Collection classes

One of the most important hierarchies in the Smalltalk class library is the Collection classes.

Objects of these classes are containers in which other objects can be stored.

Here is the complete collection hierarchy in IBM Smalltalk. Instance-variable names are given in parentheses.

Object  
  Collection  
  Bag  
  KeyedCollection  
  Dictionary  
  LookupTable  
  FixedSizeLookupTable  
  IdentityDictionary  
  SequenceableCollection  
  AdditiveSequenceableCollection  
  OrderedCollection  
  SortedCollection  
  ArrayedCollection  
  Array  
  ByteArray  
  EsString  
  DBString  
  String  
  Symbol  
  Interval  
  Set

Classes whose name begins with Es or Abt are generally private (i.e. have no documented API). They have not been listed above, except when they have a public subclass.

We will consider some of the more important collection classes.

Class Collection is an abstract class, but its methods are inherited by all collections.
There are methods for converting from one kind of collection to another:

- \texttt{asBag}
- \texttt{asSortedCollection:}

There are methods to test the contents:

- \texttt{includes:}
- \texttt{isEmpty}

There are two methods for adding elements to a collection, \texttt{add:} and \texttt{addAll:}

How do you suppose that \texttt{add:} is implemented?

\texttt{addAll:} is implemented like this:

\begin{verbatim}
addAll: aCollection
"Include all the elements of aCollection as the receiver's elements. Answer aCollection."

aCollection do: [:each | self add: each].
^aCollection
\end{verbatim}

\textbf{Class Bag}

A “bag” is an unordered collection that may contain duplicates.

Also, \texttt{Bags} have no “external keys” (like array indices) that can be used to identify the elements.

Items may be added to a \texttt{Bag} with the \texttt{add:} method, or the \texttt{add: withOccurrences:} method.

\begin{verbatim}
bookBag add: 'S, O & D'.
bookBag add: 'notebook'
  withOccurrences: 3.
bookBag sortedCounts
\end{verbatim}
The occurrencesOf: message returns the number of occurrences of its argument in the receiver.

The sortedCounts method returns a list (actually, a SortedCollection) of ordered pairs (actually, Associations) of

    (item, number of occurrences)

in decreasing order of frequency.

**Class set**

A “set” is an unordered collection with no external keys.

What is the difference between a Set and a Bag?

How does add: for a set differ from add: for a bag?

**Abstract class SequenceableCollection**: A sequenceable collection is a collection whose elements are—

- ordered, and
- externally named by integer indices.

Many important classes are subclasses of SequenceableCollection.

Subclasses are distinguished by—

- how the ordering is created, and
- whether the elements are restricted kinds of objects.

The subclasses are—

* ordered, determined externally
  * OrderedCollection elements ordered by user adding and removing
  * LinkedList uses a chain of elements that must be links*
ArrayedCollection elements accessible by integers as external keys.

ordered, determined internally

Interval elements must be numbers

Let us consider some Sequenceable-Collections, beginning with the most familiar.

**Class Array**

An array is an ordered collection with external keys.

What are these external keys?

```
    at:
    at: put:
```

Few methods are implemented in class Array itself.

Mainly, these methods include those for printing, and a hash function.

**Class ByteArray:** Like an array, but its contents are bytes.

What is the advantage of having a separate class for ByteArrays?

**Class OrderedCollection:** Ordered by the sequence in which elements are added and removed.

OrderedCollections can act as deques:

```
    addFirst:  Add the argument to the beginning of the receiver. Answer the argument.
    addLast:   Add the argument to the end of the receiver.
    removeFirst Remove the first element of the receiver and answer it. If receiver is empty, report an error.
```
removeLast  Remove the last element of the receiver.

Which of these methods would you use if you wanted to implement a stack?

A queue?

There are also accessing methods

    first
    last

and several other methods whose meaning is nearly self-explanatory:

    add: after:
    add: before:
    addAllFirst:

**Class SortedCollection**

In an OrderedCollection, the elements can appear in any order.

SortedCollection is a subclass of OrderedCollection in which the elements are kept sorted according to some rule.

A SortedCollection is sorted in “increasing” order unless otherwise specified:

    SortedCollection new

creates a collection sorted in increasing order.

The order in which a collection is to be sorted can be specified in a sortBlock.

Here is how to create a SortedCollection sorted in decreasing order:

    SortedCollection
      sortBlock: [:a :b | a > b]

Elements must be added to the collection with add: (not addLast:, at:put:, etc.).
Class Dictionary

Class Dictionary is an unordered collection that stores (key, value) pairs (called Associations).

It is like an array where the “subscripts” need not be integers, nor even numbers.

A (key, value) pair is stored in a dictionary like this:

```plaintext
myDict at: 'car' put: 'Corvette'
```

Given a key, one can look up a value—

```plaintext
myDict at: key ifAbsent: aBlock
```

The value of key is looked up; if key is not in the dictionary, the result of evaluating aBlock is returned.

Given a value, one can look up a key—

```plaintext
myDict keyAtValue: 'Corvette'
```

The key associated with a particular value may not be unique. Why?

If the key is not found, nil is returned.

There is a message testing whether a key is in the dictionary:

```plaintext
includesKey:
```

Class IdentityDictionary

[Liu §6.5] provides objects that are like Dictionaries, except that (key, value) entries are treated as the same if they have identical keys rather than equal keys.

For example, if we use Strings as keys, should we be using a Dictionary or an IdentityDictionary?
What would happen if we inserted the first entry, then the second entry into a Dictionary?

Into an IdentityDictionary?

So, what kind of objects can be used as keys in an IdentityDictionary?

Symbols are guaranteed to be unique. There’s only one instance of a particular symbol.

That is, if two symbols are spelled the same way, they are the same:

```plaintext
'a string' asSymbol
== 'a string' asSymbol.
```

For a symbol, = and == are the same.

One common use of symbols: As keys to dictionaries.

**Manipulating Collections**

**Overriding equality**

[Liu §6.8] As we have seen, classes are allowed to define their own = (equality) methods.

However, a class that overrides equality doesn’t assure that items with equal keys will map to the same Dictionary entry!

That is because the hash function determines where in a Dictionary an element maps.
To cause equal keys to map to the same directory entry, a class must override \texttt{hash} as well as \texttt{=}. 

For example, if \texttt{=} for \texttt{Taxpayers} compares Social Security numbers, then \texttt{hash} for \texttt{Taxpayers} should hash Social Security numbers.

\textbf{Copying objects}

[Liu §6.11] Most objects are made up of other objects. There are two different ways of copying them.

- Create new \texttt{pointers} to each of the subobjects (shallow copy).

- Copy each of the subobjects (deep copy).

Suppose the value \texttt{w} is assigned to the first field of the copy.

Does the value of the first field of the original object change—

- if the copy was a deep copy?
• if the copy was a shallow copy?

Thus, what way(s) can the value of the first field of both the original and its copy be changed at the same time?

Consider this example where personnel records are cloned via shallow copying—

![Diagram of personnel records]

To change the insurance limit, we could give each employee a new value for it, or change the value in the shared “insurance limit” object.

**Sequencing over collections**

There are many different kinds of collections. To bring some order to all this variety, a common set of operations has been defined for accessing their elements.

In all these operations, a block is used to operate on the individual elements.

In Lecture 2, we have already seen `do`, `reverseDo`, `collect`, `select`, and `detect`.

- `aCollection inject: initialValue into: binaryBlock`.

  Evaluates `binaryBlock` with `initialValue` and the first element, then again with its result and the second element, then its result and the third, and so on. Returns the final result.
aDictionary := (Dictionary new)
    at: #Key1 put: #Value1;
    at: #Key2 put: #Value2;
    yourself.

aDictionary inject: '' into:
    [:string:value|string,value]

• **aCollection1 with: aCollection2 do:**
  binaryBlock

  The collections must be the same size. Evaluates the
  binaryBlock with successive elements from each of the
  two collections as the parameters. Returns the receiver.

  result := OrderedCollection new.
  #(1 2 3) with: #(1 2 3) do: [:v1 :v2| result add: v1+v2].

• **aCollection1 findFirst: aBlock.**

  Returns the key of the first element for which aBlock
  evaluates to true; returns 0 if none exists. Applies only to
  the integer-keyed collections.

  #(1 2 3) findFirst: [:element|element even]  2

• **aCollection1 findLast: aBlock.**

  Returns the key of the last element for which aBlock
  evaluates to true; returns 0 if none exists. Applies only to
  the integer-keyed collections.

  #(1 2 3) findLast: [:element|element odd]  3

*inject:into:* can be used to simplify loops that perform
operations on successive elements. Instead of writing

```
|product|
product := 1.
```
(1 to: 6) do: [:element | product := product * element]

you can write

(1 to: 6) inject: 1 into:
[ [:product: element | product * element] ]

**Stream classes**

All stream classes have the property that they fetch (read) or store (write) elements in order, one element at a time.

Here is the hierarchy for stream classes:

```
Object
  Stream
    PositionableStream
      ReadStream
      WriteStream
        CodeStream
        ReadWriteStream
```

Some methods for reading streams include—

```
next
  Answer the next object accessible by the receiver

next:
  Answer the next n elements accessible by the receiver, where n is the argument.

nextMatchFor:
  Access the next element and test whether it is equal to the argument.
```

Some methods for writing streams include—

```
nextPut:
  Place the argument as the next element accessible by the receiver. Answer the argument.

nextPutAll:
  Takes a collection as an argument, and stores its elements as the next element accessible by the receiver.
```

There are three methods for interrogating streams:
size Returns the number of elements in the stream.

atEnd Answer whether the receiver is positioned at the end.

isEmpty Answer whether the receiver is positioned at the beginning (misnamed).

There are several methods for positioning streams:

reset Repositions the stream to indicate that no objects have been read or written.

setToEnd Repositions the stream to indicate that all objects have been read or written.

position Answers the position of the stream. "0" means no objects have been read or written.

position: Sets the position to the specified value, providing it is within range.

Example: | play |
play := ReadStream on: #(A Funny Thing Happened on the to the Forum).
play size.
play isEmpty.
4 timesRepeat: [play next].
play next.
play nextMatchFor: #the.
play reset.
play next.

Summary:

- The Collection classes share a common protocol base.
  - A Bag is the simplest kind of collection, where elements can be duplicated.
  - A Set is like a Bag, but duplicates are not allowed.
  - An Array behaves in a fashion similar to other programming languages.
  - An OrderedCollection is like an array, except that elements can be added at the ends or in the middle.
• A SortedCollection is like an OrderedCollection, except that it is kept sorted according to some rule.

• A Dictionary is used to store (key, value) pairs.

• To use Collections effectively, we must know the implications of overriding equality and copying objects.

• Several methods have been defined to sequence over collections. The same method works for many different kinds of collections.

• A Stream provides an easy way of iterating over a collection.