if (reader.HasRows)
{
    reader.Close();
    Response.Redirect(string.Format("WebForm2.aspx?UserName={0}&Password={1}", TextBox1.Text, TextBox2.Text));
}
else
{
    reader.Close();
    Label1.Text = "Wrong user or password";
}

"The straw that really broke the camel’s back in this case was the naming of WebForm2."
Issues
Names

Structure
  fullNames
  ClassName
  No "_"
  No abbreviations
  Trivial issue

Content
  Use names that convey role of variable
  Help the reader understand code
  Hard
Formatting

Be consistent

Indent to show block structure

control-o

Formats for you
Information Hiding

Don't let outside world know how class works

Don't provide access to internal data structures
Inheritance

B is a type of A
  B might be subclass of A

Class A uses a B
  B is a field in A

LinkedList is not a type of a Node

Stack is not a type of a LinkedList
Single Abstraction

A Class represents a single abstraction

Stack and Nodes are two separate abstractions
Abstraction = Data + Operations

Class has both
  Operations
  Data
Transcript

Transcript (Console) are used for debugging

Methods should return values not print to the Transcript
Strings as Exceptions

Don't return a string from method to indicate error condition

How does calling code tell difference between
  Actual value
  Error
Class Methods verses instance methods
Average of odd numbers
A Solution

Collection>>average

self isEmpty ifTrue: [^0].
^self sum / self size

Collection>>sum

self isEmpty ifTrue: [^0].
^self fold: [:a :b | a + b]

Collection>>odds

^self select: [:each | each odd]

Collection>>averageOfOdds

^self odds average

testAverageOdds

self

assert: #(1 2 3) averageOfOdds = 2;
assert: #(5) averageOfOdds = 5;
assert: #() averageOfOdds = 0;
assert: #(2 4 6) averageOfOdds = 0
averageOdd

<table>
<thead>
<tr>
<th>total</th>
<th>count</th>
</tr>
</thead>
</table>
total := 0.
count := 0.
self isEmpty
  ifTrue: [^total]
  ifFalse:
    [1 to: self size
do:
      [:x |
        (self at: x) odd
          ifTrue:
            [total := total + (self at: x).
             count := count + 1]].
count = 0 ifTrue: [^count].
^total / count]
Issues
avgOfOdd

| partialSum oddCount currentIndex |
oddCount := partialSum := 0.
currentIndex := 1.
[currentIndex <= self size] whileTrue:
[ (self at: currentIndex) \2 = 0
  ifFalse: [ oddCount := oddCount + 1.
    partialSum := partialSum + (self at: currentIndex).
  ].
  currentIndex := currentIndex + 1.
].
[partialSum / oddCount]
on: ZeroDivide
do:
[ :exception |
  Transcript show: 'Divide by zero exception';
  cr].
^(partialSum/oddCount).
arrayAverage
  "Gets the average of all odd integers in the array"

| total |

"Clear the total, read in the array, gather only the odd numbers, calculate"
total := 0.
self do:[:each |
  each isNil
  ifFalse:
    ifFalse:
      [each even ifFalse: [each isInteger ifTrue: [total := each + total]]]].^total
arrayAverage
| a sum n_odd average |
a := self.
sum :=0.
n_odd :=0.
(a collect: [:each|each odd ifTrue:[(sum :=sum+each).
n_odd :=n_odd+1]]).
average :=sum/n_odd.
^average.
oddAvg
|v sum temp average oddNos|

temp:=1.
sum := 0.
oddNos := 0.

[temp<=self size]
whileTrue:

[  
v:=(self at:temp)asInteger.
  v\2=0 ifTrue:
    "This is to check the modulus..If mod =0 ; then its is an even number "
    sum:=sum+v.
    temp:=temp+1.
  ]
  ifFalse:
    "if the mod is not=0 then add the number to odd number's list and traverse to next number of the array"
    sum:=sum+v.
    temp:=temp+1.
    oddNos := oddNos+1.].

average:=(sum/oddNos)asInteger.  
^average.  "finding the average of the Odd numbers"
oddAvg

| total oddCount temp avg someVar |

```plaintext
total := 0.
oddCount := 0.
someVar := 1.
avg := 0.
[someVar <= self size]       "iterating all array elements through a while loop and
checking which elements are not divisible by 2"
whileTrue:

temp := self at:someVar.
   temp \ 2 ~= 0 "checking if the array element is odd"
   ifTrue:

       total := total + temp.   "total is a variable which sums up all odd numbers in

the array"
       oddCount := oddCount + 1. "oddCount variable counts the number of odds in the array"
   ].
someVar := someVar + 1.
].
oddCount = 0         "this IF condition checks if there are zero odd
numbers in the array"
ifTrue:[ ^'there are no odd numbers in this array'.]
ifFalse:

    avg := total / oddCount.
    ^avg.
].
```
Issues

extractOddNumbersFromArray
"returns an array consisting of only odd numbers"

^self reject: [:each | each even ]
avgOfOddNos
|i j p q |
i := 1.
j:=0.
p:=0.
[i<=self size] whileTrue:
    [q:=self at: i.
     q\2 = 1
     ifTrue: [
      p:= p+q.
      j:=j+1.
     ].
     i := i+1.
    ].
^p/j
averageOfOddNos
| count sumOdd |
sumOdd := 0.
count := 0.  "Variable to keep track of the number of odd numbers in the array"

"IfTrue block is executed for array element only if it the element is odd. "
self do: [:each | ( each odd) ifTrue: [ count := count + 1.  
sumOdd := sumOdd + each].].

(count = 0)
  ifTrue: ["0 -No odd numbers"]
  ifFalse: [^ ( sumOdd / count) asFloat.].
Issues

OddArrayAverage
"Returns the average of all the odd numbers in the array"

| average numElements |
average := 0.
numElements := 0.
self do:
    [:each |
        each odd
        ifTrue: [average := average + each.
            numElements := numElements + 1]].
numElements = 0 ifTrue: [^0].
^average := average // numElements
Issues

averageOddNumbers
"Calculates the Average of the odd numbers in the Array assuming all the elements are numbers. Returns 0 if there are no odd numbers and nil if it is an empty array"

| oddElementsSum numberOfOddElements |
oddElementsSum:=0.
numberOfOddElements:=0.
self isEmpty ifTrue:[^nil].
1 to: self size do:
  [:idx |
    (((self at: idx)\2) == 1) ifTrue:
       numberOfOddElements :=numberOfOddElements + 1]].
^numberOfOddElements = 0 ifTrue:[0]
ifFalse:
  [(oddElementsSum/numberOfOddElements) asFloat].

Tuesday, October 11, 2011
averageOfOdds
| count  dividend |
count := 0.
dividend := 0.
self collect: [:each | each odd
   ifTrue:[count := count + each.
       dividend := dividend + 1.]].
count > 0
   ifTrue:[^count / dividend ]
   ifFalse:[^0].
Issues

calcOddValAvg

| sum val i avg |

i:=1.
sum:=0.
[i<=self size] whileTrue:
[ 
val:=(self at:i)asInteger.
val\2=1 ifTrue:[

sum:=sum+val.
i:=i+1.
]
ifFalse:[i:=i+1.].
]

avg:=(sum/self size)asInteger.
^avg.
oddNumbersAverage

|oddNumbers oddNumberSum average|
oddNumberSum := 0.
oddNumbers := self select: [:each| each odd].
oddNumbers do: [:each | oddNumberSum := oddNumberSum + each].
[average := (oddNumberSum/(oddNumbers size)).
   Transcript show: average printString; cr]
   on: ZeroDivide
   do: [:exception | Transcript show: 'No odd numbers in Array'; cr. exception
   resume].
   ^(average)
TestOddArrayAverage4
  "Fourth test of the method OddArrayAverage"
  | testArray |
  testArray := #( 1 2 3 3 4 5 6 8 9 13 15).
  self assert: testArray OddArrayAverage = 7.
TestSum
|Average|
Average:= #(2 3 5 1) oddAvg.
self assert: Average = 3.
valuesBetween: a and: b
A solution

valuesBetween: a and: b

^self select: [:each | each > a and: [each < b]]
valuesBetween: a and: b

| resultArray |

"Check if a and b are in proper order ie. a<b"

a <= b ifTrue:[
    resultArray := self select: [:each | each >= a ].
    resultArray := resultArray select: [:each | each <= b ].

] ifFalse:[
    resultArray := self select: [:each | each <= a ].
    resultArray := resultArray select: [:each | each >= b ].
].

resultArray isEmpty ifTrue:[^('No array elements within the specified range')].

^resultArray.
valuesBetween: a and: b
| c d |
c := (self select: [:each | each>a] ).
d := (c select: [:each | each<b] ).
^d.
valuesBetween:a and: b
(a>b)
ifTrue: [
^self select: [: each | each <a and: [each > b] ].]
ifFalse: [
^self select: [: each | each >a and: [each < b] ].]
valuesBetween: a and: b

| newArray x y |
 x := a.
y := b.
newArray := self select: [:value | x < value].
newArray := newArray select: [:value | value < y].
^newArray
valuesBetween: a and: b

|y|
y := self select: [:each | (each > a) & (each < b)].
^y
valuesBetween:a and:b

"The method returns an Array that contains all the elements of the receiver that are between the values a and b. If the array is empty returns nil"

| resultArray |
| self isEmpty ifTrue:[^nil].
resultArray := self select: [:eachElement |
  eachElement >= a & (eachElement <= b)
  ifTrue: [true]
  ifFalse: [false]].

^resultArray.
TestBetweenVal

|oArray Arr|

Arr:=#(2 3 4 1 8 9).
oArray:=#(2 3 4 1 8 9)valuesBetween:2 and:8.
self assert: oArray=#(3 4).
squared
A solution

Squared

\(^{self \ collect: \ [:each \ | \ each \ squared]}\)
squares

"This method returns a collection that contains the squares of the values in the receiver collection"

| arrayValues |
arrayValues := Array new: self size.

"Multiply the values/Square them"
arrayValues := self collect: [:a | a * a].
^arrayValues
squaresCollection
  "Square all elements of array"
  ^(self)

  (1 to: self size) do: [ :each |
    self at: each put: (self at: each) squared.
  ].

  ^(self)

  ^(self)

  ^self collect: [:value | value squared].
Issues

squares

| a |
a := self.
^a collect: [:each | each * each]
**Issues**

squareCollection

^((self collect: [:each | (each * each) asFloat])

Tuesday, October 11, 2011
testSquares

| c testC |
c := OrderedCollection with: 5 with: 9.
c := c squares.
testC := OrderedCollection with: 25 with: 81.
self assert: testC = c.
Issues

removeFirst
    | storage |
    (self isEmpty)
        ifTrue: [^nil].
    (size = 1)
        ifTrue: [storage := head getValue. head := nil. tail := nil. size := 0. ^storage.].
    storage := head getValue.
    head := head getNext.
    size := size - 1.
    ^storage.
valuesBetween: a and: b

| myArray newArray |
myArray := self.
newArray := myArray select: [:each | ((myArray indexOf: each) > a) & ((myArray indexOf: each) < b)].
^newArray
squares

[^self collect: [:each | each squared]] on: MessageNotUnderstood
do:
  [:exception |
    Transcript
    show: 'Your collection contains a non-number';
    cr]
Stack
Issues

initialize
"Initialize a newly created instance. This method must answer the receiver."

super initialize.
" *** Edit the following to properly initialize instance variables ***"
myObject := nil.
" *** And replace this comment with additional initialization code *** "
^self
Issues

Smalltalk.Core defineClass: #Stack
  superclass: #{Core.Object}
  indexedType: none
  private: false
  instanceVariableNames: 'oList '
  classInstanceVariableNames: "
  imports: "
  category: "

size
| n_total | val |
val := head.
n_total := 0.
[val ~= nil] whileTrue: [n_total := n_total+1. val := val next.]
^n_total.
Issues

TestPush

| myStack |

myStack := Stack new.

myStack push: 'a'.
myStack push: 'b'.
myStack push: 'c'.
myStack push: 'd'.

self assert: myStack printString = "d"."c"."b"."a"."
Issues

testNode

| newNode  anotherNode |
newNode := Node new.
anotherNode := Node new.
n newNode data: 1.
anotherNode data: 'cat'.
n newNode nextNode: anotherNode.

self deny: newNode data = 'cat'.
s self assert: newNode data = 1.
s self assert: newNode nextNode isNil not.
s self assert: newNode nextNode data = 'cat'.
testClear

| s |
s := Stack new.
s clear.
self assert: s pop = nil
Issues

testDo
| s a |
s := Stack new.
   s push: 3.
s push: 4.
s push: 5.
s push: 6.
a := 0.
s do: [:each | a := a + 5]. "poor test"
Issues

row: rowIndex column: columnIndex

| element |
^element := (nKMatrix at: rowIndex) at: columnIndex
Issues

printOn: aStream

(self currentLinkValue) printOn: Transcript
Issues

removeFirst

| rtnData |

self isEmpty
  ifTrue: [^nil].

rtnData := head data.

size = 1
  ifTrue: [head := tail := nil]
  ifFalse: [head := head next].

size := size - 1.
^rtnData.
moveNext
^nextNode.
Issues

pop

| lastNode |
top = bottom = nil ifTrue: [Transcript show: 'Nothing to pop, first push object in stack'].
lastNode := top.
top := top nextLink.
lastNode nextLink: nil.
^lastNode data
Issues

Smalltalk.Core defineClass: #Stack
    superclass: #{Core.LinkedList}
    indexedType: #none
    private: false
    instanceVariableNames: 'firstNode lastNode '
    classInstanceVariableNames: "
    imports: "
    category: "

Smalltalk defineClass: #Stack
  superclass: #{Smalltalk.Node}
  indexedType: #none
  private: false
  instanceVariableNames: 'head top '
  classInstanceVariableNames: "
  imports: "
  category: "
Issues

Smalltalk.Core defineClass: #Stack
  superclass: #{Core.Object}
  indexedType: #objects
  private: false
  instanceVariableNames: 'data nextlink counter bottom top nextLink ' 
  classInstanceVariableNames: "
  imports: "
  category: "

Tuesday, October 11, 2011
Issues

pop

|x|

self isEmpty
ifTrue: [^nil]
ifFalse:
  [x := top data.
   top := top next.
   ^x]
do: aBlock
    | tmpCSNode tmpObject |
    tmpCSNode := first.
    [tmpCSNode == nil] whileFalse:
        [tmpObject := tmpCSNode data.
        aBlock do: tmpObject.
        tmpCSNode := tmpCSNode next]
Issues

clear
| p q i |
i := 1.
[ i <= self size ]
whileTrue:

| p q i |
i := 1.
[ i <= self size ] whileTrue:

p := q := start.
q := q node.
[q = nil]
ifTrue: [end := nil.]
ifFalse:

[i <= self size] whileTrue:

[p := q := start.]
q := q node.
[q = nil]
ifTrue: [end := nil]
ifFalse:

[[q node ~= nil]
whileTrue:

[q := q node.]
p := p node.
].
p node: nil.
end := p.
].
counter := counter -1.
i := i + 1.
].
clear (after control-o)

| p q i |
i := 1.
[i <= self size] whileTrue:

[p := q := start.]
q := q node.
[q = nil]
ifTrue: [end := nil]
ifFalse:

[[q node ~= nil]
whileTrue:

[q := q node.]
p := p node.
].
p node: nil.
end := p.
].
counter := counter -1.
i := i + 1]
Issues

pop
    | tmpObject |
tmpObject := self removeFirst.
^tmpObject
push:valueNode
"Adds the argument on top of the stack"

| obj |
  obj := Stack new.
  self top isNil
    ifTrue:
      [self top:obj.
       self nodeValue:valueNode]
    ifFalse:
      [obj nextNode: (self top).
       obj nodeValue:valueNode]
push:a

| newNode |
end = nil
ifTrue:[
    newNode := Stack new.
    newNode data: a.
    start := newNode.  "start variable always points to the starting of the stack"
    end := newNode.   "end variable always points to the top of the stack"
    counter :=1.      "counter is an instance variable which counts the number of nodes in the stack"
    ^newNode data.
]

ifFalse:[
    newNode := Stack new.
    newNode data: a.
    end node: newNode.
    end := end node.
    end node: nil.
    counter :=counter +1.
    ^end data.
].
Issues

clear
self isEmpty
  ifTrue: [^nil]
  ifFalse: [top := nil.
    count := 0.]
printOn: aStream
    "comment stating purpose of message"

    | numChars |
    numChars := self size - 1.
aStream nextPut: $(.
self do:
    [:each |
    aStream print: each.
    numChars = 0
    ifFalse:
        [numChars := numChars - 1.
        aStream
            nextPut: $,;
            space]].
    aStream nextPut: $)
Issues

printOn: aStream

| temp myAray i |
i := 1.
temp := start.
myAray := Array new: counter. "myAray has been initialized to the size of the stack"

[temp ~= nil] "while loop puts all nodes of the stack into the myArray"
whileTrue:
    myAray at:i put: (temp data).
i := i+1.
temp := temp node.
].

i := i - 1.
[ i >= 1 ] "this while loop prints the content of myArray in reverse order"
whileTrue:
    aStream print: (myAray at: i).
i := i-1.
].
Issues

clear

firstElement := nil.
^firstElement
push: element

|INode|
|INode:= IStack new.

last=nil

ifTrue:
  [
    numberOfNodes := 0.
  ].

numberOfNodes = 0

"Check wether the number of nodes is equal to zero, if true then add new node and assign last= first"

ifTrue:
  [
    INode data:element.
    last:= first:= INode.
    numberOfNodes :=numberOfNodes +1.  "Increment the node count for future reference"
  ]

ifFalse:  
  "if false then just add node and assign last= node"

  [
    INode data: element.
    INode successiveNode: last.
    last:=INode.
    numberOfNodes :=numberOfNodes +1.
  ].

^INode data.
Issues

removeNodeFromFront

| oldNode |
oldNode := firstNode.

( firstNode == lastNode )
  ifTrue: [
    firstNode := nil.
    lastNode := nil.
  ]
  ifFalse: [
    firstNode := oldNode nextNode.
  ].
oldNode nextNode: nil.
^oldNode currentNodeValue.
TestClear

|o|

o:= (IStack new)push:'a';push:'b';push:'c';clear.
self assert: o='Stack cleared...!'.
Issues

clear

| temporary |
[last ~= nil] whileTrue:
    [temporary := last.
     last := last successiveNode].
"Stack cleared...!"
Issues

StackTest

| aStack |

aStack := Stack new.

aStack push: 1.
aStack push: 2.
aStack push: 3.
aStack size.
aStack pop.
aStack pop.
aStack pop.
aStack pop.
aStack size.
HtmlTable
Issues

asHtml

| currRow currCol | 

currRow := 1.
Transcript clear.
Transcript show: '<table>'.
Transcript cr.

[currRow <= rows]
whileTrue:
[  
  Transcript tab.
  currCol := 1.
  Transcript show: '<tr>'.
  Transcript cr.
  [ currCol <= columns]
  whileTrue:
    [  
      
    ]
]  
]  
]
Issues

asHtml

| i  j  val |

Transcript clear.
Transcript show: '<table>' printString; cr.
i := 1.
[i <= nRows] whileTrue:

[j <= nCols] whileTrue:

Transcript show: '<tr>' printString; cr.
j := 1.
[j <= nCols] whileTrue:

Transcript show: '<td>' printString.
val := (i-1)*nCols + j.
Transcript show: (matrix at: val) printString.
Transcript show: '</td>' printString.
j := j+1.

]. Transcript show: '</tr>' printString; cr. i := i+1.]

Transcript show: '</table>' printString; cr.
^Transcript.
This method returns an HtmTable object

<table>
<thead>
<tr>
<th>h arr</th>
</tr>
</thead>
<tbody>
<tr>
<td>h := HtmTable new.</td>
</tr>
<tr>
<td>h rowLength: r.</td>
</tr>
<tr>
<td>h colLength: c.</td>
</tr>
<tr>
<td>arr := Array new: r * c.</td>
</tr>
<tr>
<td>h dataArray: arr.</td>
</tr>
<tr>
<td>^h</td>
</tr>
</tbody>
</table>
Issues

nbykMatrix

^nbykMatrix
matRow := row.
matrixRow := row.
matCol := col.
matrixColumn := col.
Issues

| htmlTable counter |
htmlTable := HtmlTable new.
htmlTable matRow: numberOfRows matCol: numberOfColumns.
htmlTable := Array new: numberOfRows.
counter :=1.
[counter <= numberOfRows]
whileTrue: [

htmlTable at: counter put: (Array new: numberOfColumns).
counter := counter + 1.
].
^htmlTable.
Issues

removeFirst

size = 0
  ifTrue: [^nil]
  ifFalse:
    [| temp | 
    temp := Node new.
    temp := head.
    head := head next.
    size := size - 1.
    ^temp value]
Issues

size1

^size
initialize: nRows and: nCols

"Initialize a newly created instance. This method must answer the receiver."

"check dimension data types"
( (nRows isKindOf: Integer) & (nCols isKindOf: Integer) )
  ifFalse: [ ^self error: 'Invalid table dimensions: invalid <type>' ].

"check dimension <= 0"
(nRows * nCols <= 0)
  ifTrue: [ ^self error: 'Invalid table dimensions: <size> <= 0' ].

super initialize.

rowSize := nRows.
colSize := nCols.

tableArr := Array new: (nRows * nCols) withAll: ".

^self
Issues

rows: numberOfRows columns: numberOfColumns
"anHtmlTable"

| iterator anHtmlTable |
iterator := 1.
anHtmlTable := HtmlTable new.
anHtmlTable twoDArray: (Array new: numberOfRows). "rew bad idea"
numberOfRows timesRepeat:
    [anHtmlTable twoDArray at: iterator put: (Array new: numberOfColumns).
    iterator := iterator + 1].
^anHtmlTable
Issues

rows: numberOfRows columns: numberOfColumns

|i|

i := 1.
a := Array new: numberOfRows.
rowIndex := numberOfRows.
columnIndex := numberOfColumns.

[i <= numberOfRows] "Constructing a 2*2 matrix using Arrays"
while True:
[
    a at: i put: (Array new: numberOfColumns).
i := i + 1.
].

^self