

21. Details of collection classes

In this section, we describe the details of the collection classes `Set` and `OrderedList`. There are other collection classes in the `qBeta` library, but they are not described here. We do not provide details of class `Array`, since `Array` is a built-in mechanism of `qBeta`.

Class Collection

Class `Collection` is a general superclass of all collection classes in the `qBeta` library.

```
class Collection(ElmType:< Object):  
  insert(e: ref ElmType):<  
    inner(insert)  
  has(e: ref ElmType) -> B: var boolean:<  
    inner(has)  
  isEmpty -> b: var boolean:<  
    inner(has)  
  remove(e: ref ElmType):<  
    inner(remove)  
  size -> v: var integer:<  
    inner(size)  
  clear:<  
    inner(clear)  
  inner(Collection)
```

Class `Collection` has the following attributes:

- A virtual class parameter, `ElmType`, which specifies the type of the elements that may be inserted into the `Collection`.
- A method `insert` for inserting an element in the `Collection`.
- A method `has` that returns true if the parameter `e` is member of the `Collection`, false otherwise.
- A method `isEmpty` that returns true if there are no members in the `Collection`, false otherwise.
- A method `remove` for removing an element `e` for the `Collection`.
- A method `size` that returns the number of elements in the `Collection`.
- A method `clear`, that remove all members of the `Collection`.

Class Set

Class `Set` is a subclass of `Collection`.

It has a local class `link` that is used to represent the elements of the list and a local reference, and a variable `head` that refers to the first element of the list:

```
class Set: Collection  
  insert::<  
    head:= link(e,head)
```

```

...
class link(e: ref ElmType, next: ref link):
    ...
    head: ref link

```

The representation of a `Set` objects is a so-called *linked list*. Each element in the list is a `link` object. Each `link` object contains a reference, `e` being an element in the `Set` and a reference `next` referring to the next element in the linked list.

The variable `head` refers to the first object in the linked list.

The virtual binding of `insert` creates a new `Link` object where the first parameter is `e`, the element to be inserted and the second parameter is the reference `head` referring to the current list representing the `Set`.

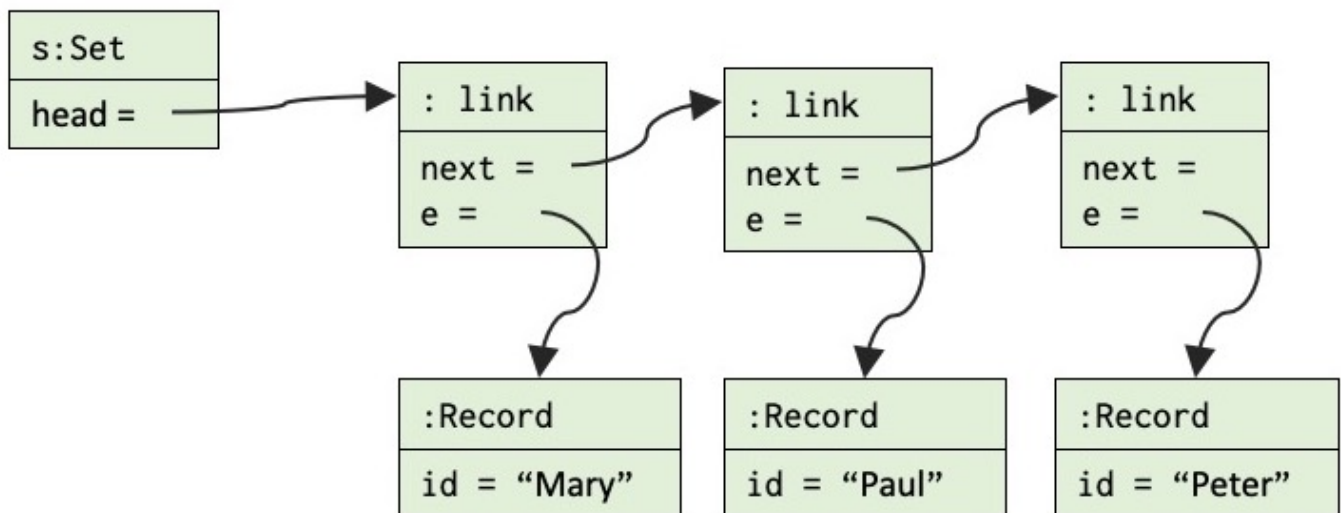
Consider the following example:

```

s: obj Set(Record)
class Record(id: var String):
    ...
s.insert(Record("Peter"))
s.insert(Record("Paul"))
s.insert(Record("Mary"))

```

After execution of the above code, the following snapshot shows the situation of the `Set s` and its internal representation:



A `Set` also has an iterator `scan` that executes an inner for each element of the `Set`:

```

class Set: Collection
    ...
    scan:
        current: ref ElmType
        nxt: ref link
        scanTail:
            ...
            nxt := head

```

```

loop: do
  if (nxt /= none) :then
    current:= nxt.e
    inner(scan)
    nxt:= nxt.next
    restart(loop)

```

As can be seen, the loop object of scan goes through the linked list starting represented by link objects starting with head and for each link object it assigns its e reference to current and executes an inner.

Scan has a local control pattern scanTail, which at a given point during a scan iterates over the rest of the elements in the Set.

The full implementation of Set is shown below:

```

class Set: Collection
  insert::<
    head:= link(e,head)
  has::<
    b := false
    scan
    if (current == e) :then
      b := true
      leave(has)
  isEmpty -> b: var Boolean::<
    b := head == none
  remove::<
    ...
  size::<
    scan
    v := v + 1
  clear::<
    head := none
  scan:
    current: ref ElmType
    nxt: ref link
    nxt := head
    scanTail:
      current: ref ElmType
      nextt: ref link
      nextt := this(scan).nxt.next
      loop: do
        if (nextt /= none) :then
          current := nextt.e
          inner(scanTail)
          nextt := nextt.next
          restart(loop)
    loop: do
      if (nxt /= none) :then
        current:= nxt.e
        inner(scan)

```

```

        nxt:= nxt.next
        restart(loop)
    :else
        nxt:= nxt
%private
class link(e: ref ElmType next: ref link):
    ...
    head: ref link
    inner(Set)

```

Note that class `Link` and `head` are `%private` attributes of `OrderedList`.

Class OrderedList

The implementation of class `Orderedlist` is similar the implementation of class `Set`. It has a class `link` and a reference `head` to the first `link` object holding the members of an `OrderedList`.

In addition it has a reference variable `tail` referring to the last member of the linked list. The reason is a new member is inserted at the end of the linked list and not at the beginning as for class `Set`.

```

class OrderedList: Collection
    :::
    insert::
        if (head == none) :then
            head := Link(e):next(none)
            tail := head
        :else
            t: ref Link:next
            t := link(e):next(none)
            tail.next := t
            tail := t
    :::
    link(e: ref ElmType):next(next: b Link:next):
        ...
    head: ref link:next
    tail: ref Link:next

```

When the first element is inserted (`head == none`) in an `OrderedList`, a `link` object is generated and `head` and `tail` are set to refer to this object.

When additional members are inserted (`head != none`), a `link` object is generated and `tail.next` and `tail` is set to refer to this object.

Below we show a modified version of the above example using `Set`, but using an `OrderedList`.

```

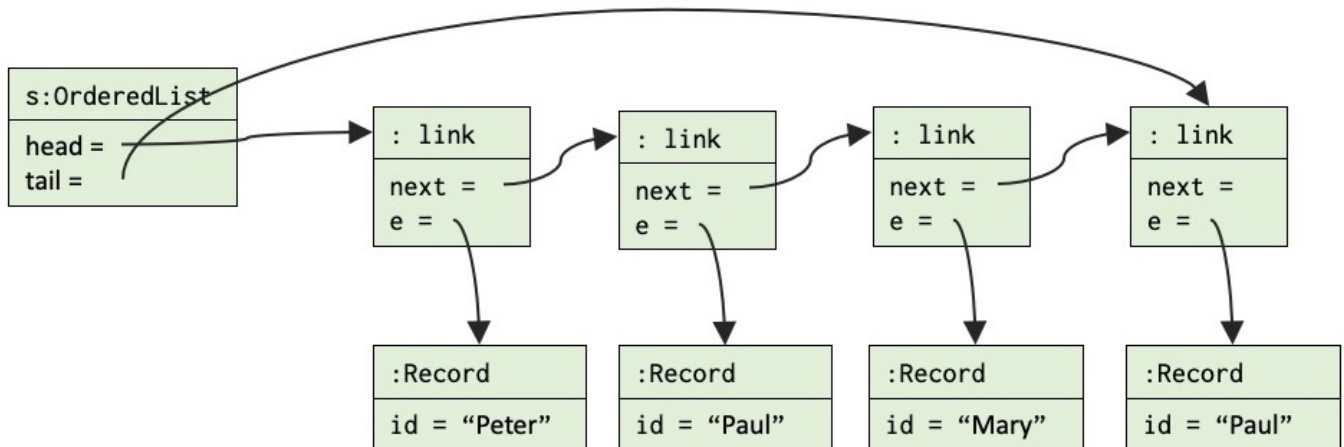
s: obj OrderedList(Record)
class Record(id: var String):
    ...
r: ref Record
s.insert(Record("Peter"))
r := Record("Paul")

```

```
s.insert(r)
s.insert(Record("Mary"))
s.insert(r)
```

We insert the object `Record("Paul")` as referred to by `r` twice to illustrate that an `OrderedList` may have the same element more than once, which is not the case for `Set`.

After execution of the above code, the following snapshot shows the situation of the `OrderedList s` and its internal representation:



As for `Set`, `OrderedList` also has a `scan` method.

The full implementation of `OrderedList` is shown below:

```
OrderedList: obj
class OrderedList: Collection
  insert::
    if (head == none) :then
      head := Link(e):next(none)
      tail := head
    :else
      t: ref Link:next
      t := link(e):next(none)
      tail.next := t
      tail := t
  has::<
    b := false
    scan
      if (current == e) :then
        b := true
        leave(has)
  isEmpty::<
    b := head == none
  remove::<
    ...
  size::<
    scan
      v := v + 1
```

```

clear::<
    head := none
    tail := none
scan:
    current: ref ElmType
    nxt : ref Link:next
    nxt := head
    loop: do
        if (nxt /= none) :then
            current := nxt.e
            inner(scan)
            nxt := nxt.next
            restart(loop)
%private
link(E: ref ElmType):next(next: Link:next):
    ...
head: ref link:next
tail: ref Link:next

```