Open Object Rexx Tutorial

Overview

by Howard Fosdick

Open Object Rexx, or *ooRexx*, is an open source programming language based on Rexx. Sometimes people refer to Rexx as *"classic Rexx"* to distinguish between the two. Classic Rexx supports procedural programming, while ooRexx supports both procedural and object-oriented programming.

ooRexx is a true superset of classic Rexx. So any classic Rexx program you write will run without alteration under ooRexx. This is a great advantage for several reasons. It means your code is portable between the two systems. And, so are your skills. So if you already know Rexx, learning ooRexx is as simple as expanding your skillset to include the additional object-oriented features it provides.

This chapter presents a simple ooRexx tutorial. Assuming that you know classic Rexx, it bootstraps you into the world of object-oriented programming with simple, complete program examples.

Open Object Rexx retains the advantages of simplicity and clear syntax from classic Rexx. It adds the classes, methods and all the other features of object-oriented programming with a minimum of new syntax. Concentrate on solving the problem at hand, and leverage Open Object Rexx's new features in the quest.

Let's quickly discuss the sample scripts in this chapter. The first is very short; it merely tests for the existence of a file. It illustrates how to use a stream object and a few of its methods perform I/O. The next two scripts show you how to create your own classes and methods. Then, a more ambitious script implements a video checkout application for a DVD library. The final example demonstrates concurrency, where methods within a class run at the same time.
asfs

[This tutorial was excerpted from Chapter 28 of the *Rexx Programmer's Guide*, Wiley, © 2005, with the first page rewritten]

The examples in this chapter were all tested under Linux. Open Object Rexx runs on several platforms including Linux, Windows, Solaris, and AIX.

A First Program

Open Object Rexx features a *class library*, a built-in set of classes and methods that adds a wide range of capability to traditional Rexx. The *methods* in this library perform actions, much like the built-in functions of classic Rexx. One easy way to start object-oriented programming with Object Rexx is just to program as in classic Rexx, but to replace function calls with object methods.

Here is an example. This simple script reads one input parameter from the command line, a filename. The program determines whether the file exists and displays a message with its result:

```
[root /usr/bin/oorexx]$ ./oorexx file_exist.orex  square.orex
File exists
[root /usr/bin/oorexx]$ ./oorexx file_exist.orex  nonesuch.txt
File does not exist
[root /usr/bin/oorexx]$
```

The program is called File Exist. In the first preceding run, the input file exists. In the second run, the file did not exist in the current directory. All the script does is write a message indicating whether the file specified on its command line exists. Recall that the syntax `./oorexx` is merely a way of ensuring, on Unix-derived operating systems, that the module named `oorexx` in the current directory will be invoked to run the script.

Here is the code for the first program:

```
/*********************************************************************/
/*  FILE EXIST                                                       */
/*                                                                   */
/*  Tells if a specified file exists.                                */
/*********************************************************************/
parse arg file .                          /* get user input file  */

infile = .stream~new(file)                /* create stream object */

/* Existence test returns either full filename or the null string     */

if infile~query('exists') = ''  then      /* test if nonexistent  */
   .output~lineout('File does not exist')     /* no such file exists  */
else
   .output~lineout('File exists' )            /* found the filename   */

exit 0
```

To work with file I/O in Open Object Rexx, you can either use classic Rexx instructions such as `say`, `pull`, `parse pull`, `charin`, `charout`, `chars`, `linein`, `lineout`, and `lines`, or you can use the new object-oriented methods. This script uses the methods.

The first action when using OO input/output is always to create a *stream instance*:

```
infile = .stream~new(file)                    /* create stream object     */
```

This object manages I/O for one input or output file. To create it, we send the message `new` to the built-in `.stream` class. The `.stream` class is denoted by the period immediately prior to the keyword `stream`. We passed the filename from the user to the `new` method as its input parameter. Now we have an object created representing an I/O stream for that file.

The following code invokes the `query` method on the new stream object. The method returns the null string if the file does not exist in the current directory. If the file does exist, it returns the fully qualified name of the file:

```
if infile~query('exists') = ''  then          /* test if nonexistent */
   .output~lineout('File does not exist')     /* no such file exists */
```

The `lineout` method is invoked with the character string to write as its input parameter. `.output` is one of the special *built-in objects* described in Chapter 27. It is a *monitor object* that forwards the messages it receives to `.stdout`, the stream object representing standard output.

Of course, if the file does exist, the script writes the appropriate message:

```
else
   .output~lineout('File exists' )            /* found the filename  */
```

This script uses the built-in object-oriented classes and methods for I/O. This same program could be coded using classic Rexx instructions such as `say` and `pull`, and it would have run under Open Object Rexx as well. All classic Rexx functions have object-oriented counterparts (methods) in Open Object Rexx.

The trick to learning ooRexx is to learn its class library. Remembering what it offers as built-in classes and methods is as important as knowing what functions are available in classic Rexx. This knowledge is the lever that enables you to exploit the language fully and let its built-in capabilities do the work for you.

Squaring a Number — Using Our Own Class and Method

The previous example shows how to leverage Open Object Rexx's large class library. This is especially useful when performing tasks that would otherwise require a lot of work, for example, in creating graphical user interfaces or performing database I/O. Now let's look at how to create and use our own classes and methods within ooRexx scripts.

This simple script squares a number. The user enters the number as a command-line argument, and the script writes back its squared value:

```
[root /opt/orexx/pgms]$  square.orex  4
The original value: 4  squared is: 16
```

Here is the code for this script:

```
/*****************************************************************/
/* SQUARE                                                        */
/*                                                               */
/* Returns the square of a number                                */
/*****************************************************************/
parse arg input .          /* get the number to square from user   */

value = .squared~new       /* create an object for a squared value */
sqd = value~square(input)  /* invoke SQUARE method on INPUT value  */

say 'The original value:' input 'squared is:' sqd

exit 0

::class squared            /* Here is the class.                   */

  ::method 'square'        /* Class SQUARED has 1 method, SQUARE.  */
  use arg in               /* get the input argument               */
  return ( in * in )       /* square it and return that value      */
```

In this script, the first task is to create an instance to square the value. Depending on what school of OOP you follow, this instance might also be called an *object instance*, the *instantiation of an object class*, or just an *object*. We're not concerned with formal terminology here; you don't need to be to use ooRexx. This statement creates the object:

```
value = .squared~new       /* create an object for a squared value */
```

The next line in the script invokes the `square` method to perform the work of squaring the number. It does this by sending the appropriate message to the object:

```
sqd = value~square(input)  /* invoke SQUARE method on INPUT value  */
```

Look down further in the code to see the code of the `square` method and its class, called `squared`. All classes and methods must follow the procedural code located at the top of the program. New classes and methods are always placed at the bottom of the script. Two colons indicate a *directive*, an identifier for a class, method, routine or external code block. This line defines our new class called `squared`:

```
::class squared             /* here is the class                    */
```

After the class directive, we place our methods and their code. This class has but a single method named `square`. This line defines this method:

```
::method 'square'           /* Class SQUARED has 1 method, SQUARE.   */
```

The name of a method is a character string. The interpreter matches this string with the message sent to the object (to invoke it) in the procedural code. Here the method name is coded within quotation marks, as many programmers prefer. You can also define a method name without the quotation marks:

```
::method  square           /* class SQUARED has 1 method, SQUARE   */
```

The method's code immediately follows this *method directive*. Its first line reads its input argument:

```
use arg in                 /* get the input argument               */
```

In Open Object Rexx, you code `use arg` instead of `arg` or `parse arg` to retrieve one or more input arguments. The method squares the number provided in the input argument and returns it as its return string through the `return` instruction:

```
return ( in * in )         /* square it and return that value      */
```

That's it! You can create your own classes and methods in this fashion. Run the methods in those objects in the same way that you run the built-in methods provided by ooRexx.

In many ways, creating objects and methods is similar to creating subroutines and functions in classic Rexx. The difference is in the hierarchical structure of object-oriented programming. The ability to inherit behaviors (attributes and methods) through the class hierarchy minimizes the amount of new coding you have to do. This advantage becomes most pronounced in large programming projects, where the chances for code reuse are higher. It becomes significant in any situation in which you can optimally leverage ooRexx's built-in class library.

Another Simple OO Program

Here's another simple OO script. This one allows the user to enter a shell name and responds with the full name of the shell or the operating system. Here's an example interaction with the program:

```
Enter the shell name:
csh
OS is: C Shell
```

The user enters a shell name, `csh`, and the script responds with its full name. Here's another interaction:

```
Enter the shell name:
COMMAND
OS is: Windows 2K/2003/XP
```

The user enters the shell for his or her operating system, `COMMAND`, and the program responds that the shell is used under several versions of the Windows operating system. The program recognizes a couple other inputs (`ksh` and `CMD`), but comes back with this message for any other input:

```
Enter the shell name:
pdksh
OS is: unknown
```

Here is the code for the script:

```
/*********************************************************************/
/* WHICH OS                                                          */
/*                                                                   */
/* Tells which operating system you use depending on the command shell. */
/*********************************************************************/
os = .operating_systems~new       /* create a new object             */

os~write_command_shell            /* invoke the method to do the work   */

exit 0

::class operating_systems         /* class with 2 methods following it  */

  ::method init                   /* method INIT prompts for shell name */
    expose shell                  /* EXPOSE the shared attribute        */
    say 'Enter the shell name:'   /* prompt for and read user input     */
    parse pull shell .
    return

  ::method write_command_shell    /* This method determines the OS.     */
    expose shell
    select                        /* determine the OS for this shell    */
      when shell = 'CMD'      then string = 'DOS or Windows 9x'
      when shell = 'COMMAND'  then string = 'Windows 2K/2003/XP'
      when shell = 'ksh'      then string = 'Korn Shell'
      when shell = 'csh'      then string = 'C Shell'
      otherwise string = 'unknown'
    end
    say 'OS is:' string           /* write out the OS determined        */
    return 0
```

This script only contains three lines of procedural code. The first line creates a new instance of the class `.operating_systems`. Send the class the `new` method message to create a new instance of the class:

```
os = .operating_systems~new             /* create a new object         */
```

The second line in the program runs the method `write_command_shell` in the class. This method does all the real work of the program:

```
os~write_command_shell                  /* invoke the method to do the work   */
```

The last line of procedural code, the `exit` instruction, ends the program. Class and method directives follow, along with the code that defines them. The class(es) and their method(s) are always placed at the bottom of the code in the script. This line defines the `.operating_systems` class:

```
::class operating_systems               /* class with 2 methods following it */
```

This class is followed by its two methods. The first is the `init` method:

```
::method init                           /* method INIT prompts for shell name */
```

The `init` method is a specially named method. Open Object Rexx always runs the `init` method (if there is one) whenever its creates a new instance via the `new` message. So, the first line in the script not only created a new instance of the `operating_systems` class but also automatically ran the `init` method.

In the `init` method, the first line of code uses the `expose` instruction to access the variable named `shell`. By using `expose`, the method has read and update capability on any variables or attributes it names. The `expose` instruction is the basic technique by which attributes can be shared among methods in a class:

```
expose shell                              /* EXPOSE the shared variable         */
```

After accessing this variable, the `init` method prompts the user to enter a shell name, reads that user input, and returns:

```
say 'Enter the shell name:'               /* prompt for and read user input     */
parse pull shell .
return
```

The second line of code in the driver runs the `write_command_shell` method. This method also accesses the attribute `shell` by its `expose` instruction:

```
expose shell
```

Then, the method executes a `select` instruction to determine the full shell name or associated operating system, and its writes this response to the user:

```
select                          /* determine the OS for this shell    */
  when shell = 'CMD'      then string = 'DOS or Windows 9x'
  when shell = 'COMMAND'  then string = 'Windows 2K/2003/XP'
  when shell = 'ksh'      then string = 'Korn Shell'
  when shell = 'csh'      then string = 'C Shell'
  otherwise string = 'unknown'
end
say 'OS is:' string             /* write out the OS determined        */
```

When this method returns, the main routine or procedural code terminates with an `exit` instruction.

This script shows how user interaction can be encapsulated within classes and methods. The procedural code in the main routine or driver can be minimal. The classes and their methods perform all the work. Once you start thinking in object-oriented terms, you'll develop the knack of viewing problems and their solutions as groups of interacting objects with their logical methods.

Implementing a Stack through Objects

Object Rexx provides a large set of *collection classes*, built-in classes that provide a set of data structures and for data manipulation. The collection classes are:

- Array — A sequenced collection
- Bag — A nonunique collection of objects (subclass of Relation)

- Directory — A collection indexed by unique character strings
- List — A sequenced collection which allows inserts at any position
- Queue — A sequenced collection that allows inserts at the start or ending positions
- Relation — A collection with nonunique objects for indexes
- Set — A unique collection of objects (subclass of Table)
- Table — A collection with unique objects for indexes

You'll rarely have to create your own data structures with all this built-in power available. But for the purpose of illustration, we use the `List` collection class as the basis to implement a stack in the next sample script.

Here is a sample interaction with the stack script:

```
Enter items to place on the stack, then EXIT
line1
line2
line3
line4
exit
Stack item # 1 is: LINE4
Stack item # 2 is: LINE3
Stack item # 3 is: LINE2
Stack item # 4 is: LINE1
```

The script prompts the user to enter several lines and then the keyword `exit`. Here the user entered four lines plus the keyword `exit`. Then the script pops the stack to retrieve and print the items. Since a stack is a last-in, first-out (LIFO) data structure, the items display in the reverse order in which they are entered.

How do we design this script? First, identify the stack as the entity or *object* with which the script works. This should be a class that we can instantiate by creating an object.

Second, try to identify which operations or *methods* need to be executed on that object. Push and pop are two key stack operations, so we'll need a method for each of these. Further reflection leads to the realization we also require an initialization method (`init`) and a method to return the number of items on the stack.

Identifying objects and their methods are the basic steps in object-oriented design. One other step (not relevant to this simple example) is determining the relationships or interactions between objects.

So, the script will have one class, called `stack`, and four methods in that class. Here are the methods and their functions:

- `init` — Initializes the stack
- `push` — Places an item (or line) onto the stack (*pushes* the item)
- `pop` — Retrieves an item from the stack (*pops* the item)
- `number_items` — Returns the number of items on the stack

With this understanding of what objects and methods are required, we can write the program:

```
/*********************************************************************/
/*  STACK                                                            */
/*                                                                   */
/*  Implments a Stack data structure as based on the LIST Collection  */
/*********************************************************************/
the_stack = .stack~new                /* create a new stack object    */

.output~lineout('Enter items to place on the stack, then EXIT')
stack_item = .input~linein~translate  /* read user's input of 1 item  */

do while (stack_item <> 'EXIT')       /* read all user's items to     */
   the_stack~push(stack_item)         /*   push onto the stack and    */
   stack_item  = .input~linein~translate   /* translate to upper case */
end

do j=1 by 1 while (the_stack~number_items <> 0)  /* pop and display   */
   say 'Stack item #' j 'is: ' the_stack~pop     /*   all stack items */
end

exit 0

::class stack                        /* define the STACK class         */

 ::method init                       /* define INITIALIZATION method   */
   expose stack_list                 /* STACK_LIST is our stack.       */
   stack_list = .list~new            /* create a new STACK as a LIST   */

 ::method push                       /* define the PUSH method         */
   expose stack_list
   use arg item
   stack_list~insert(item, .nil)     /* insert item as 1st in the LIST */

 ::method pop                        /* define the POP method          */
   expose stack_list
   if stack_list~items > 0 then      /* return item, remove from stack */
      return stack_list~remove(stack_list~first)
   else
      return .nil                    /* return NIL if stack is empty   */

 ::method number_items               /* define the ITEMS method        */
   expose stack_list
   return stack_list~items           /* return number of items in stack*/
```

As in previous scripts, the first action is to create an instance of the class object. Here we create a stack to work with:

```
the_stack = .stack~new                        /* create a new stack object     */
```

The program prompts the user to enter several lines of data:

```
.output~lineout('Enter items to place on the stack, then EXIT')
stack_item = .input~linein~translate         /* read user's input of 1 item     */
```

The script reads a line with the `linein` method applied to the default input stream via the `.input` monitor object. This input line is then acted upon by the `translate` method. The code chains these two methods so that they execute one after the other. This reads and translates the input to uppercase.

For each line the script reads, it places it into the stack. This runs the `push` method with the `stack_item` as input:

```
the_stack~push(stack_item)                  /*   push onto the stack and       */
```

After all user-input lines have been read and placed onto the stack, this loop retrieves each line from the stack via the `pop` method and writes them to the user:

```
do j=1 by 1 while (the_stack~number_items <> 0)  /* pop and display     */
   say 'Stack item #' j 'is: ' the_stack~pop     /*   all stack items   */
end
```

The loop invokes our method `number_items` on the stack to determine how many items are in the stack. All the methods `expose` the stack so that they can work with it and read and alter its contents. Here is the code that exposes the program's shared attribute:

```
expose stack_list                   /* STACK_LIST is our stack.         */
```

Let's discuss each of the methods in this program. The `init` method automatically runs when the stack object is first created; it merely creates a new List object. Recall that we selected the built-in *collection class* of type List with which to implement our stack. This line in the `init` method creates the new List object:

```
stack_list = .list~new              /* create a new STACK as a LIST    */
```

The `push` method grabs its input argument and places it into the stack. It uses the List class's `insert` method to do this. The first argument to insert is the line to place into the list, while the second is the keyword `.nil` which says to place the item first in the list:

```
use arg item
stack_list~insert(item, .nil)           /* insert item as 1st in the LIST    */
```

The `pop` method checks to see if there are items in the stack by executing the List class's method `items`. If the List contains one or more items, it returns the proper item from the List by the `remove` method:

```
 if stack_list~items > 0 then             /* return item, remove from stack    */
   return stack_list~remove(stack_list~first)
else
   return .nil                          /* return NIL if stack is empty      */
```

If there are no items in the List, the `pop` method returns the NIL object. Coded as `.nil`, this represents the absence of an object (much in the same manner the null string represents the string with no characters).

The method `number_items` simply returns the number of items currently on the stack. It does this by running the `List` method `items`:

```
return stack_list~items                 /* return number of items in stack   */
```

To summarize, this script shows how you can use an object and Open Object Rexx's built-in collection classes and methods to create new data structures. With its built-in collection classes, ooRexx is a rich language in terms of its support for data structures.

A Video Circulation Application

The next script controls videos for a store that rents movie DVDs. It is a more ambitious script that demonstrates how a number of methods can be applied to a Directory collection class object. The program presents a menu like the following one to the user. After the user makes a selection, the script performs the action the user selected. Here's an example, where the store employee adds a new DVD title to the collection:

```
1. Add New DVD
2. Check Movie Out
3. Check Movie In
4. Show Movie Status
5. Remove Lost DVD
X. Exit
Enter Your Choice  ==> 1

Enter Movie TITLE  ==> Titantic
Movie added to titles: TITANTIC
```

After each action is complete, the script clears the screen again and redisplays the menu. The program terminates when the user selects the option: `X. Exit`.

The program error-checks for logical mistakes. For example, it will not let the user check out a video that is already checked out, nor will it allow the user to check in a movie that is already on hand. The script always ensures the title the user refers to is in the collection. If not, it writes the appropriate message to the user.

As in the stack program, the Videos program implements an in-memory data structure to control the videos. This script uses the Directory built-in collection class to make an indexed list of videos. The film title is the index into the Directory; the sole data item associated with the video's title is its status. The status can either be `IN LIBRARY` or `CHECKED OUT`. For simplicity, the program assumes that there is only a single copy or DVD for each movie title.

The Directory of videos will be the class called `movie_dir`. The methods for this class are:

- `init` — Initialize the Directory
- `add_movie` — Add a film to the circulating collection
- `check_out_movie` — Check a movie out
- `check_in_movie` — Check a movie back in
- `check_status` — List the status of a movie (`IN LIBRARY` or `CHECKED OUT`)
- `lost_or_destroyed` — Remove a lost or destroyed DVD from the circulating collection

Here is the script:

```
/*****************************************************************/
/*  VIDEOS                                                       */
/*                                                               */
/*  An in-memory circulation system for films on DVD.            */
/*****************************************************************/
movie_list = .movie_dir~new     /* create new Directory object   */

do while selection <> 'X'

   /* display menu of options                                    */

   'clear'
   say "1. Add New DVD"    ;  say "2. Check Movie Out"
   say "3. Check Movie In" ;  say "4. Show Movie Status"
   say "5. Remove Lost DVD" ;  say "X. Exit"

   /* prompt user to enter his choice and the movie title        */

   call charout ,'Enter Your Choice  ==> '
   pull selection .     ;  say " "
   if (selection <> 'X') then do
      call charout ,'Enter Movie TITLE  ==> '
      pull title .
   end

   /* perform user selection                                     */

   select
      when selection = '1' then movie_list~add_movie(title)
      when selection = '2' then movie_list~check_out_movie(title)
      when selection = '3' then movie_list~check_in_movie(title)
      when selection = '4' then movie_list~check_status(title)
      when selection = '5' then movie_list~lost_or_destroyed(title)
      when selection = 'X' then exit 0
      otherwise say 'Invalid selection, press <ENTER> to continue...'
   end
   pull .                   /* user presses ENTER to continue    */
end
exit 0

::class movie_dir             /* define the MOVIE_DIR class      */

  ::method init               /* INIT - create the DIRECTORY     */
     expose mv_dir            /* expose the DIRECTORY of interest*/
     mv_dir = .directory~new  /* creates the DIRECTORY class     */

  ::method add_movie          /* ADD_MOVIE                       */
     expose mv_dir
     use arg title            /* if title is new, add it by PUT  */
     if .nil = mv_dir~at(title) then do
        mv_dir~put('IN LIBRARY',title)     /* add  by PUT method */
        say 'Movie added to titles:' title
```

```
        end
      else                    /* if title is not new, err message*/
        say 'Movie is already in collection:' title

   ::method check_out_movie    /* CHECK_OUT_MOVIE                */
      expose mv_dir
      use arg title            /* if title doesn't exist, error   */
      if .nil = mv_dir~at(title)  then
        say 'No such title to check out:' title
      else do                  /* if ALREADY checked out, error   */
        if 'CHECKED OUT' = mv_dir~at(title) then
          say 'Movie already checked out:' title
        else do                /* if no error, check out the title*/
          mv_dir~setentry(title,'CHECKED OUT')    /* alters data*/
          say 'Movie is now checked out:' title
          end
      end

   ::method check_in_movie     /* CHECK_IN_MOVIE                 */
      expose mv_dir
      use arg title
      if .nil = mv_dir~at(title) then  /* if no title, error      */
        say 'No such title to check in:' title
      else do                          /* if not checked out, err */
        if 'IN LIBRARY' = mv_dir~at(title) then
          say 'This title is ALREADY checked in:' title
        else do                        /* otherwise check it in   */
          mv_dir~setentry(title,'IN LIBRARY')   /*  alters data */
          say 'The title is now checked back in:' title
        end
      end

   ::method check_status       /* CHECK_STATUS                   */
      expose mv_dir
      use arg title            /* if no such title, error        */
      if .nil = mv_dir~at(title) then
        say 'Title does not exist in our collection:' title
      else                     /* if title exists, show its status*/
        say mv_dir~at(title)  /* retrieve data by the AT method  */

   ::method lost_or_destroyed  /* LOST_OR_DESTROYED              */
      expose mv_dir
      use arg title
      if .nil = mv_dir~at(title) then /* if no such title, error  */
        say 'Title does not exist in our collection:' title
      else do                  /* if title exists, remove it      */
        mv_dir~remove(title)  /* REMOVE method deletes from Dir. */
        say 'Title has been removed from our collection:' title
      end
```

The first line of procedural code creates the instance or object the script will work with. This is the same approach we've seen in previous example scripts, such as the Stack and Which OS programs:

```
movie_list = .movie_dir~new     /* create new Directory object   */
```

The next block of code clears the screen, displays the menu and reads the user's selection. The `select` instruction runs the proper method to perform the user's selection. The `.movie_dir` class and its six methods follow the procedural code. Let's briefly discuss each of the methods.

The `init` method creates the Directory instance:

```
mv_dir = .directory~new         /* creates the DIRECTORY class   */
```

The `add_movie` method checks to see if the title the user entered is in the circulating collection. It uses the `at` method on the Directory object to do this:

```
if .nil = mv_dir~at(title) then do
```

If `.nil` is returned, the script adds the title to the circulating collection (the Directory) with the status `IN LIBRARY`:

```
mv_dir~put('IN LIBRARY',title)     /* add  by PUT method         */
```

The `put` method places the information into the Directory. To check out a DVD, the method `check_out_movie` uses the `setentry` method on the Directory to alter the DVD's status:

```
mv_dir~setentry(title,'CHECKED OUT')      /* alters data         */
```

Method `check_in_movie` similarly runs the `setentry` built-in method to alter the DVD's status:

```
mv_dir~setentry(title,'IN LIBRARY')       /*  alters data        */
```

Method `check_status` executes method `at` to see whether or not a film is checked out:

```
if .nil = mv_dir~at(title) then
```

It then displays an appropriate message on the status of the video by these lines:

```
          say 'Title does not exist in our collection:' title
       else                     /* if title exists, show its status*/
          say mv_dir~at(title)  /* retrieve data by the AT method  */
```

Finally, the method `lost_or_destroyed` removes a title from the circulating collection by running the Directory `remove` method:

```
mv_dir~remove(title)        /* REMOVE method deletes from Dir. */
```

Like most of the collection classes, Directory supports alternative ways of invoking several of its methods. Here are a couple alternative notations:

- [ ] — Returns the same item as the `at` method
- [ ]= — Same as the `put` method

For example, we could have written this line to add the new title to the Directory:

```
mv_dir[title] = 'IN LIBRARY'
```

This statement is the direct equivalent of what we actually coded in the sample script:

```
mv_dir~put('IN LIBRARY',title)     /* add  by PUT method         */
```

Similarly, we could have checked for the existence of a title by referring to `mv_dir[title]` instead of coding the `at` method as we did. This is another way of coding that reference:

```
if  .nil  =  my_dir[title]  then
```

This statement is the same as what was coded in the script:

```
if .nil = mv_dir~at(title) then    /* if no such title, error    */
```

This script employs simple `say` and `pull` instructions to create a simple line-oriented user interface. Most programs that interact with users employ graphical user interfaces or GUIs. How does one build a GUI with Open Object Rexx? This is where the power of built-in classes and methods comes into play. The menu becomes a *menu object* and the built-in classes and methods activate it. Creating a GUI becomes a matter of working with prebuilt objects typically referred to as *widgets* or *controls*. Chapter 16 discusses graphical user interfaces for object-oriented Rexx scripting. Besides a GUI, the other feature that is missing from the Videos script is *persistence*, the ability to store and update object data on disk. This simple script implements the entire application in memory. Once the user exits the menu, all data entered is lost. Not very practical for the video store that wants to stay in business!

Object-oriented programmers typically add persistence or permanence to their objects through interfacing with one of the popular database management systems. Among commercial systems, Oracle, DB2 UDB, and SQL Server are popular. Among open-source products, MySQL, PostgreSQL, and Berkeley DB are most popular.

GUI and database capability are beyond the scope of our simple example. Here the goal was to introduce you to object-oriented programming, not to get into the technologies of GUIs and databases. Of course, these interfaces are used by most real-world object-oriented Rexx programs. Using class libraries and methods reduces the work you, as a developer, must do when programming these interfaces.

Concurrency

Object-oriented programming with Open Object Rexx is *concurrent* in that multiple objects' methods may be running at any one time. Even multiple methods within the same object may execute concurrently.

The following sample script illustrates concurrency. Two instances of the same class are created and execute their single shared method concurrently. To make this clearer, here is the script's output. The intermixed output shows the concurrent (simultaneous) execution of the two instances:

```
Repeating the message for object #1 5 times.
Object #1 is running
Object #2 is running
Repeating the message for object #2 5 times.
Object #1 is running
Object #2 is running
Driver is now terminating.
Object #1 is running
Object #2 is running
Object #1 is running
Object #2 is running
Object #1 is running
Object #2 is running
```

Object #1 and Object #2 are two different instances of the same class. That class has one method, called `repeat`, which displays all the messages seen above, from both objects (other than the one that states that the `Driver is now terminating`.)

Here is the code of the script:

```
/*********************************************************************/
/*  CONCURRENCY                                                      */
/*                                                                   */
/*  Illustrates concurrency within an object by using REPLY instruction */
/*********************************************************************/
object1 = .concurrent~new             /* create two instances,          */
object2 = .concurrent~new             /* both of the CONCURRENT class   */

say object1~repeat(1,5)               /* get 1st object running         */
say object2~repeat(2,5)               /* get 2nd object running         */

say 'Driver is now terminating.'
exit 0

::class concurrent                    /* define the CONCURRENT class    */

::method repeat                       /* define the REPEAT method       */
  use arg who_am_i, reps              /* get OBJECT_ID, time to repeat  */
  reply 'Repeating the message for object #' || who_am_i reps 'times.'
  do reps
    say 'Object #'||  who_am_i 'is running'   /* show object is running */
  end
```

The script first creates two separate instances of the same class:

```
object1 = .concurrent~new             /* create two instances,         */
object2 = .concurrent~new             /* both of the CONCURRENT class  */
```

The next two lines send the same message to each instance. They execute the method `repeat` with two parameters. The first parameter tells `repeat` for which object is it executing (either 1 for the first object or 2 for the second). The second parameter gives the `repeat` method a loop control variable that tells it how many times to write a message to the display to trace its execution. This example executes the method for each instance five times:

```
say object1~repeat(1,5)              /* get 1st object running        */
say object2~repeat(2,5)              /* get 2nd object running        */
```

Following these two statements, the driver writes a termination message and exits. The driver has no further role in the program. Almost all of the program output is generated by the `repeat` method, written to show its concurrent execution for the two instances.

Inside the `repeat` method, this line collects its two input arguments. It uses the `use arg` instruction to read the two input arguments:

```
use arg who_am_i, reps               /* get OBJECT_ID, time to repeat */
```

The next line issues the `reply` instruction. `reply` immediately returns control to the caller at the point from which the message was sent. Meanwhile, the method containing the `reply` instruction keeps running:

```
reply 'Repeating the message for object #' || who_am_i reps 'times.'
```

In this case, `reply` sends back a message that tells which object is executing and how many times the `repeat` method will perform its `do` loop. Now, the `repeat` method message loop continues running. This code writes the message five times to demonstrate the continuing concurrent execution of the `repeat` method:

```
do reps
   say 'Object #'||  who_am_i 'is running'   /* Show object is running */
end
```

This simple script shows that objects and methods execute concurrently, and that even methods within the same object may run concurrently. Open Object Rexx provides a simple approach to concurrency that requires more complex coding in other object-oriented languages.

Summary

This chapter introduces classic Rexx programmers to Open Object Rexx. It does not summarize or demonstrate all the OO features of ooRexx. Instead, it presents a simple tutorial on the product for those from a procedural-programming background.

We started with a very simple script. That first script created a stream instance and demonstrated how to perform object-oriented I/O. Two more advanced scripts followed. These showed how to define and code your own classes and methods. The Stack sample program was more sophisticated. It defined several different methods that demonstrated how to employ the List collection class to implement an in-memory stack data structure. The Videos application built upon the same concepts, this time using the Directory collection class to simulate a video circulation control system. Finally, the Concurrency script illustrated the use of two instances of the same class and the concurrent execution of their methods.

Open Object Rexx is ideal for learning object-oriented programming. It retains all the simplicity and strengths of classic Rexx while surrounding it with a plethora of OO features and the power of an extensive class library.

Test Your Understanding

1. Will every classic Rexx program run under Open Object Rexx? When might this be useful? Does this capitalize on ooRexx's capabilities?
2. Are all classic Rexx instructions and functions part of Open Object Rexx? Which additional instructions and functions does ooRexx add?
3. What is a collection and how are they used?
4. Convert these two functions in classic Rexx to ooRexx method calls:

```
reversed_string = reverse(string)
sub_string = substr(string,1,5)
```

5. Do the `stream` class methods offer a superset of classic Rexx I/O functions? What additional features do they offer?
6. What are the four kinds of directives and what is each used for? Where must classes and methods be placed in a script?
7. What symbols commonly express `at` and `put` in many collections?
8. What are the error (`.error`), input (`.input`), and output (`.output`) monitor objects used for?