Exercise 5

Hand-out: 10 May 2005
Due: 31 May 2005

Master Solution

1. Design by Contract

1.1 Introducing Contracts

Below, we have a class, ACCOUNT, add contracts to it and complete the comments if necessary.

```plaintext
class ACCOUNT
create
make
feature {NONE} -- Initialization
  make (an_amount: like balance) is
    -- Set 'balance' to 'an_amount'.
    require
      an_amount_is_big_enough:
        an_amount >= minimum_balance
    do
      balance := an_amount
    ensure
      balance_set: balance = an_amount
end
feature -- Access
  balance: INTEGER
    -- Account balance
```

minimum_balance: INTEGER is 1000
   -- Minimum amount of money on the account

feature -- Deposit

   deposit (an_amount: like balance) is
      -- Add 'an_amount' to current 'balance'.
      require
         an_amount_is_non_negative: an_amount >= 0
      do
         balance := balance + an_amount
      ensure
         balance_increased: balance = old balance + an_amount
      end

feature -- Withdrawal

   withdraw (an_amount: like balance) is
      -- Substract 'an_amount' from current 'balance'.
      require
         may_withdraw: may_withdraw (an_amount)
      do
         balance := balance - an_amount
      ensure
         balance_decreased: balance = old balance - an_amount
      end

feature -- Status report

   may_withdraw (an_amount: like balance): BOOLEAN is
      -- May 'an_amount' be withdrawn from the account?
      do
         Result := (balance-an_amount >= minimum_balance)
      ensure
         definition:
            Result = (balance-an_amount >= minimum_balance)
      end

invariant

   consistent: balance >= minimum_balance

end
1.2 Contract extraction

Here is an extract of the documentation provided with the .NET Framework for method Insert of class System.Collections.ArrayList:

```eiffel
class ARRAY_LIST

feature -- Element change
insert (an_index: INTEGER; a_value: ANY) is

public virtual void Insert (int index, Object value);

Inserts an element into the ArrayList at the specified index.

Parameters
index
The zero-based index at which value should be inserted.
value
The Object to insert.

Exceptions

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgumentOutOfRangeException</td>
<td>index is less than zero.  -or-</td>
</tr>
<tr>
<td></td>
<td>index is greater than Count.</td>
</tr>
<tr>
<td>NotSupportedException</td>
<td>The ArrayList is read-only.  -or-</td>
</tr>
<tr>
<td></td>
<td>The ArrayList has a fixed size</td>
</tr>
</tbody>
</table>

Remarks
1. If Count already equals Capacity, the capacity of the list is doubled by automatically reallocating the internal array before the new element is inserted.
2. If Capacity is explicitly set to zero, the common language runtime sets it to the default capacity when the first element is added. The default capacity is 16.

Complete the specification with contract of this feature in Eiffel syntax below:

```
-- Insert `a_value' into the list at index `an_index'.

require
  an_index_non_negative: an_index >= 0
  an_index_smaller_than_count: an_index <= count
  writable: not is_read_only
  extendible: not is_fixed_size

do
  ......

ensure
  added: count = old count + 1
  inserted: item (an_index) = a_value
  capacity_doubled_if_needed:
    ((old count = old capacity) and (old capacity /= 0))
    implies (capacity = 2 * old capacity)
  capacity_set_default_if_needed:
    (old capacity = 0) implies (capacity = 16)

end

feature -- Status report
  is_read_only: BOOLEAN is
    -- Is list read-only?
    do
      ......
    end

  is_fixed_size: BOOLEAN is
    -- Does list have a fixed size?
    do
      ......
    end

feature -- Access

  count: INTEGER
    -- Number of elements in the list

  capacity: INTEGER
    -- Number of elements in the list can contain

end
2. Iterations

Assume you have an object \textit{name_list} of type \texttt{LIST [STRING]}. The following code uses a loop to print all elements to the console. For simplicity let's assume also \textit{name_list} and all of its elements are attached (are not \texttt{Void}).

\begin{verbatim}
local
  old_cursor: CURSOR
do
  from
    old_cursor := name_list.cursor
    name_list.start
  until
    name_list.off
loop
  print (name_list.item)
  print ('%N')
  name_list.forth
end
name_list.go_to (old_cursor)
end
\end{verbatim}

Write down the second and more concise way to print out all elements.

\textbf{Solution:}

\begin{verbatim}
do
  name_list.do_all (agent print_line)
end
print_line (a_string: STRING) is
  require
    a_string_not_void: a_string /= Void
  do
    print (a_string)
    print ('%N')
  end
\end{verbatim}