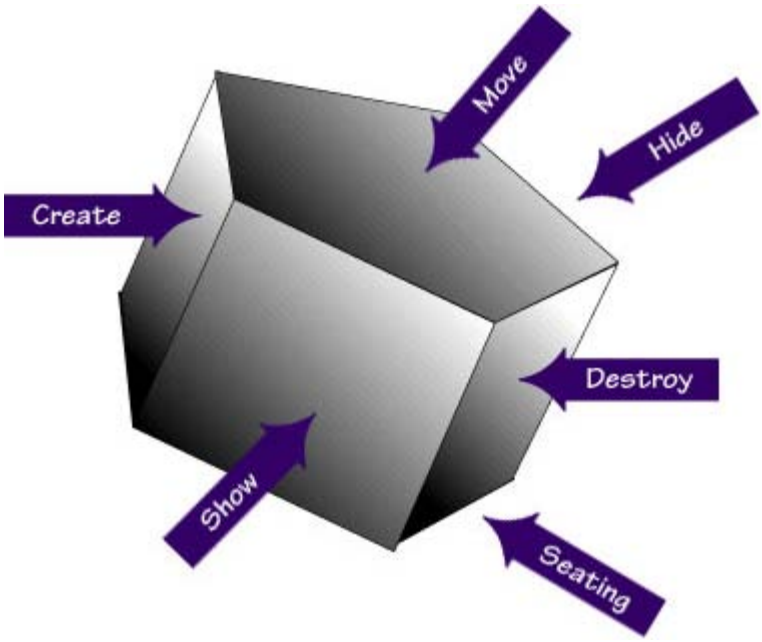
Yes, Virginia, There Are Objects In PRO/5®

By Michael Martinez

I like objects and look forward to their introduction to Business Basic, but I'm the impatient sort of programmer who doesn't want to wait for the next generation of tools to start doing things. We already have graphical objects, of course, but we manipulate them through the SYSGUI device, which is itself rather like an object that we communicate with. I want to get closer to real object-oriented programming: being able to define a *thing* and have that *thing* do stuff. This is what object-oriented programming is all about. The object can be something in memory-some special piece of data. It can be a graphical control, like a button or a list box. Or it can be a file or a disk-resident data structure.

What makes an object an object? That's a question for which there are a thousand answers. I like to say that an object is a bounded data space. The boundaries are not physical; rather, they are logical. The boundary of an object is defined partially by the type definition assigned to its data region (but you can, theoretically, create a null object that doesn't have a data region). The boundary is also defined by the way the object behaves. The behavior consists of the set of operations you can tell an object to perform, and these operations are usually referred to as methods in object-oriented parlance. A method is a mode of access to an object-part of the object's interface. We do not actually manipulate the object's contents directly, but rather ask the object to do things through its various methods. A method may be either a function or a procedure.



A function method returns some data item from its object. This may cause confusion, since the data item may be information *about* the object rather than information *from* the object's data region. For

instance, suppose we create a table object. We give this table object a method called *seating*. So, when we want to know how many people can sit at the table, we ask for the *table.seating* value. This is a function of the table object, which tells us the capacity of the table, but not how many people are actually sitting at the table.

Or our table object may need to be moved from the kitchen to the dining room, but instead of our picking up the table and physically moving it, we use the *move* method to tell the table to move to the dining room: *table.move(dining room)*. This would be a procedural method.

Suppose we want to have multiple table objects? Can we call them all *table*? No. Naming an object is similar to naming variables: each language implements its object reference structures in some fashion that lets you distinguish between objects. We cannot have multiple objects called *table*, but we don't want to dispense with the name *table*-it is much too useful a word to discard. So we'll define a class of objects called *table*, which defines what a table object is and how it behaves but which is not itself an object. Now when we create a table object we might call it *Fred* or *Cathy* or *Bill*.

By creating the table object *Fred* we use the *table* class, which defines the methods and data region for the table. So, we can ask *Fred* to tell us what its seating capacity is or we can tell *Fred* to move to the dining room. *Fred* is an object of class *table* and behaves as we would expect a table object to behave.

How does this relate to Visual PRO/5™ (or PRO/5)? We are used to Business Basic programs that perform utilitarian functions. A standard input routine is one example of such a program. Or a date validation routine. The date validation routine is very similar to a method that a date object might possess. We might ask the date object if it contained a valid date, getting back a TRUE or FALSE answer. The *date.valid* method would be a function method. We could also ask a date object to update itself to the next month (useful in generating statements or checks). This would be a procedural method.

Most people feel object-oriented programming cannot be implemented in traditional Business Basic because the language lacks encapsulation. Encapsulation refers to a programming language's interface to objects-it's the wall that keeps the programmer from tinkering with the object's innards. Languages, such as PRO/5 and Visual PRO/5 that do not support encapsulation expose your data region and methods to anyone who know how to write code.

There is a way to make your objects more remote, however: implement object managers. An object manager is a program, function, or subroutine that looks and feels like an object interface.

The program is not an object but is the interface to the object. Let's say we want to create and use a class called *Account*. This class defines objects that contain 24 data segments called "10-digit dollar amounts," a data segment called "name," and a data segment called "description." The class might use the following methods: *create, destroy, load, clear, total,* and *store*.

The *create* and *destroy* methods are used to initialize and remove the objects. The object manager can be used to create as many objects as required. The objects must be persistent from invocation to invocation, and uniquely named. The two most efficient means of implementing uniquely named, persistent objects in Business Basic is to store them in the global string table or in files. Disk accesses are slower than memory accesses so an object manager that is expected to handle only a few objects at a time may work best with the global string table.

Although it would be inefficient to use a called program to access a disk file one record at a time, let us assume that the *load* method somehow tells the object to retrieve data from some location and the *store* method causes the reverse operation to occur. An object manager implemented as a subroutine would be the best choice for managing objects that work with files. To work with the object manager, we require at least two parameter variables: Method$ and Parameters$. If the object manager is implemented as a function, only two parameters are required. If it is implemented as a subroutine or called program a third parameter variable, Result$, is required.

Let's say we store general ledger data in a file named GLACCTS. We wish to use these records in an object metaphor to simplify our coding efforts, so we have developed an object manager to handle the *Account* class. To create an object, we'll use the account code as the unique identifier. Thus, to create an object we invoke the manager with the *create* method.

Called program:

```
Method$="create"
Parameter$=GLACCT.ACCOUNT_CODE$
call"accounts.bbx",Method$,Parameter$,Result$
```

Multiline function:

```
Method$="create"
Parameter$=GLACCT.ACCOUNT_CODE$
Result$=FNAccounts$(Method$,Parameter$)
```

Subroutine:

```
Method$="create"
Parameter$=GLACCT.ACCOUNT_CODE$
gosub Accounts
```

Eventually we'll have to assign multiple parameters to an object manager, but the Parameter$ variable can handle this easily. We can implement our own terminators, such as commas, and have the object manager parse them. Although this may seem inefficient, it's easier to maintain object management logic across hundreds of programs if you don't have to change them all every time a new parameter or method is added to a class. Each method can then take responsibility for ensuring it receives the parameters it requires, including type checking if necessary.