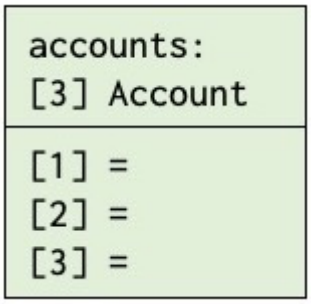
4.2 Array and for-loop

A customer may have more than one account. In this section, we extend class `Customer` to keep track of the accounts of a given customer.

We may use class `Set` for this purpose, but we use this example to introduce `Array`, which is another common collection object. An `Array`-object is an indexed sequence of references to objects where each reference in the sequence may be denoted by an integer index.



Accounts Array

Class `Array` is defined as follows:

```
class Array(range: var integer, class ElmType:< Object):
   put(R: ref ElmType):at[index: var integer]:
      ...
   get(index: var integer) -> R: ref ElmType:
      ...
```

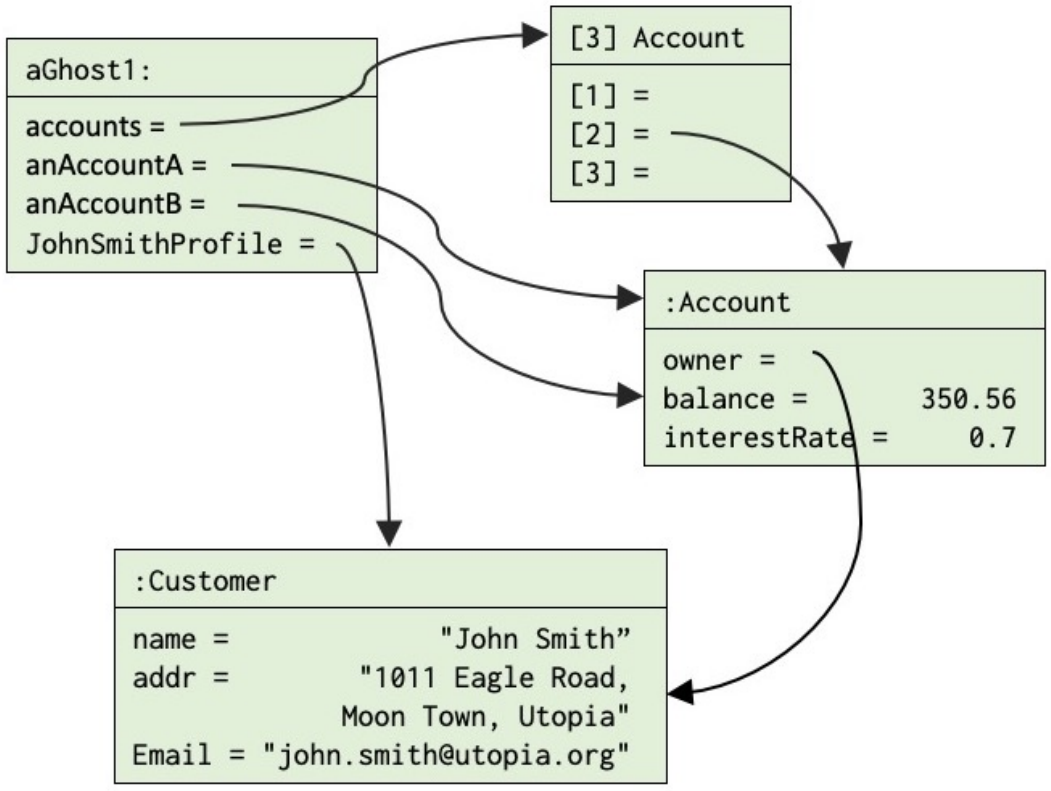
- The first parameter, `range` of `Array` is the number of elements in the `Array`.
- The second paremeter, `ElmType`, is the type of the elements in the `Array` – similar to `ElmType` for class `Set`.
- It has a method `put:at` with two parameters `R` and `index`. This method stores the reference at the position given by the value of `index`.
- It has a method `get`, which returns the reference stored at the position given by the parameter `index`.
- As can be seen, `put:at` and `get` use another syntax for defining parameters than the one we have seen in previous examples using standard brackets and comma to separate the parameters. This syntax is explained in section .

The following example shows how we may use an `Array`-object, `accounts`. Below we add `accounts` as an attribute of class `Customer`, but first we show how to use `Array` by means of a ghost object.

```
aGhost1: obj
   accounts: obj Array(3, Account)
   anAccountA, anAccountB : ref Account
   JohnSmithProfile: obj Customer("John Smith")
   anAccountA := Account(JohnSmithProfile)
   accounts.put(anAccountA):at[2]
   anAccountB := accounts.get[2]
```

- The first item in the example above declares an `Array`-object with 3 elements (the first parameter) of type `Account` (the second parameter).
- The statement `accounts.put(anAccountA):at[2]` inserts the `Account` referred to by `anAccountA` as the second element in the array.
- The statement `anAccountB := accounts.get[2]` assigns the second element in the array to the reference `anAccountB`.

The situation at the end of `aGhost1` is shown in the snapshot:



Accounts Array Example

An index may in general be an expression that evaluate to an integer value as shown in the ghost object below:

```
aGhost2: obj
   accounts: obj Array(3, Account)
```

```
anAccountA, anAccountB, anAccountC : ref Account
inx: var integer
... -- some code assigning references to anAccountA and anAccountB
inx := 1
accounts.put(anAccountA):at[inx]
accounts.put(anAccountB):at[inx + 1]
inx := 3
anAccountC := accounts.get[inx - 2]
```

- The object `aGhost2` has an `Array` `accounts` with 3 elements of type `Account`, three reference variables `anAccountA`, `anAccountB`, and an `AccountC`, and an integer variable `inx`.
- The three dots `...` stand for code not shown – see chapter .
- The variable `inx` assigned the value 1.
- The expression `accounts.put(anAccountA):at[inx]` assigns the reference hold by `anAccountA` to element no. 1 in `accounts`, since the value of `inx` is 1.
- The expression `accounts.put(anAccountB):at[inx + 1]` assigns the reference hold by `anAccountB` to element no. 2 in `accounts`, since expression `inx + 1` has the value 2.
- The statement `anAccountC := accounts.get[inx - 2]` assigns element no. 1 in `accounts` to `anAccountC` since expression `inx - 2` has the value 3 (as `inx` is assigned the value 3 before the statement) – `anAccountC` will thus refer to the same object as `anAccountA`.

If the index expression is not within the range of the array – here 1-3, the program execution will be terminated (aborted) with an error message saying that there is an index error.

We may now extend class `Customer` with an array keeping track of the accounts of the customer:

```
class Customer(name: var String):
   addr: var String
   email: var String
   maxNoOfAccounts: val 10
   noOfAccounts: var integer
   accounts: obj Array(maxNoOfAccounts, Account)
```

In the example, we assume that a customer may have at most 10 accounts as represented by the constant integer value `maxNoOfAcounts`. The integer variable `noOfAccounts` holds the number of accounts of the customer. An integer variable like `noOfAccounts` has initially the value 0 (zero).

Next we add a method `addAccount` to class `Customer`:

```
class Customer(name: var String):
   -"-
   addAccount(acc: ref Account):
      noOfAccounts := noOfAccounts + 1
      if (noOfAccounts <= maxNoOfAccounts) :then
         accounts.put(acc):at[noOfAccounts]
      :else
         console.print(“Cannot add Account”)
```

The method has a parameter `acc` referring to the `Account` to be added. An `if:then:else` statement is used to test if it is possible to add an account or if the limit of the maximum number of accounts has been reached in which case a message is printed on the console.

The `if:then:else` statement is similar to the `if:then` statement as explained in section . The part after else is executed if the condition is false. For a more detailed description see section on statements.

Next we add a method that calculates the sum of the balance on all accounts:

```
class Customer(name: var String):
   -"-
   balanceSum -> bal: var float:
      for (1):to(noOfAccounts):repeat
          bal := bal + accounts.get[inx].balance
```

The method makes use of a *for-statement,* `for:to:repeat` – also called *for-loop*, that iterates through the elements of the array. For each value in the interval `1..noOfAccounts`, the statement `bal := bal + accounts.get[inx].balance` is executed. The variable `inx` has the value 1 in the first execution 2 in the second and so one and `noOfAccounts` in the last execution. For a more detailed description see section .

The complete new version of class `Customer` is shown here:

```
class Customer(name: var String):
   addr: var String
   email: var String
   maxNoOfAccounts: val 10
   noOfAccounts: var integer
   accounts: obj Array(maxNoOfAccounts,Account)

   addAccount(acc: ref Account):
      noOfAccounts := noOfAccounts  + 1
      if (noOfAccounts <= maxNoOfAccounts) :then
         accounts.put(acc):at[noOfAccounts]
      :else
         console.print(“Cannot add Account”)
   balanceSum -> bal: var float:
      for(1):to(noOfAccounts):repeat
          bal := bal + accounts.get[inx].balance
```

Value `Array`

An `Array` may also hold a collection of an indexed sequences of values. The following example shows the declaration of an Array holding integer values:

```
SQ: obj Array(5,integer)
```

This `Array` may hold 5 `integer` values. The name of the array is `SQ`, which is an abbreviation of squares and the code below thus stores the square of the index of a given element in the `Array`:

```
for (1):to(5):repeat
```

```
   put(inx * inx):at[inx]
```

Array literal

An *array literal* is an expression for specifying an `Array` object.

The `SQ` in the previous section, may thus be assigned an `Array` object similar to the one computed by the above `for:to:repeat` loop using an array literal:

```
SQ := (1,4,9,25)
```

The expression `(1,4,9,25)` is an example of an array literal.

The next example shows the use of array literals for arrays holding references:

```
accounts: obj Array(3,Account)
acc1,acc2: ref Account
... -- statements assigning values to acc1 and acc2
accounts := (acc1, acc2, Account("Simon Jones"))
```

The array literal `(acc1,acc2,Account("Simon Jones"))` creates an `Array` object holding references to the `Accounts` referred to by `acc1` and `acc2` and a reference to a new `Acount` object generated by `Account("Simon Jones")`.