CS 535 Object-Oriented Programming & Design
Fall Semester, 2008
Doc 16 Assignment 4 Comments
Nov 16 2010

Counter Tests

TestCounter

```
| a b |
Counter reset.
a := Counter new.
a increase.
a increase.
b := Counter new.
a increase.
b increase.
self
    assert: a count = 3;
    assert: b count = 1;
    assert: Counter masterCount = 4
```

```
testWithDeny

    | a b |
    a := Counter new.
    a increase.
    a increase.
    self assert: a count == 2.
    b := Counter new.
    a increase.
    b increase.
    self assert: a count == 3.
    self deny: a count == 3
```

```
testCounter

    | a b |
    a := Counter new.
    Counter masterCountReset.
    a increase.
    a increase.
    a increase.
    a increase.
    a increase.
    a increase.
    b := Counter new.
    b increase.
    b increase.
    b increase.
    b increase.
    b increase.
    self
        assert: a count = 5;
        deny: b count = 1;
        assert: Counter masterCount = 9;
        deny: Counter masterCount = 10
```

Average

```
testAverage
	"comment stating purpose of message"

	| a size averageCalc |
	a := Array new: 10.
	size := 0.
	averageCalc := 0.
	[size < 10] whileTrue:
			[a at: size + 1 put: size.
			size := size + 1].
size := 0.
[size < 10] whileTrue:
			[averageCalc := averageCalc + (a at: size + 1).
			size := size + 1].
averageCalc := averageCalc / a size.
a average == averageCalc
```

```
Array>>calcAverage

	| total average |
	total := 0.
	1 to: self size do: [:i | total := total + (self at: i)].
	average := (total / self size) asInteger.
	^average
```

averageAllElements

```
	| sum |
	sum := 0.
	self size = 0 ifTrue: [^0].
	self do: [:each | sum := sum + (each ifNil: [0] ifNotNil: [each])].
	^sum / (self size)
```

```
arrayAverage: collectedNumbers
	"Adds all the numbers in the array and
	 divided by the number of elements in the array."

	| average sum numberOfElements |
	sum := 0.
	average := 0.
	numberOfElements := collectedNumbers size.
	collectedNumbers do: [:each | sum := sum + each].
	average := sum / numberOfElements.
	^average asFloat
```

```
average

        | count average |
        count := 1.
        average := 0.
        self size < 1
                ifTrue: [self error: 'Out of range']
                ifFalse:
                        [[count > self size] whileFalse:
                                [average := average + (self at: count).
                                count := count + 1].
                        average := average / self size asFloat.
                        ^average]
```

```
average
	"Return the average of all of the elements in the array"

	| sum |
	sum:=0.
	self isEmpty ifTrue:[^nil].
	self do: [:a | sum:=sum+a].
	^sum/(self size).
```

```
average
	"Return the average of all of the elements in the array"

	| sum |
	self isEmpty ifTrue:[^nil].
  sum:=0.
	self do: [:a | sum:=sum+a].
	^sum/(self size).
```

```
average
"Returns the average of all the elements in the array."

	| avg realSize |
	avg := 0.
	realSize := 0.
	1 to: self size
		do:
			[:i |
			(self at: i) ~= nil
				ifTrue:
					[avg := (self at: i) + avg.
					realSize := realSize + 1]].
realSize = 0
	ifTrue: [self error: 'The array is empty.'].
avg := avg / realSize.
^avg
```

```
average
"Returns the average of all the elements in the array."

	| average sum realSize |
	sum := 0.
	realSize := 0.
	1 to: self size
		do:
			[:i |
			(self at: i) ~= nil
				ifTrue:
					[sum := (self at: i) + sum.
					realSize := realSize + 1]].
realSize = 0
	ifTrue: [self error: 'The array is empty.'].
average := sum / realSize.
^average
```

```
average
"Returns the average of all the elements in the array."

	| average sum realSize |
	sum := 0.
	realSize := 0.
	self
		do:
			[:each |
			each ifNotNil
				[sum := each + sum.
				realSize := realSize + 1]].
realSize = 0
	ifTrue: [self error: 'The array is empty.'].
average := sum / realSize.
^average
```

```
average
"Returns the average of all the elements in the array."

	| average sum realSize |
	nonNilElements = self reject: [:each | each isNil].
  nonNilElements isEmpty ifTrue:[self error: 'The array is empty.'].
    sum := 0.
 nonNilElements do: [:each | sum := each + sum].
    average := sum / nonNilElements size.
    ^average
```

```
testArrayTest

	| a b c d |
	a := #(1 2 3 4 5) average.
	self should: [a = 3].
	b := #(-1 -2 -3 -4 -5) average.
	self should: [b = -3].
	c := #(2 4 6 8) average.
	self should: [c = 5].
	d := #(-2 -4 -6 -8) average.
	self should: [d = -5].

	self should: [a > b].
	self should: [b < a].
	self should: [c >= d].
	self should: [d <= c]
```

```
testArray

	| a b|
	a:= Array new: 5.
	a at: 1 put: 1.
	a at: 2 put: 2.
	a at: 3 put: 3.
	a at: 4 put: 4.
	a at: 5 put: 5.

	b:= Array new.
	b:=  #(1 1 1 1 1 1).

	self assert: a average = 3.
	self assert: b average = 1.
```

```
testEmptyValue

	| theTestArr |
	theTestArr := Array new.
	theTestArr := #().
	self should: [theTestArr average / 0] raise: ZeroDivide.
	self assert: theTestArr average = 0.
```

```
testAddFirsts

	| list firstValue |
	list := Linkedlist new.
	list addFirst: 10.
	list addFirst: 20.
	list addFirst: 30.
	list addFirst: 40.
	list addFirst: 50.
	list addFirst: 60.
	firstValue := list at: 5.
	self assert: firstValue = 20
```

```
between: firstCharacter and: secondCharacter

	^self select:
		[:each |
		each >= firstCharacter & (each <= secondCharacter)
			ifTrue: [true]
			ifFalse: [false]]
```

```
asUppercase
"Converts all lowercase characters in a string to uppercase."

	| uppercaseString |
	uppercaseString := self collect: [:a | a].
	1 to: uppercaseString size
		do:
			[:i |
			((uppercaseString at: i) asInteger > 96) & ((uppercaseString at: i) asInteger < 123)
				ifTrue: [uppercaseString at: i put: (Character value: (uppercaseString at: i)
asInteger - 32)]].
	^uppercaseString
```

```
asUpperCase

	| upperCaseString |
	upperCaseString := self collect: [:each | each asUppercase].
	^upperCaseString
```

```
String>>min: aCharacter
	"This will compare characters and result in the smallest one."
	self < aCharacter
		ifTrue: [^self]
		ifFalse: [^aCharacter]
```

```
doConvert

	| start end char tempStream |
	start := 'start' asFilename readStream.
	end := 'end' asFilename appendStream.
	char := start next.
	[char = nil] whileFalse:
			[char ~= (Character value: 44)
				ifTrue: [end nextPut: char]
				ifFalse: [end nextPut: (Character value: 46)].
			tempStream := start upTo: (Character value: 44).
			end nextPutAll: tempStream.
			char := start next.
			char ~= nil ifTrue: [end nextPut: (Character value: 46)]].
end close.
start close
```

Solutions

Counter Test

```
CounterTest>>testIncrease

    | a b |
    self assert: Counter masterCount = 0.
    a := Counter new.
    self assert: a count = 0.
    a increase.
    self
        assert: a count = 1;
        assert: Counter masterCount = 1.
a increase.
b := Counter new.
a increase.
b increase.
self assert: a count = 3.
self assert: b count = 1.
self assert: Counter masterCount = 4
```

LinkedList Test

```
LinkedListTest>>testIndexingBounds

	| list |

	list := DoubleLinkedList new.
	list add: 1.
	self should: [list at: 2] raise: SubscriptOutOfBoundsError.
	self should: [list at: 0] raise: SubscriptOutOfBoundsError
```


I used a student's linked list implementation while writing my tests. This test found a bug in their code. It was not the bug I expected to find.

LinkedList Test

LinkedListTest>>testSize

```
| list |
list := DoubleLinkedList new.
self assert: list size =0.
list add: 1.
self assert: list size = 1.
list add: 1.
self assert: list size = 2.
```

LinkedListTest>>testAddLast

```
| list data |
list := DoubleLinkedList new.
data := (1 to: 10) asArray.
data do: [:each | list addLast: each].
data  keysAndValuesDo: [:index :value |
      self assert: (list at: index) = value]
```

LinkedListTest>>testAddFirst

```
| list data |
list := DoubleLinkedList new.
data := (1 to: 10) asArray.
data do: [:each | list addFirst: each].
data reverse keysAndValuesDo: [:index :value |
      self assert: (list at: index) = value]
```

Array Average Test

```
ArratTests>>testAverage

    self assert: #(1 2 3 4 5) average = 3.
    self assert: #(4) average = 4.
    self should: [#() average] raise: Error
```


This test will break many students implementation of average. There are several different policies one can take when the array is empty. I did not care which one you used. But your tests should have tested for that case.

Array Average

```
Collection>>average

	self assertNotEmpty.
	^self sum/self size
```

```
Collection>>sum

	self assertNotEmpty.
	^self fold: [:sum :each | sum + each]
```

```
Collection>>assertNotEmpty

	self isEmpty ifTrue:[self error: 'Collection is empty'].
```

String

```
String>>asUppercase

	^self collect: [:each | each asUppercase]
```

```
Collection>>between: firstCharacter and: secondCharacter

	^self select:
		[:each | each >= firstCharacter & (each <=
secondCharacter)]
```

String

```
Collection>>min

    self assertNotEmpty.
      ^self inject: self first
            into: [:min :each | each < min ifTrue: [each] ifFalse: [min]]
```

```
Colection>>min

      self assertNotEmpty.
      ^self inject: self anyOne
            into: [:min :each | each < min ifTrue: [each] ifFalse: [min]]
```

String

```
Collection>>min

	self assertNotEmpty.
	^self inject: self anyOne
		into: [:min :each | each < min ifTrue: [each] ifFalse: [min]]
```

```
Collection>>anyOne

	self assertNotEmpty.
	self do: [:each | ^ each]
```


The difference from the previous slide is subtle. We need to inject any element of the collection into the sum. In the previous slide we used "self first", which implies that the first element is some how needed or special. Here "self anyOne" indicates to the reader that any element will do. It also allows the min method to work on collections with out a first element.

Testing min

```
testMin
	| highAscii unicode |
	self should: [" min ] raise: Error.
	self assert: 'z' min = $z.
	self assert: 'tbac' min = $a.

	highAscii := String with: (127 asCharacter) with: (126 asCharacter).
	self assert: highAscii min = (126 asCharacter).

	unicode := String with: (5000 asCharacter) with: (60000 asCharacter).
	self assert: unicode min = (5000 asCharacter).
```

```
String>>words

	| words |
	words := OrderedCollection new.
	self runsFailing: [:each | each isWordSeparator]
		do: [:each | words addLast: each].
	^words
```

```
Character>>isWordSeparator

	self isSeparator ifTrue:[^true].
	(#($. $, $; $? $! $' $") includes: self) ifTrue:[^true].
	^false
```

The file Convert

```
convert: fromFileName to: toFileName
fromCharacter: fromCharacter toCharacter: toCharacter

        | readFile writeFile  |
        readFile := fromFileName asFilename readStream.
        writeFile := toFileName asFilename writeStream.

        [readFile do:
                [:each | | next |
                next := each = fromCharacter
                                ifTrue: [toCharacter]
                                ifFalse: [each].
                writeFile nextPut: next]]
                ensure:
                        [readFile close.
                        writeFile close]
```

Tests

```
setUp

	| out |
	out := 'start' asFilename writeStream.
	out nextPutAll: 'cat,mat sat,bat, rat'.
	out close


testConvert
	self assert: 'start' asFilename contentsOfEntireFile = 'cat,mat sat,bat, rat'.
	FileConvert convert: 'start' to: 'end' fromCharacter: $, toCharacter: $..
	self assert:  'end' asFilename contentsOfEntireFile = 'cat.mat sat.bat. rat'.


tearDown

	'end' asFilename delete
```