol (continued)
**Iterator methods**

- **do: aBlock**
  
  For each element `x` in the receiver (in order of increasing position), evaluate `(value aBlock x)`. (like the corresponding `Collection` method) (10pts)

- **inject:into: thisValue binaryBlock**
  
  Evaluates `binaryBlock` once for each element in the receiver. The first argument of the block is an element from the receiver; the second argument is the result of the previous evaluation of the block, starting with `thisValue`. Answer the final value of the block. (like the corresponding `Collection` method)

**Comparison methods**

- **similar anObject**
  
  Answers whether the receiver is similar to `anObject`; an xvector is not similar to an object that is not an instance of `XVector` and two xvectors are similar if they have the same size and elements of corresponding positions are similar. (like the corresponding `Collection` method) (10pts)

- **< anXVector**
  
  Answers whether the receiver is less than `anXVector`; xvectors are compared via lexicographic order; assumes all elements answer messages of the `Magnitude` instance protocol. (like the corresponding `Magnitude` method) (10pts)

- **> anXVector**
  
  Answers whether the receiver is greater than `anXVector`. (see `<` method comments; like the corresponding `Magnitude` method)

- **<= anXVector**
  
  Answers whether the receiver is no greater than `anXVector`. (see `<` method comments; like the corresponding `Magnitude` method)

- **>= anXVector**
  
  Answers whether the receiver is no less than `anXVector`. (see `<` method comments; like the corresponding `Magnitude` method)

- **min: anXVector**
  
  Answer the lesser of the receiver and `anXVector`. (see `<` method comments; like the corresponding `Magnitude` method)

- **max: anXVector**
  
  Answer the greater of the receiver and `anXVector`. (see `<` method comments; like the corresponding `Magnitude` method)

**Producer methods**

- **+ anXVector**
  
  Answer an xvector that represents the concatenation of the receiver and `anXVector`.

- *** anInteger**
  
  If `anInteger` is non-negative, answer an xvector that represents `anInteger` concatenations of the receiver. If `anInteger` is negative, report the error `negative-repeat`. (There may be opportunities to override this method in a subclass; explain your reasoning in a comment at the overriding method implementation. Note: Remember that these classes represent space-efficient vectors. An overriding implementation should not allocate more data than the abstract superclass implementation and should make the answered xvector more efficient for (some) operations than the xvector answered by the abstract superclass implementation. (bonus 3pts))

- **reverse**
  
  Answer an xvector that represents the reversal of the receiver. (There may be opportunities to override this method in a subclass; explain your reasoning in a comment at the overriding method implementation. (see `*` method comments) (bonus 3pts))

- **fromIndex:toIndex: aStartIndex anEndIndex**
  
  Answer an xvector that represents the elements of the receiver from position `aStartIndex` to position `anEndIndex` (inclusive). If position `aStartIndex` comes after position `anEndIndex` in the receiver, then the answered xvector has elements from the end of the receiver followed by elements from the start of the receiver (i.e., the slice "wraps around"). If either position `aStartIndex` or position `anEndIndex` are out of bounds, then report the error report the error `index-out-of-bounds`. (10pts) (There may be opportunities to override this method in a subclass; explain your reasoning in a comment at the overriding method implementation. (see `*` method comments) (bonus 3pts))

**Private methods (internal to XVector classes)**

- **elem: anIndex**
  
  Answer the element at position `anIndex`; assumes that the position `anIndex` is positive and within bounds. (subclass responsibility) (10pts)

Figure 2: XVector instance protocol (continued)
ArrayXVector class protocol

withArr: anArray

Create and answer an xvector that represents the elements of anArray; since an xvector is immutable, the elements of anArray must be copied at the time of construction.

ConcatXVector class protocol

withXV1:withXV2: anXVector1 anXVector2

Create and answer an xvector that represents the concatenation of anXVector1 and anXVector2. (2pts)

RepeatXVector class protocol

withXV:withN: anXVector anInteger

If anInteger is non-negative, create and answer an xvector that represents anInteger concatenations of anXVector. If anInteger is negative, report the error negative-repeat-count. (2pts)

ReverseXVector class protocol

withXV: anXVector

Create and answer an xvector that represents the reversal of anXVector. (2pts)

SwizzleXVector class protocol

withXV1:withXV2: anXVector1 anXVector2

Create and answer an xvector that represents the swizzle of anXVector1 and anXVector2: the first element of the swizzle is the first element of anXVector1, the second element of the swizzle is the first element of anXVector2, the third element of the swizzle is the second element of anXVector1, the fourth element of the swizzle is the second element of anXVector2, and so on. If anXVector1 and anXVector2 are of unequal lengths, then the swizzle concludes with the excess elements from the rest of the longer one. (2pts)

BlockXVector class protocol

withN:withBlock: anInteger aBlock

If anInteger is non-negative, create and answer an xvector that is of size anInteger and the element at position i is obtained by (value aBlock i). aBlock may assume that it will only be evaluated with indices i such that 0 ≤ i < anInteger. If anInteger is negative, report the error negative-block-size. (2pts)

Figure 3: XVector sub-classes class protocols
### ArrayXVector class protocol

| withArr: anArray | Create and answer an xvector that represents the elements of anArray; since an xvector is immutable, the elements of anArray must be copied at the time of construction. |

### ConcatXVector class protocol

| withXV1:withXV2: anXVector1 anXVector2 | Create and answer an xvector that represents the concatenation of anXVector1 and anXVector2. (2pts) |

### RepeatXVector class protocol

| withXV:withN: anXVector anInteger | If anInteger is non-negative, create and answer an xvector that represents anInteger concatenations of anXVector. If anInteger is negative, report the error negative-repeat-count. (2pts) |

### ReverseXVector class protocol

| withXV: anXVector | Create and answer an xvector that represents the reversal of anXVector. (2pts) |

### SwizzleXVector class protocol

| withXV1:withXV2: anXVector1 anXVector2 | Create and answer an xvector that represents the swizzle of anXVector1 and anXVector2. the first element of the swizzle is the first element of anXVector1, the second element of the swizzle is the first element of anXVector2, the third element of the swizzle is the second element of anXVector1, the fourth element of the swizzle is the second element of anXVector2, and so on. If anXVector1 and anXVector2 are of unequal lengths, then the swizzle concludes with the excess elements from the rest of the longer one. (2pts) |

### BlockXVector class protocol

| withN:withBlock: anInteger aBlock | If anInteger is non-negative, create and answer an xvector that is of size anInteger and the element at position i is obtained by (value aBlock i). aBlock may assume that it will only be evaluated with indices i such that 0 ≤ i < anInteger. If anInteger is negative, report the error negative-block-size. (2pts) |

Figure 3: XVector sub-classes class protocols
3 Requirements and Submission

Your submission must be a valid μSmalltalk program. In particular, it must pass the following test:

```bash
$ cat prog06.smt | /usr/local/pub/mtf/plc/bin/usmalltalk -q > /dev/null
```

without any error messages. If your submission produces error messages (e.g., syntax errors), then your submission will not be tested and will result in zero credit for the assignment.

Submit `prog06.smt` to the Programming 06 Dropbox on MyCourses by the due date.

4 Hints

- Remember to double-check `ifTrue:ifFalse:` message sends; the receiver must be a Boolean object and the two arguments must be (nullary) blocks.

- Remember to double-check `whileTrue:` message sends; the receiver must be a (nullary) block (that answers a Boolean object) and the argument must be a (nullary) block.
  - You may (and should) add instance variables to the concrete sub-classes.
  - You may define additional (private) helper methods.
  - You may define additional classes.

- Note that the `do:` method is a concrete method of the `XVector` superclass. This is different from the `Collection` hierarchy, where the `do:` method is an abstract method of the `Collection` superclass.

- Note that the `sum` and `product` methods assume that all elements of the receiver are elements of the same `Number` subclass. Thus, these methods should work on `xvector`s of `SmallInteger`, `Fraction`, and `Float`. An inelegant solution uses `isKindOf:` to dynamically determine the specific `Number` subclass. An elegant solution uses the `coerce:` method of the `Number` protocol.

- Note that the `min` and `max` methods assume that all elements of the receiver are elements of the same `Magnitude` subclass. Thus, these methods should work on `xvector`s of `SmallInteger`, `Fraction`, and `Float`. Also, note that the methods report an error if the receiver is empty. So, the meaningful computation of the minimum or maximum element will only proceed when the receiver is non-empty.

- Note that the argument of the `similar:` method can be an arbitrary object. It would be appropriate to use the `isKindOf:` method to determine if the argument is an `xvector` and the proceed to comparing elements. Be sure to use `similar:` to compare elements, not `=`. Note that `xvector`s of different sizes are never similar.

- Note that the `<` compares the receiver and argument `xvector`s via lexicographic comparison. Lexicographic order is “dictionary order”. In particular, `< -6 -5 -4 >>` is less than `< 1 >>` and `< 1 >>` is less than `< 6 5 4 >>`

```smalltalk
-> (ifTrue: ifFalse: true 1 -1)
SmallInt doesn't understand value
-> (ifTrue: ifFalse: true block (1) 1) 8-13
```
3 Requirements and Submission

Your submission must be a valid μSmalltalk program. In particular, it must pass the following test:

\$ cat prog06.smt | /usr/local/pub/stf/plc/bin/usmalltalk -q > /dev/null

without any error messages. If your submission produces error messages (e.g., syntax errors), then your submission will not be tested and will result in zero credit for the assignment.

Submit prog06.smt to the Programming 06 Dropbox on MyCourses by the due date.

4 Hints

- Remember to double-check **ifTrue:** if**False:** message sends; the receiver must be a Boolean object and the two arguments must be (nullary) blocks.

- Remember to double-check **whileTrue:** message sends; the receiver must be a (nullary) block (that answers a Boolean object) and the argument must be a (nullary) block.

- You may (and should) add instance variables to the concrete sub-classes.

- You may define additional (private) helper methods.

- You may define additional classes.

- Note that the **do:** method is a concrete method of the **XVector** superclass. This is different from the **Collection** hierarchy, where the **do:** method is an abstract method of the **Collection** superclass.

- Note that the **sum** and **product** methods assume that all elements of the receiver are elements of the same **Number** subclass. Thus, these methods should work on xvector's of **SmallInteger**, **Fraction**, and **Float**. An inelegant solution uses **isKindOf:** to dynamically determine the specific **Number** subclass. An elegant solution uses the **coerce:** method of the **Number** protocol.

- Note that the **min** and **max** methods assume that all elements of the receiver are elements of the same **Magnitude** subclass. Thus, these methods should work on xvector's of **SmallInteger**, **Fraction**, and **Float**. Also, note that the methods report an error if the receiver is empty. So, the meaningful computation of the minimum or maximum element will only proceed when the receiver is non-empty.

- Note that the argument of the **similar:** method can be an arbitrary object. It would be appropriate to use the **isKindOf:** method to determine if the argument is an xvector and the proceed to comparing elements. Be sure to use **similar:** to compare elements, not **=**. Note that xvectors of different sizes are never similar.

- Note that the **<** compares the receiver and argument xvectors via lexicographic comparison. Lexicographic order is “dictionary order”. In particular, **< -6 -5 -4 >>** is less than **< 1 >>** and **< 1 >>** is less than **< 6 5 4 >>**.

\[ \rightarrow (\text{ifTrue: ifFalse: } b \ -1 \ 1) \]

SmallInt doesn't understand value

\[ \rightarrow (\text{ifTrue: ifFalse: } \text{true} \ (\text{block} (\ 1) \ -1) \ 3 \ 1 \ 3 ) \]

\[ -1 \]
Note that the `fromIndex:toIndex:` method of `XVector` and the constructors for the sub-classes of `XVector` should not explicitly construct a data structure proportional in size to the created vector. This is perhaps best exemplified by the following transcript:

```plaintext
-> (val xv withArr: ArrayXVector '(1 2 3 4 5))
<< 1 2 3 4 5 >>
-> (val rxv1 (withXV:withN: RepeatXVector xv 9))
<< 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 >>
-> (size rxv1)
46
-> (at: rxv1 0)
1
-> (at: rxv1 4)
5
-> (at: rxv1 5)
1
-> (at: rxv1 9)
5
-> (val rxv2 nil)
nil
-> (begin (set rxv2 (withXV:withN: RepeatXVector xv 9999)) nil)
nil
-> (size rxv2)
49995
-> (at: rxv2 0)
1
-> (at: rxv2 4)
5
-> (at: rxv2 5)
1
-> (at: rxv2 9)
5
-> (at: rxv2 4321)
2.
```

How can you efficiently compute the size of `rxv2` without explicitly constructing a 49995 element data structure, knowing that `rxv2` was constructed from `xv` and a repeat count of 9999? How can you efficiently determine the element at index 4321 of `rxv2`?
Note that the `fromIndex:toIndex:` method of `ArrayVector` and the constructors for the sub-classes of `XVector` should not explicitly construct a data structure proportional in size to the created vector. This is perhaps best exemplified by the following transcript:

```plaintext
-> (val xv (withArr: ArrayVector '(1 2 3 4 5)))
  (1 2 3 4 5)
-> (val rxv1 (withXV:withNil: (RepeatXVector xv 9)))
  (1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5)
-> (size rxv1)
  45
  (at: rxv1 0)
  1
  (at: rxv1 4)
  5
  (at: rxv1 5)
  1
  (at: rxv1 9)
  5
-> (val rxv2 nil)
  nil
-> (begin (set rxv2 (withXV:withNil: (RepeatXVector xv 9999))) nil)
  nil
-> (size rxv2)
  49995
  (at: rxv2 0)
  1
  (at: rxv2 4)
  5
  (at: rxv2 5)
  1
  (at: rxv2 9)
  5
  (at: rxv2 4321)
  2
```

How can you efficiently compute the size of `rxv2` without explicitly constructing a 49995 element data structure, knowing that `rxv2` was constructed from `xv` and a repeat count of 9999? How can you efficiently determine the element at index 4321 of `rxv2`?
A Interpreter

A reference μSmalltalk interpreter is available on the CS Department Linux systems (e.g., glados.cs.rit.edu and queeg.cs.rit.edu and ICLs 1 and 2) at:

```
/usr/local/pub/mtf/plc/bin/usmalltalk
```

Use the reference interpreter to check your code.

Source code for the interpreter is available on the CS Department file system at:

```
/usr/local/pub/mtf/plc/src/bare/usmalltalk
```

B Test Suite

Executing

```
$ cat prog06.smt prog06_tests.smt | /usr/local/pub/mtf/plc/bin/usmalltalk -qq > prog06_tests.out
```

will run the interpreter on the contents of the files prog06.smt and prog06_tests.smt (all tests) without prompts or responses printed and save the output to the file prog06_tests.out; then executing

```
$ diff prog06_tests.soln.out prog06_tests.out
```

will compare the files prog06_tests.soln.out and prog06_tests.out and print any differences.

Similarly, executing

```
$ cat prog06.smt util.smt test-A-at:ifAbsent:.smt | /usr/local/pub/mtf/plc/bin/usmalltalk -qq > test-A-at:ifAbsent:.out
```

will run the interpreter on the contents of the files prog06.smt, util.smt, and test-A-at:ifAbsent:.smt (an individual test file) without prompts or responses printed and save the output to the file test-A-at:ifAbsent:.smt.out; then executing

```
$ diff test-A-at:ifAbsent:.soln.out test-A-at:ifAbsent:.out
```

will compare the files test-A-at:ifAbsent:.soln.out and test-A-at:ifAbsent:.soln.out and print any differences.

Note: Due to the interdependencies between the classes and methods of the assignment, it is not easy to test individual pieces of functionality in isolation. You will probably find the test suite most helpful after you have a mostly completed assignment, when you can use the test suite to discover and diagnose any minor errors or missing corner cases. You will probably not find it helpful to use the test suite as the guiding force for completing the assignment.

The best suggestion is to use the system interactively to debug one method at a time.

Note: Most of the .soln.out files are simply All 6 tests passed., but a small number include lines like (debug cxv01) --> ConcatXVector(ArrayXVector(( )),ArrayXVector(( ))), which demonstrates the behavior of the debug method.