Collections In Squeak

Technion - Israel Institute of Technology

Last Update : Guy Suday, July 2018
• Everything is an object
• All actions are produced by passing messages
• Class and Instance
• Message and Method
A block has the general form:
\[
[
:parameters | | args | statements
]
\]

Executes only when received a message value
- Block Definition: [ Transcript show: 'hello' ]
- Block Execution: [ Transcript show: 'hello' ] value

Executes in the context in which it was defined

An explicit return causes a return from the block’s creation point
## Squeak VS C++

<table>
<thead>
<tr>
<th>Squeak</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := \text{self } f.$</td>
<td>$x = \text{this-&gt;f();}$</td>
</tr>
<tr>
<td>$x := \text{self addKey: 'K' Value: 5}$</td>
<td>$x = \text{this-&gt;addKeyValue('K',5);}$</td>
</tr>
<tr>
<td>✗ No global methods</td>
<td></td>
</tr>
<tr>
<td>✗ Messages are sent to a specified object</td>
<td></td>
</tr>
<tr>
<td>$x := 1+2$</td>
<td>✗ $x = 1+2;$</td>
</tr>
<tr>
<td>Control structures are regular messages</td>
<td>✓ $x = 1.\text{Plus}(2)$</td>
</tr>
<tr>
<td>Control structures have special syntax</td>
<td></td>
</tr>
<tr>
<td>Squeak</td>
<td>C++</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>No Garbage Collection</td>
</tr>
<tr>
<td>Dynamic Typing</td>
<td>Static Typing</td>
</tr>
<tr>
<td>fields are <em>object</em> private</td>
<td>fields are <em>class</em> private</td>
</tr>
<tr>
<td>Single Inheritance</td>
<td>Single or Multiple Inheritance</td>
</tr>
</tbody>
</table>
Today we will see:

• The Squeak collection hierarchy.
• Some collection operations.
• Working with collections.
• For-loops.
What are Collections?

• Collections provide the means for managing and manipulating *groups of objects*. Common collections include:
  - **Array**: fixed-size ordered group.
  - **Linked List**: dynamic-size ordered group. Insertions and removals define the order.
  - **Set**: an unordered group of unique objects.
  - **Dictionary**: like *set*, but each element is a key-value pair. Elements are accessed by their keys.
  - **String**: can be considered to be a special form of Array, where the elements must be characters.
  - **Heaps** and other forms of sorted collections, which require content that can be compared to itself.
Classification of Collections

- Collections differ from each other by their **space complexity**, the **operations** they support and the **time complexity** for each of these operations.
- Disregarding complexity, we can assign attributes to collections based on the operations they support:
  1. Can we **access items in the order** they were inserted? Ordered (List, Array) / Unordered (Set)
  2. Can we **change the collection size**? Dynamic (LinkedList) / Fixed (Array)
  3. Can we **change the collection content**? Mutable (Array) / Immutable (?)
  4. Can the collection **hold multiple copies of the same object**? Yes (Bag) / No (Set)
  5. What kind of objects can the collection hold? Must they all be of the same type (**homogeneous**) or not (**heterogeneous**)?
  6. **How do we access the items** in the collection? By value, by index, by key?
- There are even more attributes, outside our scope...
Key Collection Classes in Squeak.

This is only a partial view, Squeak class libraries contain over 90 collection classes!
Some Collection Methods

Are defined, redefined, optimized or forbidden (!) in subclasses

<table>
<thead>
<tr>
<th>Category</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing</td>
<td>size, capacity, at: anIndex, at: anIndex put: anElement</td>
</tr>
<tr>
<td>Testing</td>
<td>isEmpty, includes: anElement, contains: aBlock, occurrencesOf: anElement</td>
</tr>
<tr>
<td>Adding</td>
<td>add: anElement, addAll: aCollection</td>
</tr>
<tr>
<td>Removing</td>
<td>remove: anElement, remove: anElement ifAbsent: aBlock, removeAll: aCollection</td>
</tr>
<tr>
<td>Enumerating</td>
<td>do: aBlock, collect: aBlock, select: aBlock, reject: aBlock, detect: aBlock, detect: aBlock ifNone: aNoneBlock, inject: aValue into: aBinaryBlock</td>
</tr>
<tr>
<td>Converting</td>
<td>asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: aBlock</td>
</tr>
<tr>
<td>Creation</td>
<td>with: anElement, with:with:, with:with:with:, with:with:with:with:, withAll: aCollection</td>
</tr>
</tbody>
</table>
So how do I...?

- To check how to create a collection / add to a collection / remove from a collection / access an item in a collection etc. just open the class in the System Browser and check its methods.
- The methods are conveniently organized in categories (for example, there’s a category called “adding”).
- If you can’t find the method you expected there, check in its super class!
- Many operations are shared between collections (specifically, many are inherited from Collection).
- We’ll go over the methods for converting and for enumerating.
Example: addAll:

```
addAll: aCollection

"Include all the elements of aCollection as the receiver's elements. Answer aCollection. Actually, any object responding to #do: can be used as argument."

aCollection do: [:each | self add: each].

^ aCollection
```
• **Send** asSet, asBag, asSortedCollection etc. to convert between kinds of collections

```
arr := {1. 2. 3. 5. 3}
Transcript show: arr; cr.                  #(1 2 3 5 3)
newSet := arr asSet.
newBag := arr asBag.

Transcript show: newSet; cr.               a Set(1 2 3 5)
Transcript show: newBag; cr.               A Bag(1 2 3 3 5)
```
• Use factory methods to build new kinds of collections from old kinds.
  – Build Dictionary from Array:
    \[ \text{dict} := \text{Dictionary \ newFrom: \ {1->a. \ 2->b. \ 3->c} } \]
  • ‘->’ operator creates a tuple with \(<\text{key}, \text{value}>\)

• **Send** \text{keys}, \text{values} to extract collections from dictionaries

\[ \text{keyArr} := \text{dict \ keys.} \]
\[ \text{valArr} := \text{dict \ values.} \]
\[ \text{Transcript \ show: \ keyArr; \ cr. \quad \#(1 \ 2 \ 3) } \]
\[ \text{Transcript \ show: \ valArr; \ cr. \quad \#(a \ b \ c) } \]
**Basic Enumeration**

- **The method** `do: aBlock` **is the most basic enumerating mechanism.** The block must be a single-parameter block for invoking it with the current value.

<table>
<thead>
<tr>
<th>C++</th>
<th>Smalltalk / Squeak</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;int&gt; c = {2,4,6,8};</code>&lt;br&gt;<code>int sum = 0;</code>&lt;br&gt;<code>for (int i = 0 ; i &lt; c.size() ; i++)</code>&lt;br&gt;<code>sum += c.get(i);</code>&lt;br&gt;<code>// c’s type must support the length()</code>&lt;br&gt;<code>// and get(int) operators</code></td>
<td><code>c := {2. 4. 6. 8}.</code>&lt;br&gt;<code>sum := 0.</code>&lt;br&gt;`c do: [ :x</td>
</tr>
</tbody>
</table>
Advanced Enumerations

There are many additional, more advanced and very useful enumeration methods. Some of them are:

• **collect:*, like ML’s map:
  
  $(1 \ 2 \ 3) \ \text{collect:} \ [\! :x\! |x*x\!] \rightarrow \ (1 \ 4 \ 9)$
  
  —The new collection will be of the same type of the old one.

• **select:*, like ML’s filter (it also has an opposite, reject:):*
  
  $(1 \ 2 \ 3) \ \text{select:} \ [\! :x\! |(x \ \text{rem:} \ 2) = 1\!] \rightarrow \ (1 \ 3)$
  
  —The new collection will be of the same type of the old one.

• **inject:into:*, which returns a scalar and is like folding in ML:
  
  \{' see'.' \ you'\} \ \text{inject:} \ \text{‘I’ into:} \ [\! :a: b|a,b\!] \rightarrow \ \text{‘I see you'}$

  $(1 \ 2 \ 3) \ \text{inject:} \ 1 \ \text{into:} \ [\! :\text{sum: next}|\text{sum} + \text{next}\!] \rightarrow \ 7$

  —Is this a left fold or a right fold?
Advanced Enumerations – Cont’d

• Confused? Check the code! **All the advanced enumerations are ultimately based on do:** and are very simple to read.

• For example, this is the implementation of `occurancesOf`:

```smalltalk
occurrencesOf: anObject
  ^self inject: 0
  into: [ :x :y | ( y = anObject )
    ifTrue: [ x + 1 ]
    ifFalse: [ x ] ]
```
For Loops

• Iterating over a collection is nice, but what if we really do want to iterate according to index, such as with C++ for loops?

• Smalltalk does not offer a straightforward way to do that. We can use the `whileTrue:` method of blocks, but a more convenient solution is to use the Interval collection class.

• An interval is basically an array of numbers, but its advantage is that it’s very easy to create one from numbers.
Interval Creation and Usage

• To create an interval, the simplest way is to use the to: or to:by: methods of Number:
  – 0 to: 10 instead of #(0 1 2 3 4 5 6 7 8 9 10)
  – 0 to: 11 by: 2 instead of #(0 2 4 6 8 10)

<table>
<thead>
<tr>
<th>C++</th>
<th>Squeak</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (int i = 0 ; i &lt; 10 ; i += 2)</td>
<td>(1 to: 10 by: 2) do: [:i</td>
</tr>
<tr>
<td>buffer[i] = 0;</td>
<td></td>
</tr>
</tbody>
</table>

• Number also has the methods to:do: and to:by:do: which make things even shorter:

<table>
<thead>
<tr>
<th>Squeak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to: 10 by: 2 do: [:i</td>
</tr>
</tbody>
</table>
Summary of Main Points

• Main kinds of Collections:
  – SequenceableCollections
  – Dictionary
  – Sets and Bags
• You can convert most collections to another kind of collection.
• Sorting using converting messages.
• Many methods used for iteration: do, collect.