Smalltalk Coding Essentials

Collections

VisualWorks provides several classes for operations involving collections of objects. We can use System Browser to explore the collection classes when we need a special kind of collection. The commonly used collections for CS 2340 are:

- Array (Integer index and fastest access)
- OrderedCollection- *Integer index; preserves the order in which elements are added*
- SortedCollection- *Integer index; elements are sorted by user defined algorithm (ascending order is default)*
- LinkedList: Each element points to the next element, for maximum efficiency of dynamic lists.
- Dictionary :Non-integer index; each element consists of a key-value pair for dictionary-like lookups

We create an empty collection, and then add elements to it. All collections respond to the new message.

Array

Array allows you to maintain relative positions of elements, via an integer index. As an example, if a name were to be stored as a collection of three elements—first, middle, and

last names—it would make sense to use an Array because the relative positions of the elements must be preserved.

A ByteArray provides space-efficient storage for bytes. Its elements are restricted to the set of SmallIntegers from 0 to 255. WordArray is for manipulating 16-bit words; its elements can be integers from 0 to 65535

For an Array, which cannot add elements, it is necessary to specify the size of the array. Each element is nil until replaced with another object.

```
| Sports |
Sports := Array new: 4.
Sports at: 1 put: 'Table Tennis';
at: 2 put: 'Cricket';
at: 3 put: 'Soccer';
at: 4 put: 'Football'.
^Sports.
```

You can also create a collection by specifying up to four elements. This approach is typically used to create a small array. Variations of the with: message, for up to four elements, are provided in VisualWorks:

When an array contains only literal elements, such as numbers and strings, you can also create the array using its literal form:

```
| array1 array2 |
array1 := #( 'A' 'B' 'C' D' ).
```

Smalltalk Coding Essentials

```
array2 := #( 1 2 3 4 )
```

is used to indicate that a literal is being created.

Ordered Collection

An OrderedCollection, like an Array, has an integer index and accepts any object as an element. Unlike Array, however, an OrderedCollection permits elements to be added and removed freely. OrderedCollection adds new elements to the end of the collection. You can also position the additional element at the beginning of the collection, before a particular element, or before a particular index. It is frequently used as a stack (the last element in is the first one removed- LIFO) or a queue (FIFO-first in, first out).

```
| ordCn |
ordCn := OrderedCollection new.
ordCn add: 'CS 2340';
add: 'CS 1332;
add: 'CS 1050x';
add: 'PSYC 1501 .
add: ‘Math 1502’.
^ordCn.
```

Note that add: returns the new element. Consequently, you do not want to cascade the add: messages directly from the new message.

By Abhishek Rai

Smalltalk Coding Essentials

Sorted Collection

When elements are not added in the desired order, sorting is required. SortedCollection provides that extra capability. By default, elements are sorted in ascending order. You can override this default by specifying an alternative sort algorithm enclosed in a block. For example,

The expression: SortedCollection sortBlock: [:x :y | x >= y] creates a new collection whose elements will be sorted in descending order. The block is given two elements to compare, and is expected to answer true when the first element should precede the second element.

Sample use of sorted Collections:

```
changeBlock: policyVal
        policyVal = 123
                ifTrue: [lots sortBlock: [:a :b | a lotItemPrice value >=b lotItemPrice value]
```

Dictionary

The Dictionary class, instead of imposing an integer index on each element, permits any object to be the external key. For example, an element might consist of the word 'object' with the associated definition 'something solid that can be seen or touched'. Thus, each element in a Dictionary is typically an instance of Association, which is a key-value pair.

By Abhishek Rai

Smalltalk Coding Essentials

To add an element to a Dictionary, send an at:put: message to the dictionary. The first argument is the lookup key (typically but not necessarily a Symbol). The second argument is the object to be associated with the key.

```
| aussie |
aussie := Dictionary new.
aussie at: #Captain put: 'Ricky Ponting';
at: #Member1 put: 'Matthew Hayden';
at: #Member2 put: 'Adam Gilchrist';
at: #Member3 put: 'Glenn Mc Grath'.
^aussie
```

Inserting an Element at a Specific Location

Collection classes which preserve order, such as OrderedCollection, support protocol for inserting elements at specific positions. It is sometimes helpful to make this position Explicit. To insert an element at the beginning of an ordered collection, send an addFirst: message, which the new element as the argument.

Add Before/After.

To insert an element before or after a specific element already in the collection, send an add: before: message or add: after: message to the collection. The first argument is the element to be inserted. The second argument is the element relative to which the insertion is to take place.

By Abhishek Rai

Smalltalk Coding Essentials

Add Before Index

To insert an element at a numbered position, send an add: beforeIndex: message to the collection. The first argument is the element to be inserted. The second argument is the index of the element before which the insertion is to take place.

```
| team |
team := OrderedCollection new.
team add: 'Ponting';
addFirst: 'Clarke';
add: 'Hayden' before: 'Clarke';
add: 'Gilchrist' beforeIndex: 2.

^team
```

Expanding an Array (important+tricky)

Although an Array can contain only the number of elements with which it was created, you can expand an array by creating a copy that has a new element appended to it. The copy can then be substituted for the original. To create the copy, send a copyWith: message to the Array. The argument is the object that is to be appended to the end of the new array.

```
| array copy |
array := #( 1 2 3 4 5 6 7 8 9 ).
copy := array copyWith: 10.
```

```
array := copy.

^array
```

Removing Elements

The basic method for removing an object from a collection is to send a remove: message to the collection, with the object to be removed as argument. An empty block is an effective means of taking no action, so the process can continue without an error message or other action.

Removing an Element from an Array

To remove occurrences of an object from an array, we can create a copy of the array, omitting each occurrence of a specified object. Send a copyWithout: message to the Array. The argument is the object to be removed. The copy can then be substituted for the original array. The copyWithout: message works for all ordered collections as well as arrays.

```
| dates workingDays |

dates := #( 1 8 3 4 5 6 7 8 9 ).

workingDays := dates workingDays copyWithout: 8.

dates := workingDays.


^dates
```

Smalltalk Coding Essentials

Testing for Emptiness

To test for emptiness, send an isEmpty message to the collection. The response is true when the collection has no elements and false otherwise. Similarly, you can test whether the collection is not empty by sending a notEmpty message.

Removing an Element or Several Elements

Ordered collections provide several messages for removing a single element at a specified position or a range of elements:

```
| numbers |
numbers :=list new: 25.
1 to: 25 do: [ :i | numbers add: i].
numbers removeFirst. "Removes 1"
numbers removeFirst: 5. "Removes 2 3 4 5 6"
numbers removeLast. "Removes 25"
numbers removeLast: 5. "Removes 20 21 22 23 24"
numbers removeFrom: 8 to: 12. "Removes 14 15 16 17 18"
^numbers
```

Arbitrary collections are sorted by first being converted to an instance of SortedCollection.

asSortedCollection Returns a new collection as an instance of SortedCollection, with the collection's elements in ascending order.

reverse Returns a new collection of the same kind, but with the elements in reversed order.

By Abhishek Rai

Smalltalk Coding Essentials

The SUnit package

Unit tests are of two types: Whitebox and Blackbox. SUnit is a framework to write and perform test cases in Smalltalk. SUnit tests are included in Smalltalk image SUnit allows one to write the tests and check results in Smalltalk. We Create a test class inherited from TestCase.

```
Smalltalk defineClass: #XyzTest   superclass: #{XProgramming.SUnit.TestCase}
```

For the class under test, we override the setUp and tearDown methods to create any test fixtures. Then we move on to create test methods to test the class functionality. You can run tests in gui with TestRunner open. If the bar is green, the tests have passed. If the bar is red, it means one or more tests have failed. Within each test method, we use one of the four expressions:

- self assert: aBooleanExpression expected to be true
- self deny: aBooleanExpression expected to be false
- self should:[aBlock expected to be true]
- self shouldnt: [aBlock expected to be false]

Sample usage:

setup

"variables used are instance variables declared on the class definition page**"**

By Abhishek Rai

```
  invento := Inventory new.
  i1 := Item new.
   i1 name: 'Video Game'.
   i1 id:'89X999'.
l1:=Lot new.
iNR:= Money new.
iNR conversionRate: 51.3.
iNR currencyName:'Indian Rupee'.
l1 lotItemPrice: iNR.
l1 numOfItems: 100.
l1 prototype:i1.
  i2 :=  Item new.
   i2 name: 'TV'.
   i2 id:'69X89Y'.
l2:= Lot new.
won:= Money new.
won conversionRate: 1000.
 won currencyName:'Won'.
l2 lotItemPrice: won.
```

testBestLotForItem

```
        | l3 GG|
        l3:=  Lot new.
```

Smalltalk Coding Essentials

```
GG:= Money new.

GG conversionRate: 20.

GG currencyName:'Good Game'.

l3 lotItemPrice: GG.

l3 numOfItems: 100.

l3 prototype:i1.

invento pushLot: l3.

invento pushLot: l1.

invento pushLot: l2.

self assert: ((invento bestLotForItem: i1) = l3).
```

----------*----------*----------

By Abhishek Rai