Creating Object Classes in True BASIC

by Thomas E. Kurtz
Co-inventor of BASIC

Introduction

Now that we have seen and experienced the use of an object class, through the driver StackDriver.tru, let's now see how this object class is coded in True BASIC using modules.

You should extract the code included later in this note, save it on your computer, compile it, and test it using the access program contained in the note *Object Oriented Programming in True BASIC*.

What are Modules?

Modules are constructions in True BASIC that allow groups of subroutines to share data that are otherwise private. Without modules, or their equivalent, data entities (i.e., variable and array names) are either global or local. That is, the variable names are global to the entire program, or they are local to a particular external subroutine. (The LOCAL statement allows internal subroutines to have private variables names.)

In technical terms, modules provide "controlled scope" for variable names. Many other languages provide this capability in some form, but I will not go into comparative details here.

Modules in True BASIC provide the *encapsulation* of data together with the subroutines that use that data.

A module is bounded by MODULE and END MODULE statements. The general structure is:

```
MODULE example

      -- module header

      -- module initialization

      -- external routine
                ...
      -- external routine

END MODULE
```

The module header consists first of all the declarative statements that are needed. These include PRIVATE, PUBLIC, DECLARE PUBLIC, and SHARE statements. Compiler directives such as OPTION TYPO and OPTION NOLET should go here as well.

The module initialization includes regular True BASIC statements. These will be executed at program startup time, and provide for initialization of any variables, etc., for which the initial value is important. For example, values should be assigned to variables used as constants in module initialization. And variables designed to distinguish between first and subsequent uses of a routine should be initialized in the module initialization.

After module initialization come any number of external routines. These can be subroutines, defined functions, or pictures. Any variables used within these external routines, other than SHAREd or PUBLIC variables, are local to the routine in which they appear, since the routines are considered to be external.

Variables that are designed to be shared by the external routines within a module must appear in a SHARE statement in the module header. Such variables are like global variables in a program, but they are not accessible from outside the module! This provides the data isolation or data encapsulation so crucial to OOP.

Implementation Details

The *push-down stack* object in True BASIC takes advantage of the fact that arrays and strings occupy dynamic storage (on the heap.) The size of an array can grow or shrink. The length of a string can grow or shrink arbitrarily, up to the amount of available memory. Most requirements for dynamic storage can therefore be met by storing the desired data in arrays or strings. There is no need for an *alloc* function.

Since True BASIC does not provide pointers (e.g., to allocated storage) per se, linked data structures must be implemented with arrays, with one array set aside for use a the contents of a node, and another set aside for use as the pointer to the next node.

Push down stacks can, of course, be implemented using linked lists. But it is also possible to implement them with strings or arrays, since all the activity is from the front end of the stack. Here, we chose to implement a particular stack as a string. We keep the list of stack names in a string array.

Here is the code.

```
MODULE Stack

   ! This module provides for multiple stacks, and has as its
   ! purpose illustrating modules.

   ! The operations on stacks include:
```

```
!
!  DEF Create (stack_name$)        Creates a new stack.
!                                  Returns number of the new
!                                  stac, which is >= 1;
!                                  returning -1 means that the
!                                  stack name is already in use.
!
!  DEF Push (stack_name$, item)    Pushes an item onto a stack.
!                                  Returns the status;
!                                   0: ok
!                                   -1: stack doesn't exist.
!
!  DEF Pop$ (stack_name$)          Returns item popped from stack
!                                  Returns the popped element,
!                                           except
!                                  Returns "empty" if stack is empty.
!                                  Returns "not there" if the stack
!                                  is not there.
!
!  DEF IsEmpty (stack_name$)       Returns 0 if nonempty,
!                                  1 if empty,
!                                  -1 if not there.
!
!      DEF Close (stack_name$)     Closes a stack.
!                                           Returns -1 if not there.

OPTION TYPO

! Declarations in the module header apply throughout the module.

DECLARE DEF StackClass$, Create, Push, Pop$, IsEmpty, Close,
DECLARE DEF NameIsThere

! PRIVATE routines are NOT available from outside the module.

PRIVATE Parse, Create, Push, Pop$, IsEmpty, Close, NameIsThere,
PRIVATE ListStackNames

! SHAREd data elements are available to all routines in the
! module, but are not accessible from outside the module.

 SHARE stacknames$(0)
 SHARE stack$(0)                  The stack storage
 SHARE exists(0)                  ! Distinguishes between an empty
                                  ! and a non-existent stack

! First, here is the (only) access function for the object class

DEF StackClass$ (message$)

LOCAL command$, stack_name$, contents$, status, result$, list$

   CALL Parse (message$, command$, stack_name$, contents$)

   SELECT CASE command$
   CASE "create"
      IF Create (stack_name$) = -1 then
         LET StackClass$ = "That stack name is already taken;
                                  try another."
      ELSE
         LET StackClass$ = "You just created a new stack with
                                  name " & stack_name$
      END IF
```

```
      CASE "push"
           IF Push (stack_name$, contents$) = -1 then
              LET StackClass$ = "That stack doesn't exist."
           ELSE
              LET StackClass$ = "You have just pushed " & contents$ &
                                        " into stack: " & stack_name$
           END IF

      CASE "isempty"
           LET status = IsEmpty (stack_name$)

           IF status = -1 then
              LET StackClass$ = "That stack doesn't exist."
           ELSE IF status = 1 then
              LET StackClass$ = "Stack is empty; sorry."
           ELSE IF status = 0 then
              LET StackClass$ = "Stack is not empty."
           END IF

      CASE "pop"
           LET result$ = Pop$ (stack_name$)

           IF result$ = "empty" then
              LET StackClass$ = "That stack is empty."
           ELSE IF result$ = "not there" then
              LET StackClass$ = "That stack does not exist."
           ELSE
              LET StackClass$ = "Top element of that stack is:
                                    " & result$
           END IF

      CASE "close"
           IF Close (stack_name$) = -1 then
              LET StackClass$ = "That stack doesn't exist."
           ELSE
              LET StackClass$ = "Stack " & stack_name$
                                        & " no longer exists."
           END IF

      CASE "quit", "stop"

      CASE "names"
           CALL ListStackNames (list$)
           LET StackClass$ = "Current stacks are: " & list$

      CASE "?"
           PRINT "Commands are:"
           PRINT "     create stack_name"
           PRINT "     push stack_name item"
           PRINT "     isempty stack_name"
           PRINT "     pop stack_name"
           PRINT "     close stack_name"
           PRINT "     names"
           PRINT "     quit"
           LET StackClass$ = ""

      CASE else
           LET StackClass$ = "Don't recognize: " & message$

      END SELECT

   END DEF
```

```
! Now, the definitions of the supporting routines.

SUB Parse (command_line$, command$, stack_name$, contents$)

    LOCAL p, c$

    ! Peel off the first component

    LET c$ = lcase$(trim$(command_line$))
    LET p = pos(c$, " ")
    IF p = 0 then LET p = len(c$) + 1
    LET command$ = c$[1:p-1]

    LET p = ncpos(c$, " ", p)
    LET c$ = ltrim$(c$[p:1000])

    IF c$ = "" then EXIT SUB

    ! Peel off the second component

    LET p = pos(c$, " ")
    IF p = 0 then LET p = len(c$) + 1
    LET stack_name$ = trim$(c$[1:p-1])

    LET p = ncpos(c$, " ", p)
    LET c$ = ltrim$(c$[p:1000])

    IF c$ = "" then EXIT SUB

    ! Take the third component

    LET p = pos(c$, " ")
    IF p = 0 then LET p = len(c$) + 1

    LET contents$ = trim$(c$[1:p-1])

END SUB

DEF Create (stackname$)

    LOCAL n, i

    LET n = ubound(exists)

    IF NameIsThere (stackname$) > 0 then
       LET Create = -1
       EXIT DEF
    END IF

    FOR i = 1 to n           ! First, try to find a unused spot
        IF exists(i) = 0 then      ! Found one
           LET exists(i) = 1  ! So, use it ..
           LET stacknames$(i) = stackname$
           LET Create = 0
           EXIT DEF          ! .. and exit
        END IF
    NEXT i

    ! New spot needed; allocate new storage

    MAT Redim stack$(n+1), exists(n+1), stacknames$(n+1)
    LET exists(n+1) = 1       ! Use it
```

```
    LET stacknames$(n+1) = stackname$
    LET Create = 0

END DEF

DEF NameIsThere (name$)

    LOCAL n, i

    LET n = ubound(exists)

    LET NameIsThere = -1
    FOR i = 1 to n
        IF lcase$(name$) = lcase$(stacknames$(i)) then
           LET NameIsThere = i
           EXIT DEF
        END IF
    NEXT i

END DEF

DEF Push (sn$, item$)

    LOCAL sn, numchars

    LET sn = NameIsThere (sn$)

    IF sn = -1 then
       LET Push = -1
       EXIT DEF
    END IF

    LET numchars = len(item$)
    LET item$[1:0] = num$(numchars)
    LET stack$(sn)[1:0] = item$    ! Insert new element on front
    LET Push = 0                   ! Signal "no error"

END DEF

DEF Pop$ (sn$)

    ! Pops the top element from the stack named sn$.
    ! Returns: "no there"         if the stack doesn't exist
    !          "empty"           if the stack is empty
    !          the value         otherwise.

    LOCAL sn, numchars

    LET sn = NameIsThere (sn$)

    IF sn = -1 then          ! Check to make sure stack sn is there
       LET Pop$ = "not there"
       EXIT DEF
    END IF

    IF len(stack$(sn)) = 0 then    ! See if stack sn is empty
       LET Pop$ = "empty"
       EXIT DEF
    END IF

    ! Next, REMOVE the top element of the stack.
```

```
    LET numchars = num(stack$(sn)[1:8])      ! Number of characters
    LET Pop$ = stack$(sn)[9:numchars+8]      ! The actual string

    LET stack$(sn)[1:numchars+8] = ""   ! Erase 8+numchars bytes

END DEF

DEF IsEmpty (sn$)

    ! Determines if the stack named sn$ is empty.
    ! Returns: -1 if stack doesn't exist
    !           0 if the stack is NOT empty
    !           1 if the stack IS empty

    LOCAL sn

    LET sn = NameIsThere (sn$)

    IF sn = -1 then           ! Make sure stack sn is there
       LET IsEmpty = -1
       EXIT DEF
    END IF

    IF len(stack$(sn)) = 0 then    ! See if any elements in stack
       LET IsEmpty = 1
    ELSE
       LET IsEmpty = 0
    END IF

END DEF

DEF Close (sn$)

    ! Attempts to close the stack named sn$.
    ! Returns 0 if ok, -1 if stack doesnn't exist.

    LOCAL sn

    LET sn = NameIsThere (sn$)

    IF sn = -1 then           ! Check to make sure stack sn is there
       LET Close = -1
       EXIT DEF
    END IF

    LET stack$(sn) = ""       ! If there, remove its contents and ..
    LET exists(sn) = 0        ! .. remove it from existance

END DEF

SUB ListStackNames (m$)

    ! Lists the names of the available stacks,
    ! and if they are empty or not.

    LOCAL i, lm

    LET m$ = ""
    FOR i = 1 to ubound(exists)
        IF exists(i) = 1 then
           LET m$ = m$ & stacknames$(i) & ", "
        END IF
```

```
        NEXT i

        IF m$ = "" then
           LET m$ = "There are none."
        ELSE
           LET lm = len(m$)
           LET m$[lm-1:lm] = ""
        END IF

    END SUB

END MODULE
```

Notice that this code is robust. There is no limit on the number of push-down stacks, nor the size of any stack, nor of the length of any element in a stack (except for the total amount of memory available.) Furthermore, nothing the user can enter as a command will cause the object class to collapse; error messages are returned for all invalid command sequences. Robustness is a most important property of an object class!

Whether this is the best way to implement push-down stacks is open to question. However, the purpose of this example is to illustrate how one can obtain most of the advantages of OOP in True BASIC using Modules. But it is hard to imagine that the coding details would be more easily understoood in any other language.

Thomas E. Kurtz, April 17, 2000

Comments or questions to: tom@truebasic.com