Object Oriented Programming in True BASIC

by Thomas E. Kurtz
Co-inventor of BASIC

What is OOP?

Object-Oriented Programming, or OOP for short, is the latest and hottest technique that is supposed to revolutionalize programming. Immense improvements in productivity are claimed, through reusability of software.

One recalls the introduction of Structured Programming in the 1970's. Getting rid of "spaghetti code" was a major contribution, but, unfortunately, programming productivity in the main did not markedly improve.

The productivity gains for OOP depend on the size and nature of the project -- the larger the project the more important are any gains -- a sensible discussion is beyond the scope of this note. But with respect to True BASIC, I do want to make two assertions: first, you can achieve most of the benefits of OOP through proper use of True BASIC MODULEs; second, using modules (i.e. True BASIC's OOP tool) makes it possible to manage much larger programming projects comprising 10,000 or more lines of code.

So, what is OOP, really. We might facetiously define it as anything that you can do in C++. Seriously, we have heard numerous definitions, but they boil down to something like this: *Data and procedures are encapsulated, and instructions to and results from operations on the data are communicated to and from the outside through messages*.

The most important aspect of OOP is encapsulation. This means that the ugly coding details are hidden from the using programmer. The term most often used is *object class*. The object class provides a simplified user interface which is often described as passing messages back and forth, messages that tell the object class what you want to do with it, and return responses from the object class.

Other concepts often associated with OOP are: inheritance, operator overloading, and morphology. Inheritance means something like this: Suppose you create a window using one of the windowing systems, such as X-Windows on Unix. Now suppose you want to create a second window that is just like the first but a bit smaller. Inheritance means that you can specify the properties of the second window *without repeating* the properties of the first window, but merely by referring to the first window.

The concept of inheritance is similar to the concept of *subtypes* in most languages that allow generalized typing, such as Pascal and Ada. But OOP requires that the subtyping apply to the procedures that operate on the objects as well.

Some languages, such as Ada and C++, offer the possibility of operator overloading. We all know what "x * y" means, or do we? If x and y are simple numeric variables, then I guess we do know what that means. But what if x and y are vectors? It might then mean the inner product, or it could mean element-by-element multiplication. The point is, the same symbol could mean different things depending on the type of the variables. One common instance is found in some versions of Basic – the operator "+" stands for addition, if its arguments are numeric, and concatenation, if its arguments are strings.

Still another idea is *morphology*, which is the concept of *overloading* applied to procedures. That is, you can have two procedures with the same name but which are different, depending on the type of the arguments. For example, in True BASIC terms, you might want to have two subroutines for sorting lists, one to sort a numeric list and one to sort a string list, as follows:

```
        CALL Sort (x())
and     CALL Sort (x$())
```

That is, the name of the subroutine is the same but the arguments differ. (Of course, this is not allowed in True BASIC.)

What you can do in TB

Encapsulation is provided by modules in True BASIC. The subroutines contained within the module can share variables, provided these variables are named in SHARE statements. These variables can be numeric or string, simple or arrayed. SHAREd entities can also include file or channel numbers, such as #3.

Isolation is provided by the fact that SHAREd variables cannot be accessed from subroutines outside the module.

If you wish to have variables accessible throughout the program, as in Fortran COMMON, you can do so by including their names in PUBLIC statements. PUBLIC variables can also be shared, obviously.

Variables that are SHAREd or PUBLIC retain their values between calls to the subroutines that are using them.

Message Passing in True BASIC is made possible by allowing strings as arguments to subroutines. Handling strings in True BASIC is easier than in any other language, since strings are a primitive data type and the language contains several functions for searching, inserting, etc.

What you cannot do in TB

True BASIC does not provide *inheritance*. This is because inheritance applies to data types and subtypes. For example, in Pascal one can define a data type called *day*.

```
type day = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

and then define a subtype called weekday

```
    subtype weekend of day = (Sun, Sat);
```

A more realistic example is to define a generic kind of window, which might be quite complicated, and then to define several special kinds of windows whose properties are those of the generic window plus other properties.

Because True BASIC does not include type declarations, it cannot provide inheritance. However, the number of uses of inheritance is not so great to put True BASIC at a distinct disadvantage for typical application programs. (True BASIC does, however, provide a limited form of inheritance through its COPY method; see the subroutine Object.)

True BASIC does not allow overloading of operators except in MAT statements where the keyword MAT reduces the possibility of confusion. That is, if A, B, and C are two-dimensional matrices, and v and w are vectors,

```
    MAT C = A*B
    MAT C = A*v
    MAT C = w*A
    MAT C = 7*A
```

* standards for matrix multiplication in the first three cases, and scalar multiplication in the fourth case. In the second case, the vector v is taken as a column vector, while in the third case, w is taken as a row vector.

While True BASIC does not allow morphology, not much is lost. Instead of

```
    CALL Sort (x())
    CALL Sort (x$())
```

you have to do something like this:

```
    CALL SortN (x())
    CALL SortS (x$())
```

You can write SortS simply by makeing a *copy* of SortN, and then making global editing changes of x to x$. It would take a matter of seconds!

Even with morphology, the programmer must write all versions of the procedure. For example, inside the sort procedure are statements of the type:

```
  IF x < y then
      ! Do this if x occurs before y in the collating sequence
  ELSE
      ! Do this otherwise.
  END IF
```

The problem is the symbol “<”. It is obvious what this means with numbers. It is less obvious what this means when applied to strings. (For instance, it does not, in True BASIC, mean dictionary order: A, a, B, b. Rather, it means ASCII character order: A, B, ..., Z, ..., a, b, ..., z, ...) If x and y are more complicated data structures, such as file records, the programmer must provide the code that determines the collating order of the file records.

Problems with Number Typing

While number typing (double precision, single precision, long integer, short integer) is not directly related to OOP, we choose to make a comment about it here.

Coercion is a term applied to the automatic conversion of numbers from one type to another. For example, in Fortran there are two number types – fixed (integer) and floating. Unless otherwise specified, numeric variables whose initial letter is I, J, K, L, M, or N are assumed to be fixed, while other initial letters imply floating. Only fixed numbers are allowed in DO statements.

Suppose you are computing the average of N numbers. You might do it this way in Fortran:

```
     DO 10, I = 1, N
10   SUM = SUM + X(I)
     EN = N
     AVE = SUM/EN
```

The automatic coercion occurs in the third statement, where a fixed numeric quantity is converted to a floating numeric value.

Some languages require conversion functions. A typical use in Pascal might be:

```
     en := float(n);
```

Some languages provide automatic coercion within expressions, as in this Fortran-like example.

```
     DO 10, I = 1, N
10   SUM = SM + X(I)
     AVE = SUM/N
```

None of this is required in True BASIC because there is only one number type.

Some will counter by saying “Yes, but ...” and following with a discussion of greater runtime speed with two-byte and four-byte integers, in contrast with eight-byte IEEE floating point. We concede this, but we challenge you to find real examples where True BASIC provides unacceptable performance because of its lack of short and long integer data types.

In fact, True BASIC uses special techniques internally to speed up the FOR loop, for example. In most cases, the insides of the FOR loop take much more time than the loop management, even when floating point numerics are used for the loop management.

An Example

This example will be of a module that will allow you create, use, and close an arbitrary number of push-down stacks. The example is simple enough to understand so that we can concentrate on the technical details.

```
PROGRAM StackDriver

LIBRARY "StackClass.trc"

OPTION TYPO

DECLARE DEF StackClass$

LOCAL message$

PRINT "Type ? for list of commands."
DO

   INPUT prompt "Enter your message: ": message$

   IF message$ = "stop" or message$ = "quit" then EXIT DO

   PRINT StackClass$(message$)
   PRINT

LOOP

END
```

This driver assumes that the file “StackClass.trc” contains an object class whose access function is named StackClass$. The input or command message is provided in the argument message$, and the result, even if an error, is sent back as the value of the access function. The only limitation due to True BASIC is that the contents of the stacks must be simple strings.

As far as the user programmer is concerned, she is working with an object class whose internal details she cannot see, but which provides any number of push down stacks. This example may seem trivial, but it does illustrate the main point of object oriented programming – encapsulation.

To emphasize the distinction between using an object, and creating it, we leave the creation details to a separate note: *Creating Object Classes*. However, you should download that note and extract the code contained in it, save it on your computer, compile it, and run the above access program.

Thomas E. Kurtz, April 17, 2000

Comments or questions to: tom@truebasic.com