Advanced Object-oriented programming

Xbase^{++} is one of the most powerful programming languages available, but many features of the Xbase^{++} language remain undiscovered. This article reveals a rarely used feature of Xbase^{++}: the dynamic creation of classes.

The programming task

I am using 2-dimensional arrays quite often as a replacement of a database file. The arrays contain temporary data built from scratch during runtime of my application. All I know of the 2-dimensional array is its data structure and the number of elements it has. I.e. column 1 contains this data, column 2 contains that data, and so many elements are available.

Accessing and traversing a 2-dimensional array can be a tedious programming job that I wanted to get rid of. The columns of a 2-dimensional array are usually addressed with a #define constant while the current row of a 2-dimensional array is represented by a loop counter. The following code demonstrates this typical approach:

```
#define ADR_FIRSTNAME   1
#define ADR_LASTNAME    2

SELECT Customer

FOR i:=1 TO Len(aAddress)
   REPLACE FIELD->FIRSTNAME WITH aAddress[i, ADR_FIRSTNAME]
   REPLACE FIELD->LASTNAME  WITH aAddress[i, ADR_LASTNAME ]
   SKIP
NEXT
```

What is this code all about? There is an array *aAddress* containing data to be transferred to a database file. The structure of the array is known (#define constants) but the number of elements is unknown (Len(aAddress)).

In my opinion, #define constants are uncomfortable to program with. I'd rather like to write the above code as follows:

```
SELECT Customer

DO WHILE ! oAddress:eof()
   REPLACE FIELD->FIRSTNAME WITH oAddress:FIRSTNAME
   REPLACE FIELD->LASTNAME  WITH oAddress:LASTNAME
   oAddress:skip()
   SKIP
ENDDO
```

The variable *oAddress* is an object that knows of the array *aAddress*! The object allows for accessing individual array columns by using symbolic names rather than #define constants. The object also knows a "row pointer" so that the array can be treated almost like a database.

Treating a 2-dimensional array almost like a database is extremely convenient. To achieve this, an object is required that knows how to translate symbolic names to numeric indexes pointing to a single array column. The object must also be able to maintain a "row pointer", a "begin of file" and an "end of file" flag, so that the object can be used by a browser to navigate through the array.

Since I don't like to create classes for every data structure, I came up with the following programming task:

1. Program a Meta class that is able to create classes dynamically who's objects are able to treat a 2-dimensional array like a database file.
2. Objects of dynamically created classes must be able to access columns of a 2-dimensional array by a symbolic name, i.e. assigning a value to an instance variable of the object must also change the value of an array element.

This code demonstrates the initial task:

```
aAddress[ 1, ADR_FIRSTNAME ] := "Bill"
? oAddress:FIRSTNAME                    // --> Bill

oAddress:FIRSTNAME           := "Michael"
? aAddress[ 1, ADR_FIRSTNAME ]          // --> Michael
```

Assigning a value to an array element or to an instance variable of an object has the same effect: the contents of one array element is changed.

The Meta class concept

Let me explain the term "Meta class" first for those who don't have a clue what this is. A Meta class is a class that can create Class Objects. A Class Object defines all member variables and methods for all instances of that class.

Usually a class object is created using the CLASS .. ENDCLASS declaration. A Meta class object can do the same without explicit declaration. It can bee seen as a "Class factory" since classes can be created on the fly and without the CLASS declaration.

The Meta class I want to create must be able to generate a class that handles a 2-dimensional array. For example:

```
aAddr    := { { "John", "Doe" }, ;
              { "Jane", "Doe" }  }

aColumns := { "FIRSTNAME", "LASTNAME" }

oClass   := MetaClass():createClass( "Address", aColumns )

oAddr    := oClass:new( aAddr )

? oAddr:FIRSTNAME   // --> John
```

This code demonstrates the basic principle that can be applied to any 2-dimensional array. It consists of three steps:

1. Create a symbolic name for each column of a 2-dimensional array (aColumns)
2. Create a class for a 2-dimensional array of known structure (oClass)
3. Create an object of the new class and pass the array on to it (oAddr)

When these three steps are done, elements of the array can be accessed via instance variables of the object.

Implementing a Meta class

I have named the Meta class for 2-dimensional arrays "RecordSet" because that's what a 2-dimensional array is: a Set of Records. The RecordSet class creates a Class object by its method *:createClass()* which accepts the name of the new class as a string and an array of column names.

```
01: CLASS METHOD RecordSet:createClass( cClassName, aColumnNames )
02:    LOCAL oClass := ClassObject( cClassName )
03:    LOCAL i, imax:= Len( aColumnNames )
04:    LOCAL aMethod, cBlock, cName, nType
05:
06:    IF oClass <> NIL
07:       RETURN oClass
08:    ENDIF
09:
10:    nType := CLASS_EXPORTED + METHOD_INSTANCE + ;
11:             METHOD_ACCESS  + METHOD_ASSIGN
12:
13:    aMethod:= Array( imax )
14:    FOR i:=1 TO imax
15:       cName  := aColumnNames[i]
16:       cBlock := "{|o,x| IIf(x==NIL,"                        + ;
17:                         "o:getVar(" + Var2Char(i) + "),"   + ;
18:                         "o:putVar(" + Var2Char(i) + ",x))}"
19:       aMethod[i] := { cName, nType, &(cBlock), cName }
20:    NEXT
21:    oClass := ClassCreate( cClassName, { self }, {}, aMethod )
22:    oClass:initClass( aColumnNames )
23:
24: RETURN oClass
```

If I would have to declare a CLASS for the "John/Jane Doe" array, mentioned in the previous section, I would have to write this code:

```
01: CLASS Addr FROM RecordSet
02:    EXPORTED:
03:    INLINE ACCESS ASSIGN METHOD FIRSTNAME( c )
04:    RETURN IIF( c==NIL, ::records[ ::recno, 1 ] , ;
05:                        ::records[ ::recno, 1 ] := c )
06:
07:    INLINE ACCESS ASSIGN METHOD LASTNAME( c )
08:    RETURN IIF( c==NIL, ::records[ ::recno, 2 ] , ;
09:                        ::records[ ::recno, 2 ] := c )
10: ENDCLASS
```

These two code snippets are equivalent for the "John/Jane Doe" array and they explain the value of Meta classes best: I need 10 lines of code for the declaration of a CLASS that knows only **two columns** of a particular 2-dimensional array. I need just 24 lines of code for a CLASS METHOD that creates classes which know **any column** of an arbitrary 2-dimensional array.

The secret of the RecordSet class is that it does not create instance variables for each column of an array. It creates ACCESS/ASSIGN methods instead. Each ACCESS/ASSIGN method is mapped to the generic :getVar() / :putVar() methods of the RecordSet class (see *cBlock* line #16) which receive the numeric index for one array column (Var2Char(i)).

Since the :getVar() / :putVar() methods are declared and implemented in the RecordSet class, this class is not only a Meta class but also the Super class for all classes it creates. All classes created by RecordSet are derived from RecordSet (line #21 -> { self } is the class object of RecordSet)

When a Meta class object serves as Super class object for derived classes, all methods can be implemented in the Meta class that must be known in derived classes.

The navigational concept

We have seen that addressing a particular array column via symbolic name is achieved in the RecordSet class by mapping the symbolic name to a generic method, :getVar() and :putVar(). The implementation of these methods could be as simple as this:

```
INLINE METHOD getVar( nColumn )
RETURN ::records[ ::recno, nColumn ]

INLINE METHOD putVar( nColumn, xValue )
RETURN ::records[ ::recno, nColumn ] := xValue
```

The 2-dimensional array is referenced in the instance variable *::records*. Both methods receive a numeric index *nColumn* that points to the desired array column. Since a column index is not sufficent to address a single array element of a 2-dimensional array, the object must know the "current record" or the row pointer of the array. This information is stored in the instance variable *::recno* and is required for all 2-dimensional arrays.

When an object knows a row pointer, it is able to manipulate it. All we have to do is to add methods that change the row pointer. For example:

```
INLINE METHOD skip( n )
   IF n == NIL
      n := 1
   ENDIF

   ::recno += n

   IF ::recno < 1
      ::recno := 1
   ENDIF

   IF ::recno > ::lastrec
      ::recno := ::lastrec
   ENDIF
RETURN self
```

This code implements the method *:skip()* and works similar to the SKIP command known for database files. It simply changes the row pointer and keeps it within a valid range.

Summary

1. The dynamic creation of classes is an extremely powerful but rarely used feature of Xbase^{++}
2. This article points out the concept of Meta classes and how they can be implemented.
3. Meta classes are Super classes for derived classes
4. The RecordSet class is a Meta class. It creates classes that allow for accessing single elements of 2-dimensional arrays via symbolic names

The entire source code for the RecordSet class is available at this location:

ftp://ftp.alaska-software.com/weblib/documents/RecordSet.prg

The code demonstrates how to use the RecordSet class in a generic approach for browsing 2-dimensional arrays.