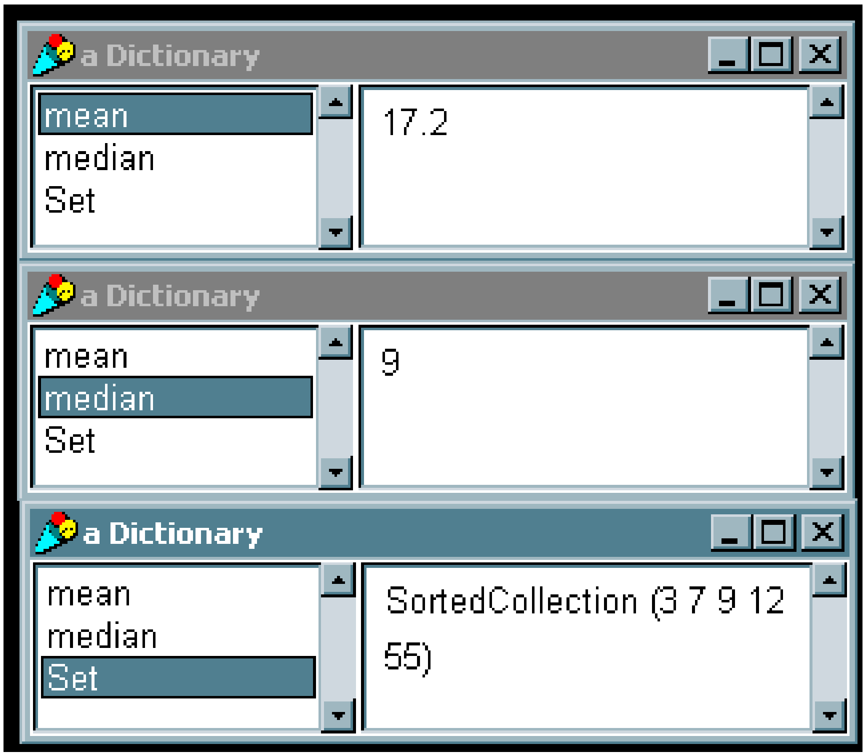
Lecture 15: System & Magnitude Classes

- **Overview**
  - Shared Object Protocols
    - Messages implemented for all objects
  - 3 Classes
    - Magnitude Classes
      - Numbers & characters
    - Collection Classes
      - Lists, Arrays, and Dictionaries
    - Streams
      - Text, Files, and Sockets
- **Shared Object Protocols**
  - 3 messages that can be applied to an object relating to its class
    - `class` finds out what class an object belongs to
      - `#(this is an array) class` ← Array
      - Similar to class are:
        - `isKindOf: aClass` returns true if aClass is a parent class of the reciever
          - `#(this is an array) isKindOf: Collection` ← true
        - `ismemberOf: aClass` returns true if the receiver is an instance of aClass.
          - `#(this is an array) isMemberOf: Collection` ← false
    - `isSequenceable` returns Boolean value depending on whether the receiver is created from a subclass of `SequenceableCollecetion`
      - #(this is an array) isSequenceable ← true.
      - `(Bag with: 'this' with: 'is' with: 'a' with: 'bag') isSequenceable` ← false
      - NOTE: class SequenceableCollection is called class IndexedCollection in smalltalk express, and isSequenceable is not available
    - `respondsToArithmetic: returns` Boolean
      - `respondsToArthithmetic` is implemented using the more general message, `respondsTo: aSymbol`, testing the symbols `#+, #-, #*`, and `#/`
  - `Comparing objects`
    - `==, ~~` CANNOT be overridden
    - `=, ~=` CAN be overridden
    - `isNil, notNil`
- Example: how to test and compare objects.
  - Suppose we want to write a method that takes a set, and creates a dictionary. The dictionary stores the sorted list of members, the median, and the mean.

```
compileStats: aSet
    |aDictionary sum setSize|
    aDictionary := Dictionary new.
    (aSet isKindOf: Set)
          ifFalse: [self notify: 'warning, argument is not a kind of
                 class Set'. ^nil].
    aSet class == SortedCollection
          ifTrue: [ aDictionary at: 'Set' put: aSet]
          ifFalse:
                 [ | aNewSet |
                 aNewSet := SortedCollection new.
                 aNewSet addAll: aSet.
                 aDictionary at: 'Set' put: aNewSet].
    (aDictionary at: 'Set') do:
          [:x | x respondsToArithmetic
                 ifFalse: [
                        self notify: 'Not numeric set'.
                        ^nil]].
    setSize := (aDictionary at: 'Set') size.
```

```
aDictionary at: 'median' put: ((aDictionary at: 'Set') at:
       ((setSize/2) rounded)).
sum :=  0.
(aDictionary at: 'Set') do: [ :x | sum := sum + x].
aDictionary at: 'mean' put: ((sum/ setSize) asFloat).
^aDictionary.
```

- Set(7, 12, 3, 9, 55) would result in the following dictionary

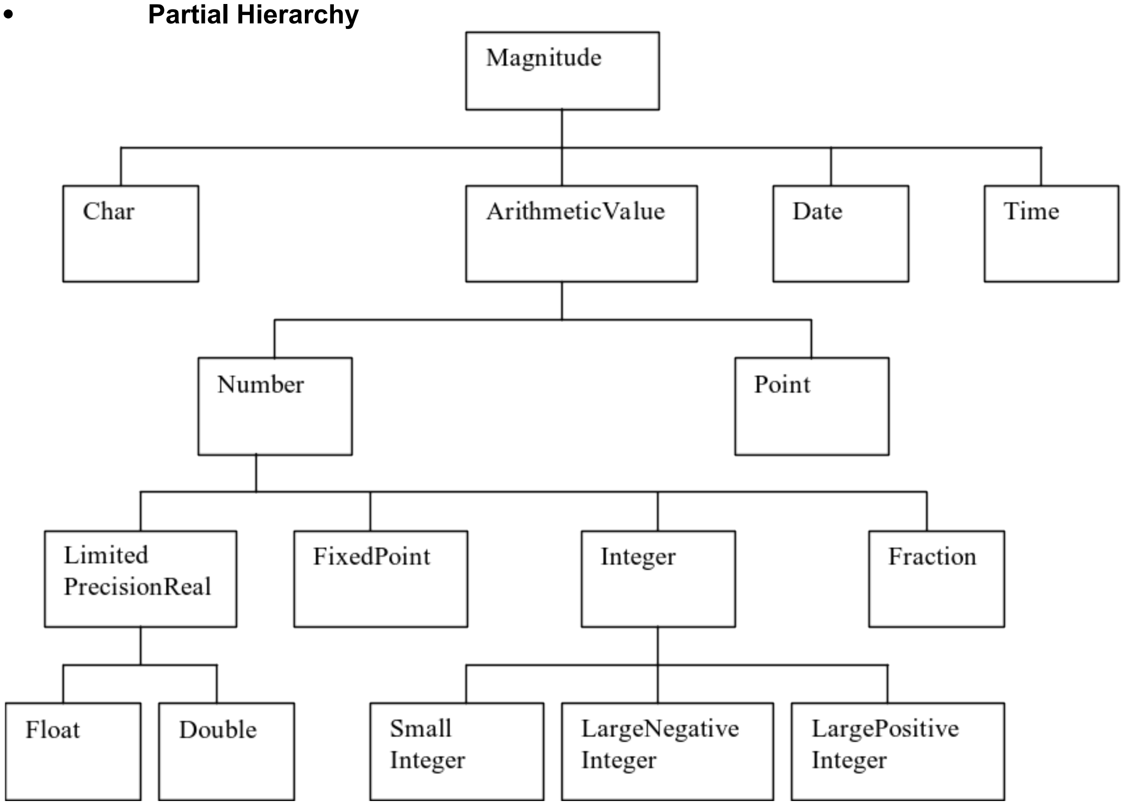


- **4 basic subclasses of the Magnitude class**
  - `Char`
    - Similar to `char` in C, basic class can be treated similarly to number
  - `ArithmeticValue`
    - Superclass for all numerical classes
  - `Date`
    - Very different from C style of date & time, comparable and human readable
  - `Time`
    - Very different from C style of date & time, comparable and human readable
- **Methods provided for comparison**
  - `aMagnitude between: oneMagnitude and: anotherMagnitude` (range comparison)
  - `aMagnitude max: anotherMagnitude` (max of the two magnitudes)
  - `aMagnitude min: anotherMagnitude` (min of two magnitudes)
  - `aMagnitude hash`
  - `<, <=, >, >=`
- **Example: More methods for complex numbers**

```
abs
       "Returns the absolute value of a complex number"
       ^(self realPart squared + self imaginaryPart squared)sqrt

< aComplex
```

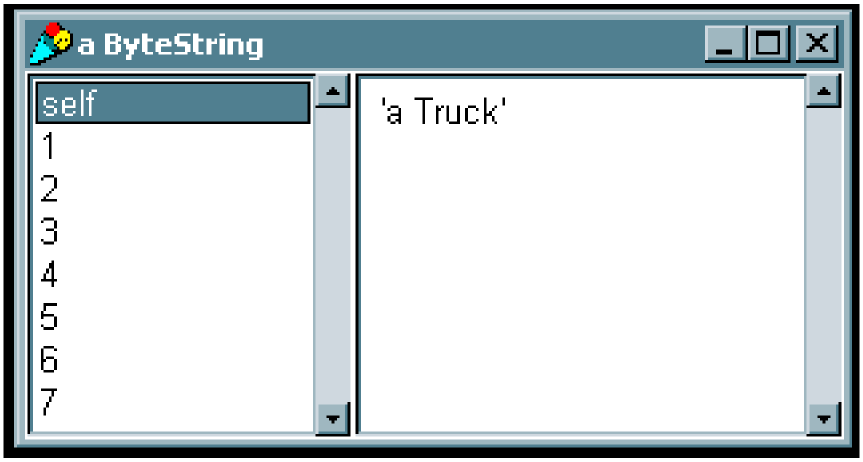
```
  "Returns True if the reciever is less than aComplex"
  aComplex isKindOf: Complex
         ifTrue: [^self abs < aComplex abs]
         ifFalse: [^self error: 'Not a complex number'].

max: aComplex
       "Returns the greater value of aComplex and the receiver"
       self < aComplex
              ifTrue: [^aComplex]
              ifFalse: [^self].

= aComplex
  "Returns True if the receiver is equal to aComplex"
  aComplex isKindOf: Complex
         ifTrue: [
                ^self realPart=aComplex realPart and: [
                           self imaginaryPart = aComplex imaginary
                           part ]]
                      ifFalse: [^self error: 'Not a complex number']

  hash
         "hashes the absolute value of the reciver"
         ^self abs hash.
```

- **Partial Hierarchy**



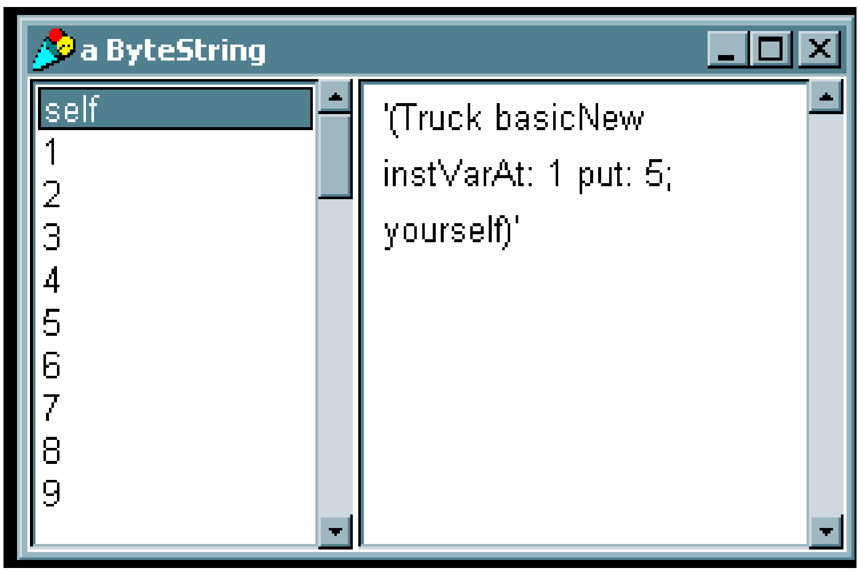
- **Type Conversion**
  - Converting to strings
    - To produce a string representation of an object use:
      - `objectName printString`

```
| aTruck |
aTruck := (Truck new) withSpeed: 5.
(aTruck printString) inspect.
```



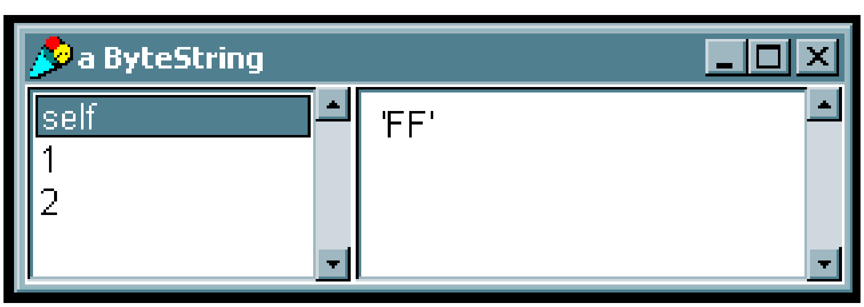
- `objectName storeString`

```
| aTruck |
aTruck := (Truck new) withSpeed: 5.
(aTruck storeString) inspect.
```



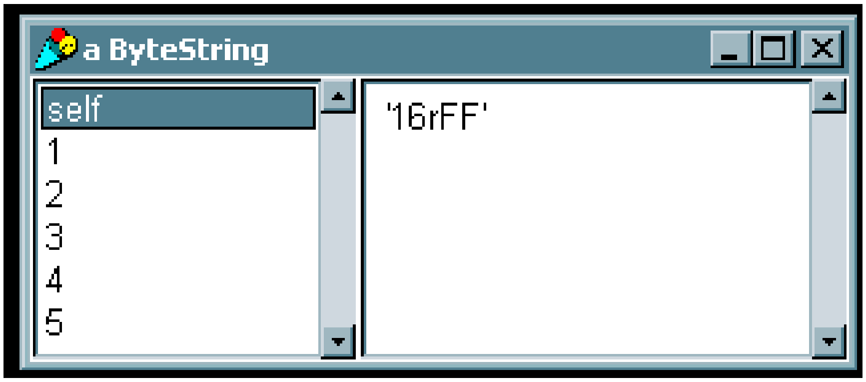
- To produce the string representation of a number, the above can be used, or more specialized methods may be used
  - `anInteger printStringRadix: aRadix` (used for base aRadix representation)

```
| anInteger |
anInteger := 255.
(anInteger printStringRadix: 16) inspect.
```



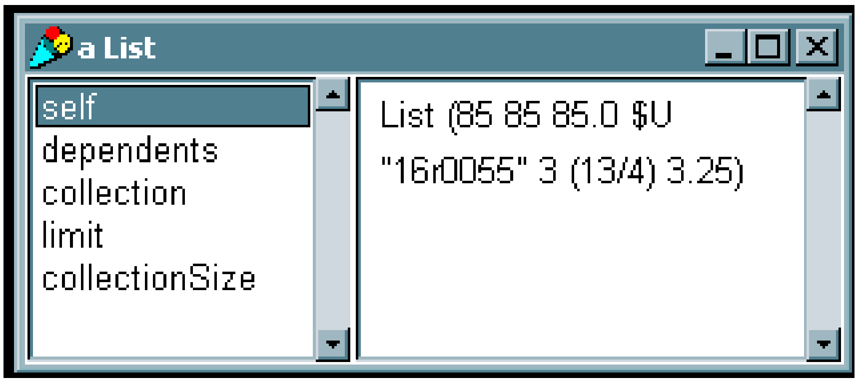
- `anInteger storeStringRadix: aRadix`

```
| anInteger |
anInteger := 255.
(anInteger storeStringRadix: 16) inspect.
```



- Converting strings to numbers
  - Requires streams to get strings from
    - This topic will be discussed in a later lecture.
  - Ex: Number `readFrom`: (ReadStream on: aStream)
- Type Conversion
  - Conversion is automatic and transparent
  - Conversion in direction integer -> fraction -> float to maintain accuracy
  - To explicitly do conversion use following methods
    - `asInteger`
    - `asFraction`
      - `asRational` in VisualWorks
    - `asFloat`
    - `asCharacter (integers only)`

```
| anInteger aFloat aList|
anInteger := 85.
aFloat := 3.25.
aList := List new.
aList add: anInteger asInteger.
aList add: anInteger asRational.
aList add: anInteger asFloat.
aList add: anInteger asCharacter.
aList add: aFloat asInteger.
aList add: aFloat asRational.
aList add: aFloat asFloat.
aList inspect.
```



- **Truncation, floor, ceiling and remainders**
  - Truncation done through `quo:` method
    - `11 quo: 5` => 2
    - `11 quo: -5` => -2
  - floor ceiling done though `//` operator
    - `11 // 5` => 2
    - `11 // -5` => -3
  - ceiling done through \\ operator
    - `11 \\ 5` => 3
    - `11 \\ -5` => -2
  - remainder is done through `rem:` method
    - `11 rem: 5` => 1
    - `11 rem: -5` => -1
- **Mathematical Operations**
  - Smalltalk provides basic subset of functions including
    - Trigonometry functions: `sin, cos, arcSin, arcCos`
    - Natural exponents and logarithms (`exp` and `ln`)
    - Exponents and logarithms
    - `gcd` and `lcd`
    - Ex:

```
55 gcd: 30 ← 5
6 lcm: 10 ← 30
0.523599 sin ← 0.5
6 exp ← 403.429
2.718284 ln ← 1
6 raisedTo: 3 ← 216
25 log: 5 ← 2
```

- **Date and Time**
  - Simple protocol for referencing and converting times & dates
  - Creating an time or date object

- Use `now` method for creating the current time
  - `currentTime := Time now.`
- Use `today` method for creating the current date
  - `currentDate := Date today.`
- You can create an object with both current date and time
  - `rightNow := Date dateAndTimeNow.`
  - `rightNow := Time dateAndTimeNow`
- Can create any time or date easily
  - `aDate := Date newDay: aDayOfTheYearInteger year: aYearInteger`
- Time and Date Conversions

- Timing execution and delays
  - Smalltalk provides a simple way to time the execution of a loop

```
| block1 block2 ms1 ms2 |
block1 := [100 timesRepeat: [Time now. Date today]].
ms1 := Time millisecondsToRun: block1.

block2 := [100 timesRepeat: [Time dateAndTimeNow]].
ms2 := Time millisecondsToRun: block2.
```

- Smalltalk includes a similar class Delay. The Delay class is useful for creating timers. Timers can be used to update clocks or send messages regularly.
  - Delay should be used with the wait method
  - The following shows a simple clock, which writes to the Transcript.

```
[[true] whileTrue:
       [Transcript show: (Time now printString).
       (Delay forSeconds: 1) wait]] fork.
```

Lecture 16: The Collection Classes

- **Smalltalk's optimized Collection classes**
  - Unlike C, Smalltalk provides optimized classes for most types of collections
  - There are three types of Collections
    - Not keyed
      - Example: Bag
    - Keyed by integer
      - Example: Array, List, OrderedCollection
    - Keyed by value
      - Example: Set, Dictionary
  - For most situations, one of 5 types will suffice
    - SortedCollection
      - Sorts elements when inserted
      - Example returns SortedCollection ('a' 'b' 'c')

```
| aSortedCollection |
aSortedCollection := SortedCollection new.
aSortedCollection add: 'c'.
aSortedCollection add: 'a'.
aSortedCollection add: 'b'.
aSortedCollection inspect.
```

    - List
      - Most flexible, keeps elements in the order in which they were added.
      - Lists can be sorted.
      - Elements can be inserted anywhere
      - Example returns List ('a' 'b' 'c')

```
| aList aSortedList|
aList := List new.
aList add: 'c'.
aList add: 'b'.
aList add: 'a'.
aSortedList := aList sort.
```

    - Array
      - Does not require adding, removing, or sorting elements
      - Example returns #('d' 'b' 'c')

```
| anArray |
anArray := Array new.
anArray at: 1 put: 'a'.
anArray at: 2 put: 'b'.
anArray at: 3 put: 'c'.
anArray at: 1 put: 'd'.
```

    - Set
      - Discards duplicate elements
      - Does not support replacing elements
      - Example
        - aSet ← Set ('a' 'b')
        - aList ← List ('a' 'b' 'a')

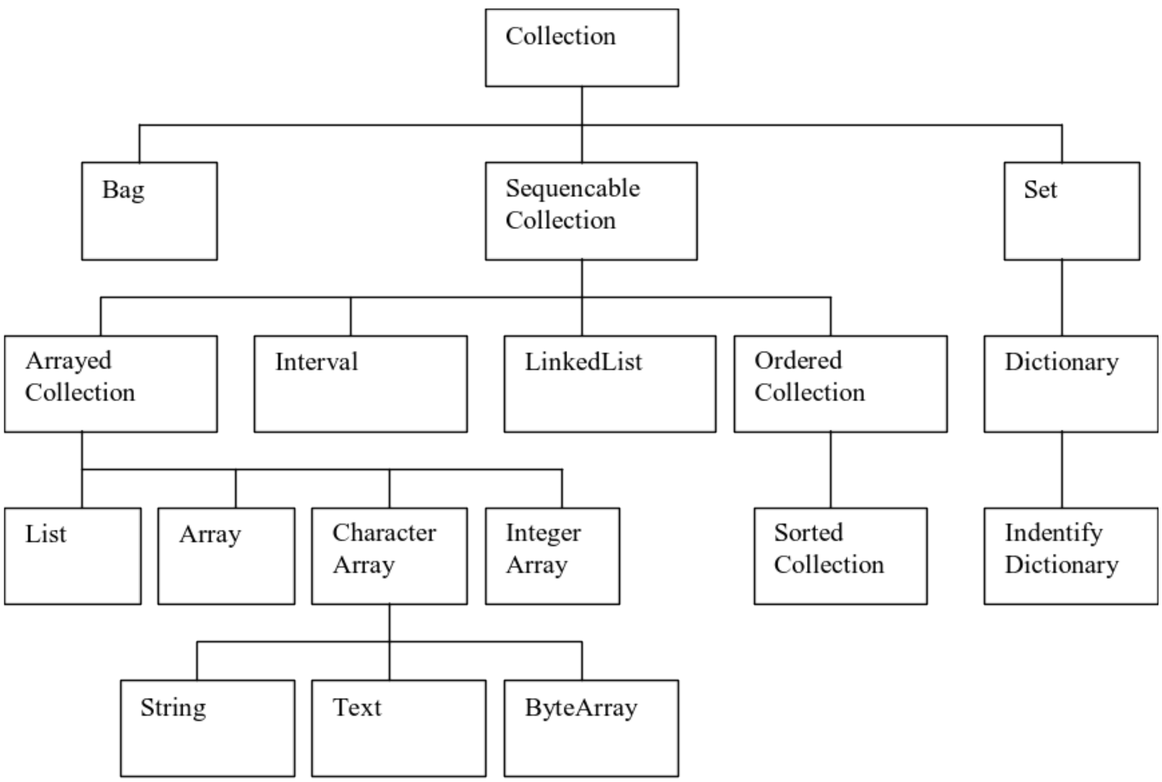
```
|  aList aSet |
aList := List new.
aList add: 'a'.
aList add: 'b'.
aList add: 'a'.
aSet := Set new.
aSet addAll: aList.
```

    - Dictionary
      - New Concept to C programmers: Dictionary

- Otherwise known as Associated Hashtable
  - Add keys and values, and reference values through keys
  - Useful for global variables
  - Possible to associate keys with any kind of object
  - Ex:

```
| aThesaurus aCollection |
aCollection := Bag new.
aCollection add: 'big'.
aCollection add: 'enormous'.
aCollection add: 'huge'.
aThesaurus := Dictionary new.
aThesaurus at: 'large' put: aCollection.
aThesaurus at: 'small' put: 'little'.
```

- **Partial Hierarchy**



- **Iteration (what you can do with collections)**
  - Iterate over a collection
    - `do: []`
      - Ex: (sum ← 15)

```
| sum aCollection |
aCollection := Bag new.
aCollection add: 3.
aCollection add: 5.
aCollection add: 7.
sum := 0.
aCollection do: [ :x | sum := sum + x].
```

    - `reverseDo: []`
      - Ex: (OrderedCollection(c b a))

```
| aReverseCollection aOrderedCollection |
aOrderedCollection := OrderedCollection new.
aOrderedCollection add: #a.
```

```
aOrderedCollection add: #b.
aOrderedCollection add: #c.
aReverseCollection := OrderedCollection new.
aOrderedCollection reverseDo:
       [:x | aReverseCollection add: x].
```

- `collect: []`
  - Useful to Create new collections from existing ones
  - Ex: (Bag(25 25 25… 0 0 0))

```
| someIntegers someNumbers|
someNumbers := Bag new.
1 to: 25 by: 0.2 do: [ :x | someNumbers add: x].
someIntegers := Bag new.
someIntegers := someNumbers collect:
        [ :x | x asInteger].
```

- Iterate over a collection and return a subset
  - `select: []`
    - Ex: (retuns only integers between 1 & 25 as floats)

```
| someIntegers someNumbers|
someNumbers := Bag new.
1 to: 25 by: 0.5 do: [ :x | someNumbers add: x].
someIntegers := Bag new.
someIntegers := someNumbers select:
        [ :x | (x // 1) asFloat  =  x].
```

  - `reject: []`
    - Ex: (returns only integers between 1 & 25 as floats)

```
| someIntegers someNumbers|
someNumbers := Bag new.
1 to: 25 by: 0.5 do: [ :x | someNumbers add: x].
someIntegers := Bag new.
someIntegers := someNumbers reject:
        [ :x | (x // 1) asFloat  ~=  x].
```

- Find occurrences of an object within the collection
  - `detect: []`
    - Ex: `#(a 'abc' 3 4 5) detect: [:x | x isInteger].` ← 3
    - Ex: `#(a 'abc' 3 4 5) findFirst: [:x | x isFloat]`
      `ifNone[nil]` ← nil
  - `findFirst: []`
    - Ex: `#(a 'abc' 3 4 5) findFirst: [:x | x isInteger].` ← 3
  - `findLast:`
    - Ex: `#(a 'abc' 3 4 5) findLast: [:x | x isInteger].` ← 5
- Perform special operations
  - `inject: into: []`
    - For using temp variables and initializing them outside the block
    - Ex: set the temp variable to 100
      `#(1 2 3) inject:100 into: [:x :y | x := x + y].` ← 106
  - `with: do: []`
    - takes one object from the receiver and one from the argument
    - Ex: (result aBag= #('aA' 'cC'))

```
| aBag |
aBag := Bag new.
#('a' 'b' 'c') with: #('A' 'Z' 'C') do:
    [:x :y | x asUpperCase = y
```

```
ifTrue: [aBag add: (x,y)]].
```

Lecture 17: An example using the Collection Classes

- Matrices are not provided by the collection classes, but can be added very easily. We will demonstrate the Collection classes by creating a Matrix class.
- The class should provide methods to add, multiply, and transpose matrices and scalars together.
- The matrix will be represented in row-major order, through a collection of rows, where each row is an ordered collection. To accomplish this, the matrix needs only two variables to keep count on the number of rows and columns
- The Class definition is straight forward

```
    OrderedCollection variableSubclass: #Matrix2D
    instanceVariableNames: 'numrows numcols '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Examples-Matrix2D'

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

    Matrix2D
    class instanceVariableNames: ''!
```

- The instance creation method creates a 2 dimensional matrix sized according to the arguments, and initializes all elements to 0.0. Note that the matrix is represented by aNumber1 OrderedCollections.

```
!Matrix2D class methodsFor: 'instance creation'!

rows: aNumber1 cols: aNumber2
      "Creates a 2D matrix of size aNumber1 X aNumber2 and
       initializes all elements to 0."

      | matrix |
      matrix := self new.
      1 to: aNumber1 do: [ :i | | temp |
             temp := OrderedCollection new.
             1 to: aNumber2 do: [ :j | temp addLast: 0.0 ].
                    matrix addLast: temp ].
      matrix setrows: aNumber1 cols: aNumber2.
      ^matrix! !
```

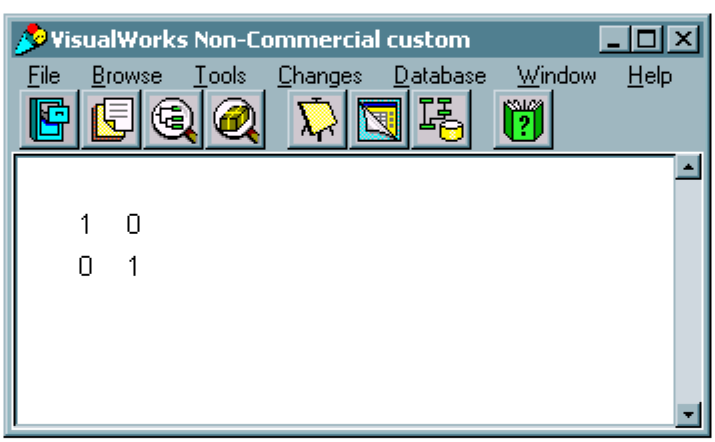
- The methods for accessing the matrix are straight forward as well, and are used only for getting elements and setting elements.

```
!Matrix2D methodsFor: 'accessing'!

at: anArray put: aNumber
    "Place an element (aNumber) in the row and column
     specified by anArray in the receiver."

    (self at: (anArray at: 1)) at:(anArray at:2) put:aNumber.!

atRow: aNumber1
    "Return an ordered collection from row aNumber1of the receiver."

    ^(self at: aNumber1).!

atRow: aNumber1 put: anOrderedCollection
    "Puts an OrderedCollection into the matrix as a row.”

    super at: aNumber1 put: anOrderedCollection.!
```

```
    atRow: aNumber1 atCol: aNumber2
        "Return an element from row aNumber1, column aNumber2
         in the receiver."

        ^(self at: aNumber1) at: aNumber2.!

atRow: aNumber1 atCol: aNumber2 put: aNumber3
        "Place an element (aNumber3) in row aNumber1, column aNumber2
         in the receiver."

        (self at: aNumber1) at: aNumber2 put: aNumber3.!

cols
        "Returns the number of cols in the matrix."

        ^numcols.!

readAt: anArray
        "Returns an element from the row and column
         specified by anArray in the receiver."

        ^(self at: (anArray at: 1)) at:(anArray at:2).!

rows
        "Returns the number of rows in the matrix."

        ^numrows.!

setrows: aNumber1 cols: aNumber2
        "Sets the size of the matrix."

        numrows := aNumber1.
        numcols := aNumber2.! !
```

- To illustrate the access methods, we will create a 2x2 identity matrix with the code below. Recall an identity matrix is one which the top left to bottom right diagonal has 1 as the values of its elements, and all other values are 0.

```
| matrix1 |
matrix1 := Matrix2D  rows: 2 cols: 2.
matrix1 at: #(1 1) put: 1.
matrix1 at: #(1 2) put: 0.
matrix1 at: #(2 1) put: 0.
matrix1 at: #(2 2) put: 1.
matrix1 writeToTranscript.
```

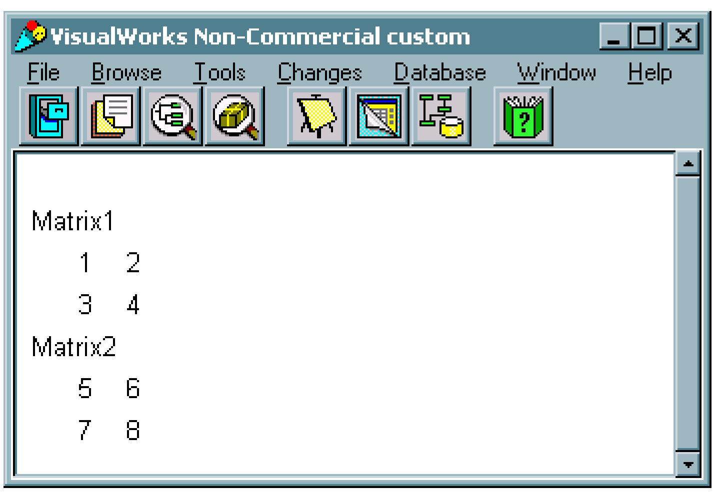


- The method `writeToTranscript`, as used above prints each row, an element at a time.

```
writeToTranscript
   "Writes the matrix to the Transcript."

   Transcript show: ' ';cr.
   1 to: (self rows) do: [ :i |
         Transcript show: ' '; tab.
         1 to: (self cols) do: [ :j |
               Transcript show:
                      (self atRow: i atCol: j) printString; tab.
         ].
         Transcript show: ' ';cr.
   ].
```

- Although mathematical operations may appear to be complicated, the operations to be applied to the matrices are simple Collection manipulations.
- For the operation examples, the following matrices will be used. The code to create them is also shown below.

```
| matrix1 matrix2|
matrix1 := Matrix2D  rows: 2 cols: 2.
matrix1 at: #(1 1) put: 1.
matrix1 at: #(1 2) put: 2.
matrix1 at: #(2 1) put: 3.
matrix1 at: #(2 2) put: 4.
Transcript show: 'Matrix1'.
matrix1 writeToTranscript.
matrix2 := Matrix2D  rows: 2 cols: 2.
matrix2 at: #(1 1) put: 5.
matrix2 at: #(1 2) put: 6.
matrix2 at: #(2 1) put: 7.
matrix2 at: #(2 2) put: 8.
Transcript show: 'Matrix2'.
matrix2 writeToTranscript.
```



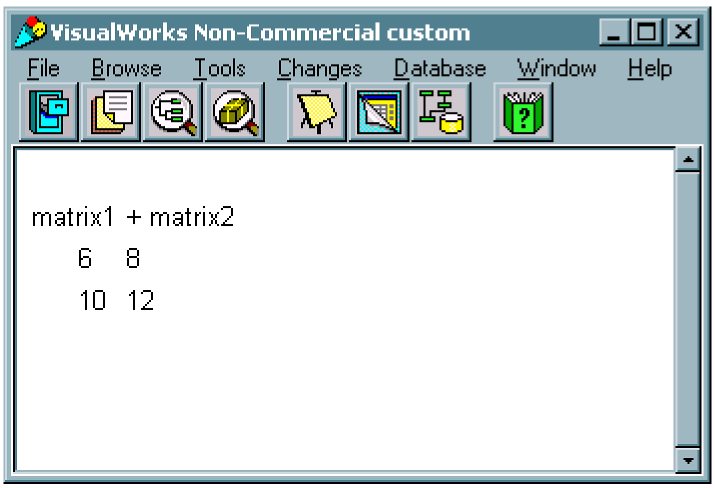
- matrixAdd: aMatrix adds aMatrix to the receiver and returns the sum of the two. A check is done to make sure both matrices are the same size

```
matrixAdd: aMatrix
   "Adds the receiver and aMatrix, that is, Receiver + aMatrix."
```

```
| matrix |
( (numrows == ( aMatrix rows)) & (numcols == ( aMatrix cols)) )
ifFalse:
          [ Transcript nextPutAll:
                'matrixAdd - bad matrix size' ;endEntry.
       ^nil. ].
matrix := Matrix2D rows: numrows cols: numcols.
1 to: numrows do: [ :row |
       1 to: numcols do: [ :col |
              matrix atRow: row atCol: col put:
                     ((self atRow: row atCol: col) +
                         (aMatrix atRow: row atCol: col)).
       ].
].
"Returns a new matrix"
^matrix
```

- Below is an example of adding two matrices.

```
Transcript show: 'matrix1 + matrix2'.
(matrix1 matrixAdd: matrix2) writeToTranscript.
```



- `matrixMult: aMatrix` multiplies the receiver and aMatrix and returns the product. A check is done to make sure the number of rows in the receiver is equal to the number of columns in aMatrix (rule of matrix multiplication).
- Recall the product of two matrices is as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \cdots & A_{1m} \\ A_{21} & A_{22} \cdots & A_{2m} \\ \vdots & & \\ A_{n1} & A_{n2} \cdots & A_{nm} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \cdots & B_{1m} \\ B_{21} & B_{22} \cdots & B_{2m} \\ \vdots & & \\ B_{n1} & B_{n2} \cdots & B_{nm} \end{bmatrix}$$

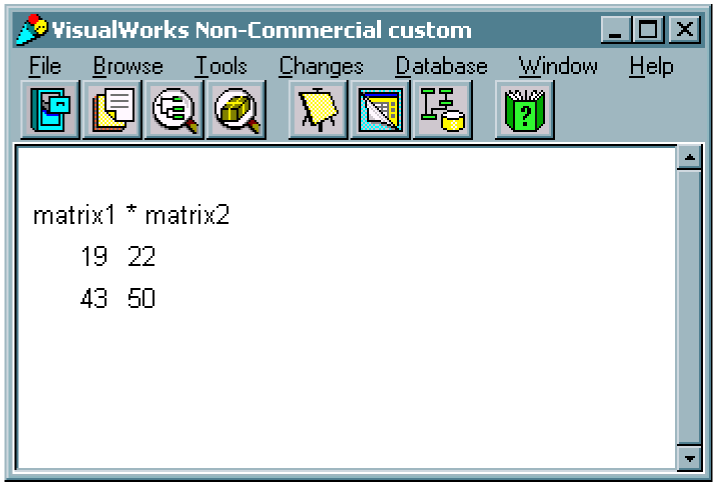
$$AxB = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} + \ldots + A_{1n}B_{n1} \cdots & A_{11}B_{1m} + A_{12}B_{2m} + \ldots + A_{1m}B_{nm} \\ A_{21}B_{11} + A_{22}B_{21} + \ldots + A_{2n}B_{n1} \cdots & A_{21}B_{1m} + A_{22}B_{2m} + \ldots + A_{2m}B_{nm} \\ \vdots & \\ A_{n1}B_{11} + A_{n2}B_{21} + \ldots + A_{nm}B_{n1} \cdots & A_{n1}B_{1m} + A_{n2}B_{2m} + \ldots + A_{nm}B_{nm} \end{bmatrix}$$

- The following code implements the equation above:

```
matrixMult: aMatrix
       "Multiplies the receiver and aMatrix, that is, Receiver *
aMatrix."

       | nrows ncols matrix sum |
       nrows := self rows.
       ncols := self cols.
       (ncols == ( aMatrix rows))
       ifFalse: [ Transcript nextPutAll:
           'matrixMult - bad matrix size' ;endEntry.
                       ^nil. ].
       matrix := Matrix2D rows: nrows cols:(aMatrix cols).

       1 to: (aMatrix cols) do: [ :k |
              1 to: nrows  do: [ :i |
                     sum := 0.
                     1 to: ncols do: [ :j |
                            sum :=  sum + ((self atRow: i atCol: j) *
                                (aMatrix atRow: j atCol: k)).
                     ].
                     matrix atRow: i atCol:k put: sum.
              ]
       ].
       ^matrix
```

- Below is an example of multiplying two matrices.

```
Transcript show: 'matrix1 * matrix2'.
(matrix1 matrixMult: matrix2) writeToTranscript.
```

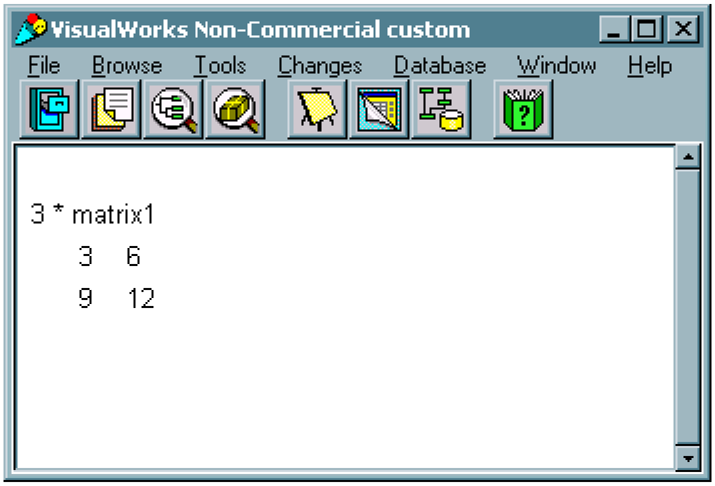


- The methods to add and multiply scalars are very similar to the matrixAdd: method, but even simpler.

```
scalarAdd: aNumber
    "Adds aNumber to each element of the receiver."

    | nrows ncols matrix |
    nrows := self rows.
    ncols := self cols.
    matrix := Matrix2D rows: nrows cols: ncols.
    1 to: nrows do: [ :row |
           1 to: ncols do: [ :col |
                  matrix atRow: row atCol: col put:
                         ( self atRow: row atCol: col ) + aNumber.
           ].
    ].
    ^matrix

scalarMult: aNumber
    "Multiplies each element of the receiver by aNumber."

    | nrows ncols matrix |
    nrows := self rows.
    ncols := self cols.
    matrix := Matrix2D rows: nrows cols: ncols.
    1 to: nrows do: [ :row |
           1 to: ncols do: [ :col |
                  matrix atRow: row atCol: col put:
                         ( self atRow: row atCol: col ) * aNumber.
           ].
    ].
    ^matrix
```

- Below is an example of multiplying a scalar and a matrix.

```
Transcript show: '3 * matrix1'.
(matrix1 scalarMult: 3) writeToTranscript.
```

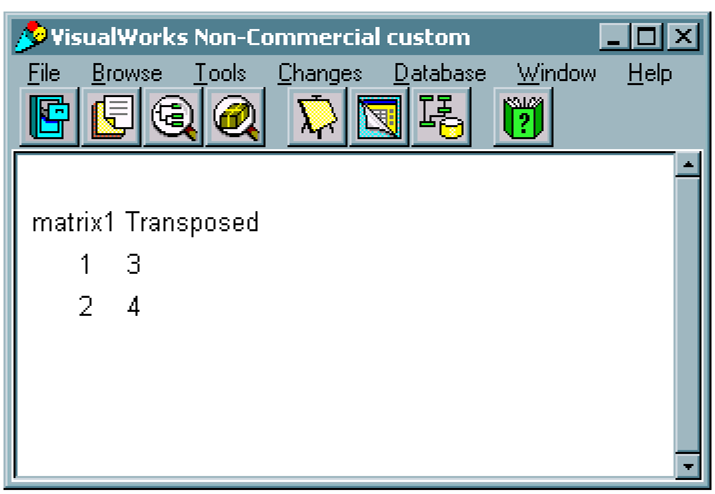


- The Transpose exchanges rows and columns.

```
transpose
      "Returns the transpose of the receiver."

      | nrows ncols matrix |
      nrows := self rows.
      ncols := self cols.
      matrix := Matrix2D rows: nrows cols: ncols.
      1 to: nrows do: [ :row |
            1 to: ncols do: [ :col |
                  matrix atRow: row atCol: col put:
                     ( self atRow: col atCol: row ).
            ].
      ].
      ^matrix
```

- Below is an example of transposing matrix.

```
Transcript show: 'matrix1 Transposed'.
(matrix1 transpose) writeToTranscript.
```



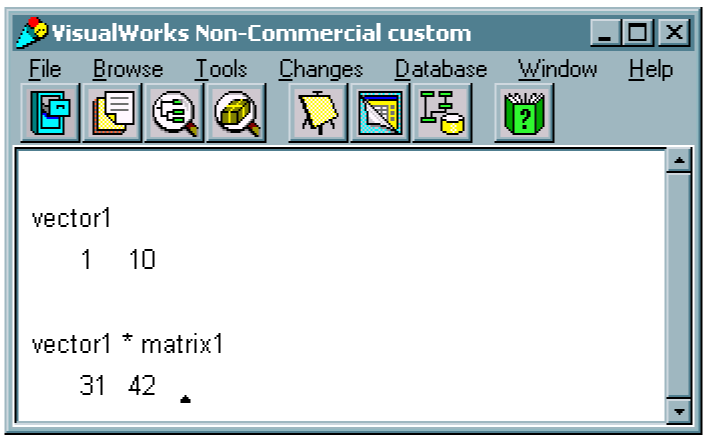
- Vectors are easily represented through the implementation of the Matrix class we have demonstrated, since a vector is nothing more than a single row of a matrix.
- Recall the product of a vector and a matrix is a vector as follows:

$$V = [V_1 \quad V_2 \ \ldots \ V_n], \quad M = \begin{bmatrix} M_{11} & M_{12} \ldots & M_{1m} \\ M_{21} & M22 \ldots & M_{2m} \\ \vdots & & \\ M_{n1} & M_{n2} \ldots & M_{nm} \end{bmatrix}$$

$$V \times M = [V_1 M_{11} + V_2 M_{21} + \ldots + V_n M_{n1} \ \ldots \ V_1 M_{1m} + V_2 M_{2m} + \ldots + V_n M_{nm}]$$

- The following code will create a vector and multiply it by matrix1

```
vector1 := Matrix2D rows:1 cols:2.
vector1 at: #(1 1) put: 1.
vector1 at: #(1 2) put: 10.
Transcript show: 'vector1'.
vector1 writeToTranscript.
Transcript show: 'vector1 * matrix1'.
(vector1 matrixMult: matrix1) writeToTranscript.
```



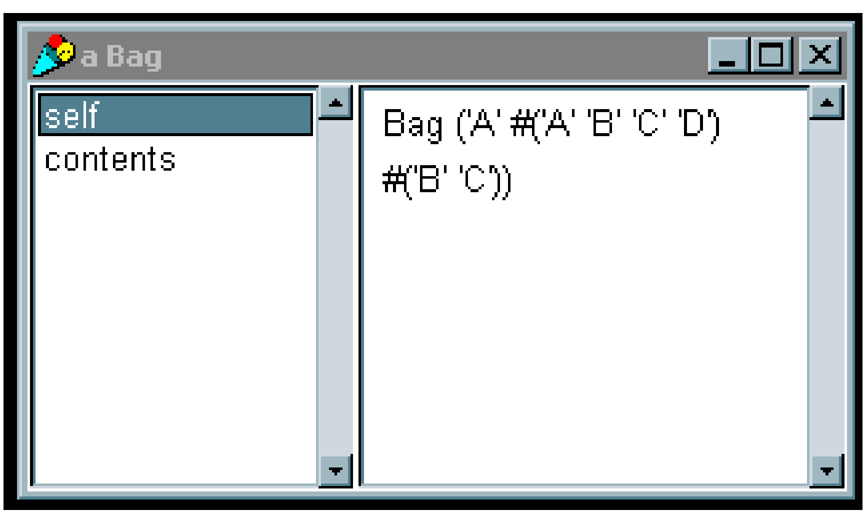
Lecture 18: The Stream Classes

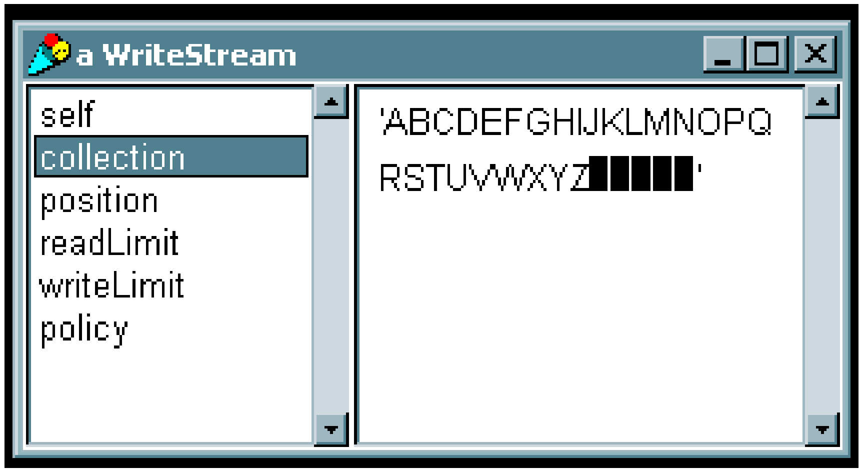
- **Streams**
  - Streams provide basic communication between the Virtual machine and the system
  - Types of streams
    - Semaphores
    - Sockets
    - Files
    - `stdin` & `stdout`
  - IMPORTANT: The programmer must close all open streams. Smalltalk will not close them for you, as in most compiled languages. The operating system has a limit on the number of open streams, and will quickly run out if the streams are not closed
- **Important methods for all Streams**
  - Accessing
    - `next` returns the next object in the stream
    - `next: anInteger` returns the next `anInteger` number of objects
    - `contents` returns all of the objects in the collection
    - `close` closes the stream
    - Ex:

```
| aStream anObject |
aStream := ReadStream on: #('A' 'B' 'C' 'D').
anObject := Bag new.
anObject add: aStream next.
anObject add: (aStream next: 2).
anObject add: aStream contents.
aStream close.
anObject inspect.
```

  - Writing
    - `nextPut: anObject` places `anObject` in the stream so that it is the next accessible
      - Ex: Generate & write the alphabet to a stream

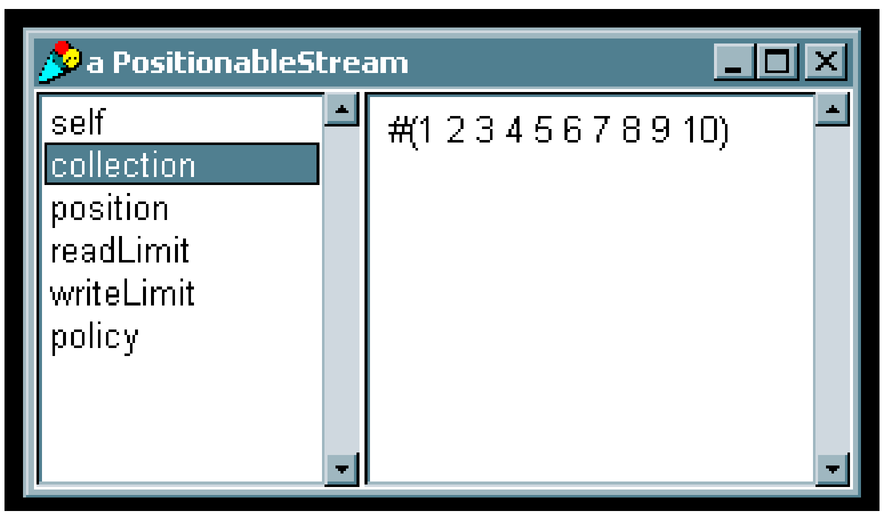
```
| aStream|
aStream := WriteStream on: (String new).
65 to: 90 do: [:aNumber | aStream nextPut: aNumber
asCharacter].
aStream close.
aStream inspect.
```





- `nextPutAll: aCollection` puts the contents of `aCollection` into the stream.
  - Example: Putting the number 1->10 into a Stream

```
| aStream aCollection |
aCollection := Array new:10.
1 to: 10 do: [ :aNumber | aCollection at: aNumber put:
aNumber].
aStream := ReadWriteStream on: (Array new: 10).
aStream nextPutAll: aCollection.
aStream close.
aStream inspect.
```



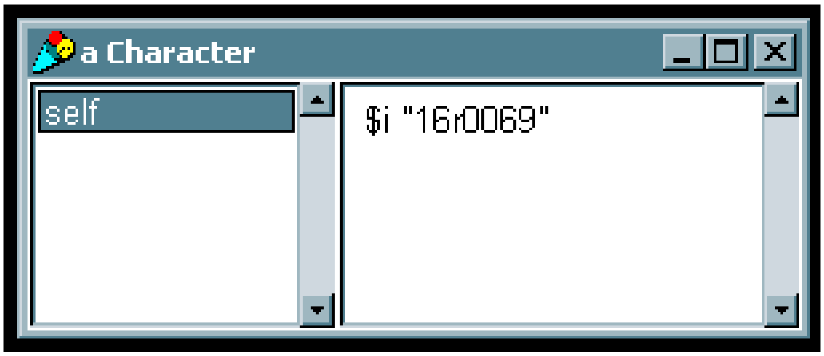
  - Example: Different way to get the same results. Which is the safer way? Which is the more “elegant” way?

```
| aStream aCollection |
aCollection := Array new: 10.
aStream := PositionableStream on: (aCollection).
1 to: 10 do: [ :aNumber | aCollection at: aNumber put: aNumber].
aStream close.
aStream inspect.
```

- **Important methods for Positionable Streams**
  - Accessing
    - `position` returns the position in the stream
    - `peek` returns the next object without advancing the position
    - `reset` resets the position in the stream

- `skip: anInteger` skips anInteger positions in the stream
- Ex:

```
| aStream anObject |
anObject := 'This is a single String'.
aStream := ReadWriteStream on: (String new).
aStream nextPutAll: anObject.
aStream reset.
aStream skip: 2.
(aStream peek) inspect.
```



- **Important methods for ReadStreams**
  - Instance Creation
    - `ReadStream on: aCollection`
  - All other positionable stream methods and general methods will work except for ones which write (such as at:put: methods)
- **Important methods for WriteStreams**
  - Instance Creation
    - `WriteStream on: aCollection`
  - Accessing
    - `flush` write all unwritten information to the stream
      - Good "book-keeping" habit to do before closing streams or saving images.
  - Similar to ReadStream, can access all methods of more general streams, but cannot read from streams
- **Important methods for External and File Streams**
  - Instance creation
    - 2 step process- make the filename, then apply the method to the filename. For the entire list of possible methods, refer to LaLonde, or the system browser under Filename->stream creation.

```
| aStream aFilename |
aFilename := Filename named: 'yourfile.txt'.
aStream := aFilename writeStream.
```

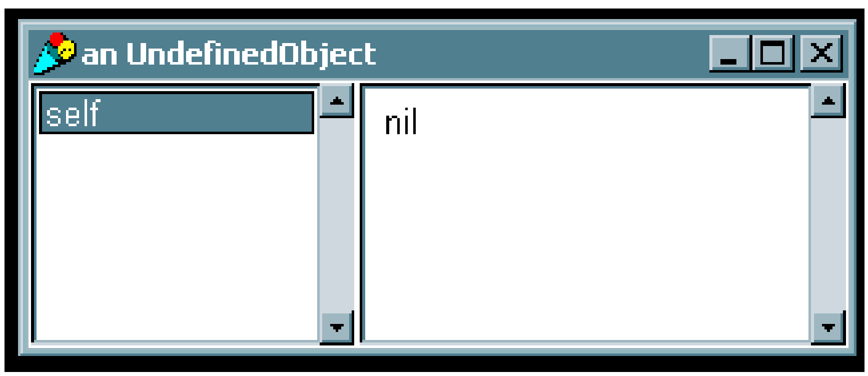
  - Accessing
    - `nextNumber: n` returns the next n bytes in the stream
    - `nextString` returns the next String from the stream.
    - `skipwords: nWords` advances the position nWords number of words (2 bytes, not to be confused with strings)
    - `wordPosition` returns the position in words
    - `wordPosition: wp` advances the position to wp in words
  - Writing
    - `nextPut: anObject` places `anObject` in the stream so that it is the next accessible
    - `nextPutAll: aCollection` puts the contents of `aCollection` into the stream.

- Example, writing to a file.

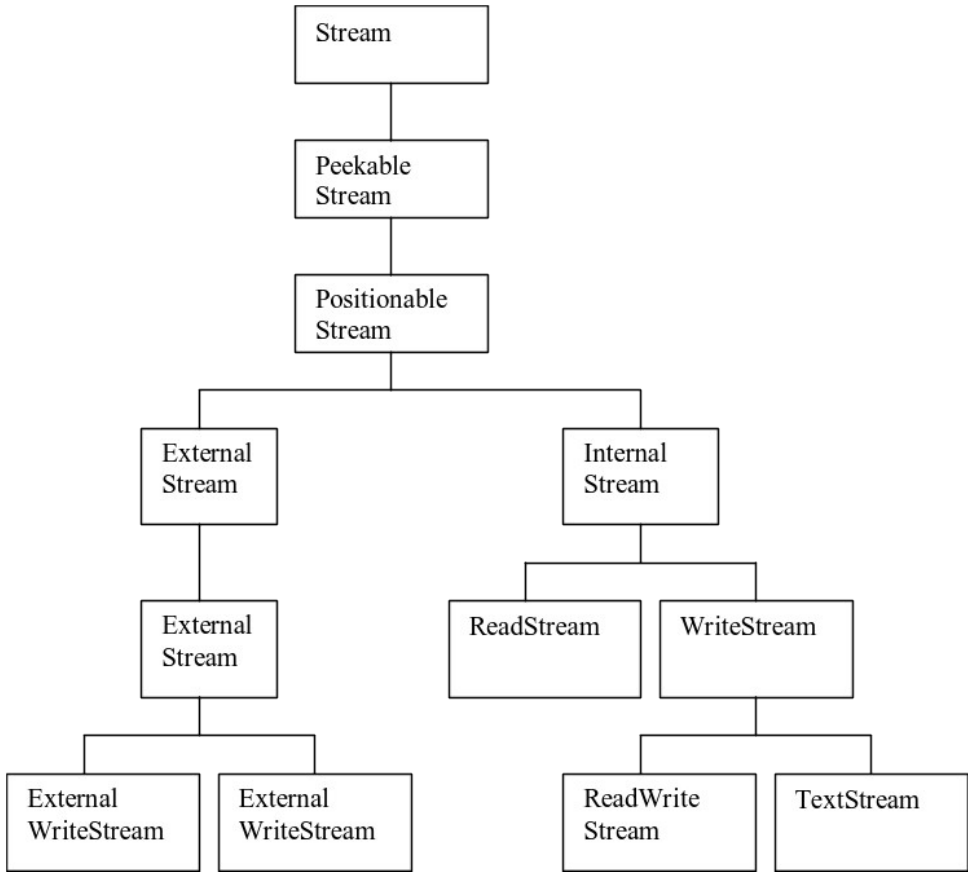
```
| aStream aFilename |
aFilename := Filename named: 'temp.txt'.
aFilename delete.
aStream := aFilename readWriteStream.
1 to: 20 by: 5 do: [ :aPosition |
   aStream wordPosition: aPosition.
   aStream nextPut: $D].
aStream close.
```

- **Common Mistakes**
- Writing a collection to a stream is not the same as writing the contents of the collection.
  - Example: What is wrong with this? Shouldn't we see the number 2 instead of 'nil'? No, the first object is the collection, the second object is the end of the stream.

```
| aStream aCollection |
aCollection := Array new: 10.
aStream :=ReadWriteStream on: (aCollection).
1 to: 10 do: [ :aNumber | aCollection at: aNumber put: aNumber].
aStream reset.
(aStream peek) inspect.
```



- **Hierarchy**



Stream
Peekable Stream
Positionable Stream
External Stream
Internal Stream
External Stream
ReadStream
WriteStream
External WriteStream
External WriteStream
ReadWrite Stream
TextStream

Lecture 19: Matrix Example using Streams

- Recall the Matrix example. Now, rather than getting the matrix from standard in, we will read the matrix from a file. To maintain simplicity, we will keep the rules strict, but to allow for flexibility we will intelligently get the dimensions of the matrix. The rules of the file are as follows:
  - One matrix to a file
  - The matrix must be complete. That is, all rows must contain the same number of elements
  - The matrix will start with '[' and end with ']'.
  - Each row will be separated by a carriage return
  - Each element will be separated by white space (tabs, cr's, spaces).
  - No element may be negative
- So, using these rules, a 3x3 identity matrix would be represented as below:

```
[ 1 0 0
0 1 0
0 0 1 ]
```

- We need two methods, one to read from a file, and one to write to a file. We will add these methods to the Matrix2D class.
- The read method, `fromFile: aMatrix` is and instance creation method (like `new`). The method follows a simple parsing algorithm:

```
Get characters until '[';
rowCount = 1;
Get nextString;
    If nextString = '\n'
            increase rowCount;
            add Collection to Matrix
            reset Collection to nil
    If nextString is a number then add it to Collection
```

- This method accomplishes this by reading one character at a time, building up a string to be converted into numbers.
- Since we don't know how big the matrix will be, we can't store the elements immediately into the matrix. Instead, each row is read into an `OrderedCollection`.
- Once the `OrderedCollection` object is built, the `addLast`: method is called to add the `OrderedCollection` object to the matrix. The number of rows is then incremented.
- The following code implements the method as discussed above

```
fromFile: aName
    "Creates a 2D matrix from a file of the name aName"

    | aStream aFilename aString aChar aCollection aMatrix|
    aFilename := Filename named: aName.
    aStream := aFilename readStream.
    aChar := aStream next.

    "Create the Matrix"
    aMatrix := Matrix2D rows:0 cols:0.

    "eat up everything until the open bracket"
    [ aChar = $[ ]
           whileFalse: [ aChar := aStream next].
    "matrix has started"
    aCollection := OrderedCollection new.
    aString := String new.
```

```
[ aChar = $] ]
  whileFalse: [
       aChar := aStream next.
       aChar asInteger = 13 "cr"
              ifTrue: [
                     aString size > 0
                       ifTrue: [
                         aCollection add:
                            (aString asNumber).
                         aString := String new.
                       ].
                     aMatrix addLast: aCollection.
                     aMatrix setrows: (aMatrix rows + 1)
                            cols: (aCollection size).
                     aCollection := OrderedCollection new.
              ].
              aChar isSeparator "any white space"
              ifTrue: [
              aString size > 0
                     ifTrue: [
                            aCollection add:
                              (aString asNumber).
                            aString := String new.
                     ]
              ].
       (aChar isDigit)
              ifTrue: [aString := aString,
                     (aChar digitValue printString)].
].
aStream close.

"Add the last one, since the ']' was on the last line"
aMatrix addLast: aCollection.
aMatrix setrows: (aMatrix rows + 1)
       cols: (aCollection size).
aCollection := OrderedCollection new.

^aMatrix
```

- The method to write the matrix to a file is considerably more simple. A ‘[‘ is written, then each row is written as characters, then a ‘]’ is written.

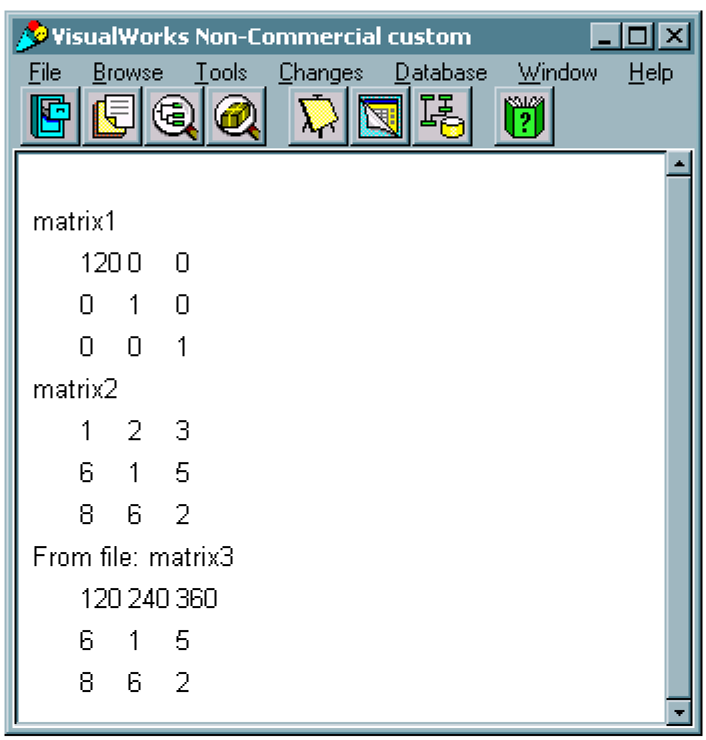
```
writeToFile: aName
       "Writes the matrix to the file aName. The format is
   such that fromFile:    can be called to read it back
   into a matrix."

       | aStream aFilename|
       aFilename := Filename named: aName.
       aStream := aFilename writeStream.

       aStream nextPut: $[.
       1 to: (self rows) do: [ :row |
              1 to: (self cols) do: [ :col |
                     ((self atRow: row atCol: col) printString)
                  do: [ :char |
                            aStream nextPut: char
                     ].
                     aStream nextPut:$ . "space"
              ].
              row = (self rows)
                     ifFalse: [
                        aStream nextPut: 13 asCharacter. "cr"].
```

```
        ].
        aStream nextPut: $].
        aStream close.
```

- To illustrate the use of the new file methods, we will read two matrices from text files ("matrix1.txt" and "matrix2.txt"), then multiply them together. Their product will be written to a file ("matrix3.txt"). Below is a screen capture of the two input text files.

- The code below will now multiply the matrices together and write the product to a file. The code also reads the output file back in and prints it to the Transcript as a form of visual sanity check.

```
| matrix1 matrix2|
matrix1 := Matrix2D fromFile: 'matrix1.txt'.
Transcript show: 'matrix1'.
matrix1 writeToTranscript.
matrix2 := Matrix2D fromFile: 'matrix2.txt'.
Transcript show: 'matrix2'.
matrix2 writeToTranscript.
(matrix1 matrixMult: matrix2) writeToFile: 'matrix3.txt'.
Transcript show: 'From file: matrix3'.
(Matrix2D fromFile: 'matrix3.txt') writeToTranscript.
```



- The code results in the output file