Object-Oriented Programming With Object REXX

BY RICK BYRLEY

In last month's column I demonstrated how to create an Object REXX "Hello World" program as an introduction to the Object REXX environment that comes with OS/2. As I mentioned, Object REXX is not only a useful programming language in its own right, but it is also an excellent introduction to object-oriented programming techniques for those used to the traditional, structured programming approach. Since "Classic REXX" programs will run under the Object REXX interpreter, REXX is perfect for experimenting with the two very different approaches and learning the principles of object-oriented programming. This month I will examine some of the fundamental concepts behind an object-oriented language and how these concepts apply to Object REXX in particular.

SOME PRELIMINARIES

Some very simple example code is provided to illustrate these principles, but first, a short course in Object REXX program structure is in order. There are two parts to any true object-oriented Object REXX program: the executable code and the class definitions. The executable code is placed at the beginning of the program and ends when the first directive is encountered. Typically the executable code will create instances of objects and then call the objects' various methods to perform the work of the program.

Directives (keywords preceded by two semicolons, e.g., ::CLASS) define classes and their methods. There are four directives and only two of them, ::CLASS and ::METHOD, will be of interest here. Note that the ::CLASS directive continues to define a class until the next ::CLASS directive is reached, and a ::METHOD directive continues to define a method until the next directive (either ::CLASS or ::METHOD) is encountered.

Now that these ground rules are out of the way, let's look at the three major principles of object-oriented programming: data encapsulation, polymorphism, and inheritance.

DATA ENCAPSULATION

There are many ways to define objects, but the simplest seems to be that an object is data and the code necessary to manipulate the data. Those of you familiar with BASIC programming might remember how data was stored within a program using the DATA statement, which made the task completely self-contained — it did not need to access data sources external to itself. I like to think of objects like those old BASIC programs: data and the code to manipulate that data. This illustrates one of the principles of object-oriented programming: data encapsulation. Each object has its own data that is protected from other objects.

An object is an abstraction defined by a class. Just like a program will do nothing until it is invoked, an object will do nothing unless an instance of the object is created. Once created — and this is important —according to the principle of data encapsulation each instance has its own copy of the object data.

For example, in the sample code "encapsulation.cmd" in Figure 1, two instances of MyClass are created, and the variable "MyVar" is set in each instance. The program then calls another method to display the value of the variable to illustrate that each instance contains its own version of the MyVar variable. This is data encapsulation.

POLYMORPHISM

As mentioned previously, an object is data and the code to manipulate that data. This code consists of object methods. A method is similar to a function in that it

Figure 1: ENCAPSULATION.CMD

```
/*****************************************************************************
*
*  PROGRAM: ENCAPSULATION.CMD
*  AUTHOR:  Rick Byrley, SofTouch Systems
*  PURPOSE: Illustrates data encapsulation in Object REXX
*
*****************************************************************************/

   /* Create two instances of the MyClass object */
   instance1 = .MyClass~new
   instance2 = .MyClass~new

   /* Call each instance's method to demonstrate data encapsulation */
   instance1~setMyVar("This is instance 1")
   instance2~setMyVar("This is instance 2")

   /* Display the value of the variable in each instance */
   instance1~sayMyVar
   instance2~sayMyVar

   ::class MyClass

   ::method 'setMyVar'
     expose myvar
     parse arg myvar

   ::method 'sayMyVar'
     expose myvar
     say myvar
```

Figure 2: POLYMORPHISM.CMD

```
/*****************************************************************************
*
*  PROGRAM: POLYMORPHISM.CMD
*  AUTHOR:  Rick Byrley, SofTouch Systems
*  PURPOSE: Illustrates polymorphism in Object REXX
*
*****************************************************************************/

   /* Create instances of the OneClass and AnotherClass objects */
   instance1 = .OneClass~new
   instance2 = .AnotherClass~new

   /* Call each instance's method set the value of the variable */
   instance1~setMyVar("This is instance 1")
   instance2~setMyVar("This is instance 2")

   /* Display the value of the variable in each instance */
   instance1~sayMyVar
   instance2~sayMyVar


   ::class OneClass

   ::method 'setMyVar'
     expose myvar
     parse arg myvar

   ::method 'sayMyVar'
     expose myvar
     say "This is OneClass MyVar:" myvar

   ::class AnotherClass

   ::metho 'setMyVar'
     expose myvar
     parse arg myvar

   ::method 'sayMyVar'
     expose myvar
     say "This is AnotherClass MyVar:" myvar
```

Figure 3: INHERITANCE.CMD

```
/*****************************************************************************
*
*  PROGRAM: INHERITANCE.CMD
*  AUTHOR:  Rick Byrley, SofTouch Systems
*  PURPOSE: Illustrates inheritance in Object REXX
*
*****************************************************************************/

   /* Create instances of the OneClass and AnotherClass objects */
   instance1 = .OneClass~new
   instance2 = .AnotherClass~new

   /* Call each instance's methods to say Hello */
   instance1~sayHello
   instance2~sayHello

   ::class AClass

     ::method sayHello
       say "Hello!"

   ::class OneClass subclass AClass


   ::class AnotherClass subclass AClass

     ::method sayHello
       say "Greetings!"
```

generally performs a discrete action. For example, a file object might have methods to read the file, write the file, copy the file, etc.

In contrast to traditional programming, methods do not carry the load of having to adapt within a single program to various kinds of data, since in an object the data is structurally identical each time.

"Reading" a text file is different than "reading" the next item in a list, although both a file and list object may use a "read" method. This use of the same label to perform different actions on different objects is called polymorphism, and is another fundamental principle of object-oriented programming.

The sample code in Figure 2 illustrates polymorphism. The program creates two objects, one from the OneClass class and the other from the AnotherClass class. Both classes have the identical methods setMyVar and sayMyVar. The first method is identical in both classes; sayMyVar, however, differs between the two classes.

INHERITANCE

Thus, each object has its own unique version of a method, more or less independent of all other objects and methods. I say more or less because a third principle of object-oriented programming, inheritance, determines which methods an object actually possesses. All objects (except for the object "object") inherit properties from another class. This feature is why object-oriented code is highly reusable: simply inherit the parts you want, override the parts you don't, code any unique parts, and you have a new object. Depending on which book you read, the class from which another class is derived is called a "parent" or "super." The super class is specified as an option to the ::CLASS directive when the class is defined by declaring the class to be a subclass of the named superclass.

The inheritance.cmd sample in Figure 3 is a simple case of inheritance. Both the OneClass and AnotherClass classes are subclasses of the AClass class. Therefore, they both inherit the "sayHello" method from the AClass class, which is why the method can be invoked on the OneClass instance (instance 1) although no such "sayHello" method is defined in the code for the class. The AnotherClass class overrides the default "sayHello" method defined by the parent class AClass, choosing to say "Greetings" rather than "Hello."

SUMMARY

Data encapsulation, polymorphism, and inheritance are the foundation of any object-oriented language, from Small Talk to C++ to Java to Object REXX. Fortunately, REXX is such an easy language syntactically that these principles aren't lost in the complexity of the code, which makes it a perfect introduction to object-oriented programming. And it's free with OS/2. ts

Rick Byrley is senior workstation division technician for SofTouch Systems, Oklahoma City, Okla., which provides both mainframe and PC software solutions. His primary focus is object-oriented programming. He can be reached at rbyrley@softouch.com.