

Object-Oriented COBOL

Subprograms often implement a kind of object-oriented behavior. Programmers will use them to create a persistent memory local to the subprogram. The subprogram then accepts an extra argument representing the action or “method” desired. This requires nested IF statements or an EVALUATE statement to decode the action. One limitation of this method is that the local variables of a subprogram are created only once per program, like a class with only static methods and variables. Object-oriented COBOL allows the creation of separately-invoked subprograms, called methods, that share instance variables; also, multiple instances of an object can be created, each with its own set of instance variables.

Each object is defined by a class definition. The class definition begins like any program/subprogram but has a class-id instead of a program-id. The environment division contains a repository for external references to other class definitions. The class definition can be divided into static class definitions (part of the factory paragraph) or object-specific variables and methods (object paragraph). Each section has its own working-storage section and methods. Each method is essentially a subprogram contained within the class or an object that is an instance of that class.

A program or class that creates objects of a class must include that class in its repository. Variables are created without a picture; instead they are usage OBJECT REFERENCE. Like Java, all objects are referenced indirectly through the variable (it’s a pointer). The objects are created through a built-in new method belonging to the class. The “new” method (and any factory methods) are called with the INVOKE command and the name of the class followed by the method name, in quotes like any subprogram.

INVOKE class-name “new” returning object-reference-variable.

Once you have a object reference, you can use the variable in place of the class name to invoke object methods of the class.

INVOKE object-reference-variable “object-method” using arguments.

INVOKE object-reference-variable “another-object-method” using arguments
returning result.

The following example implements a class for manipulating credit state information. State machines are an efficient means of storing “history”. A state value can represent the fact that a proposal has been made, is being considered, outlook is positive, but it hasn’t been approved, yet. Another state value can represent that a proposal has been made, outlook was positive, but it was rejected and now the proposal has been modified and resubmitted. If the number of possibilities are finite, then it is a simple matter to assign a number to each possibility and define the conditions where one number can become another number (in the previous example, the first state could be changed to another by a rejection, then changed to the second state by the resubmission event). State machines are used in compilers, interpreters, communications systems to track protocol states, and many other situations where short-term changes of state are expected. They are also quite useful in long-term situations since points in a complex process can be saved very compactly in a file until the next situation requires a change in the recorded situation (a change of state).

The following example has the states:

- 1: mediocre credit; need more evidence of good credit
- 2: mediocre; need a little more evidence of good credit
- 3: mediocre; need one more example of trustworthiness
- 4: good credit (just barely)
- 5: good credit
- 6: good credit (almost excellent)
- 7: excellent credit
- 8: poor credit; need at least 6 examples of trustworthiness to be good
- 9: poor credit; need at least 5 examples of trustworthiness to be good
- 10: poor credit (almost mediocre); need at least 4 examples
- 11: bad credit; 3 on-time payments to be poor credit
- 12: bad credit; 2 on-time payments to be poor credit
- 13: bad credit; 1 on-time payment to be poor credit

Transitions are caused by missed payments (later than one month), late payments (less than one month), on-time payments, bad credit report from outside company (account closed for nonpayment), poor credit report from outside company (past-due balances), mediocre credit reports (late payments) and good credit reports from other companies. Typically, bad reports "reduce" the credit rating from good toward bad and good reports and on-time payments "increase" the credit rating from bad toward excellent.

```
identification division.
class-id. CreditState as "creditstate"
        inherits from base.
* CreditState is the name of a class; externally (to other
* languages) it is known as creditstate (other languages
* don't support COBOL identifier syntax, like hyphens, and
* COBOL, by default, capitalizes all exported identifiers).
* All classes must inherit from another class; just as Java
* classes are all descendants of Object, COBOL classes are
* descendants of the base class.

repository.
        class base as "base".
* external reference to base class

factory.
* the following variables and methods belong to the class
* rather than the objects instantiated from the class.
working-storage section.
01  w-actions.
    05  pic x(30) value "on-time payment".
    05  pic x(30) value "late payment".
    05  pic x(30) value "missed payment".
    05  pic x(30) value "good report".
    05  pic x(30) value "mediocre report".
    05  pic x(30) value "poor report".
```

```

05 pic x(30) value "bad report".
05 pic x(30) value "late payment reported".
05 pic x(30) value "account past due".
05 pic x(30) value "account frozen".

01 w-action-table redefines w-actions.
05 w-action pic x(30) occurs 10 times.

01 w-state-names.
05 pic x(20) value "mediocre credit".
05 pic x(20) value "mediocre credit".
05 pic x(20) value "mediocre credit".
05 pic x(20) value "good credit".
05 pic x(20) value "good credit".
05 pic x(20) value "good credit".
05 pic x(20) value "excellent credit".
05 pic x(20) value "poor credit".
05 pic x(20) value "poor credit".
05 pic x(20) value "poor credit".
05 pic x(20) value "bad credit".
05 pic x(20) value "bad credit".
05 pic x(20) value "bad credit".

01 w-state-table redefines w-state-names.
05 w-state-name pic x(20) occurs 13 times.

* called with INVOKE CreditState "encode-action"
* USING string RETURNING action-code.
* Supposedly, methods that return a value can also be used
* like a function call:
* MOVE CreditState::"encode-action"(string) TO W-ACTION.
method-id. encode-action.
linkage section.
01 l-string pic x(30).
01 l-code pic 9(4) comp-x.
procedure division using l-string returning l-code.
move 1 to l-code.
perform until l-code > 10 or
l-string = w-action(l-code)
add 1 to l-code
end-perform.
if l-code > 10 then
move 0 to l-code
else if l-code > 7 then
subtract 3 from l-code.
exit method.
end method encode-action.

method-id. get-state-description.
linkage section.

```

```
01 l-code pic 9(4) comp-x.
01 l-msg pic x(20).
procedure division using l-code, l-msg.
    move w-state-name(l-code) to l-msg.
    exit method.
end method get-state-description.
```

```
end factory.
```

```
object.
```

```
* each time you invoke CreditState "new" returning object,
* each object references a unique set of the following
* variables; methods invoked through an object reference
* will use the appropriate set of variables for that object
working-storage section.
```

```
01 w-current-state pic 9(4) comp-x.
```

```
* The following initializes a table of 13x7 entries
* representing the transitions that can occur from one
* state to another depending on an event (called an action,
* here).
```

```
01 w-state-transitions.
```

```
* State 1 transitions to 2 (on-time), stays at 1 (late
* payment), backs to 10 (missed payment), moves to 2 (good
* report), stays at 1 (mediocre report), backs to 10 (poor
* report) or to 8 (bad report).
```

```
    05 pic 9(4) comp-x value 2.
    05 pic 9(4) comp-x value 1.
    05 pic 9(4) comp-x value 10.
    05 pic 9(4) comp-x value 2.
    05 pic 9(4) comp-x value 1.
    05 pic 9(4) comp-x value 10.
    05 pic 9(4) comp-x value 8.
```

```
* State 2 transitions to 3 (on-time), backs to 1 (late
* payment), backs to 10 (missed payment), moves to 4 (good
* report), stays at 2 (mediocre report), backs to 1 (poor
* report), and backs to 9 (bad report).
```

```
    05 pic 9(4) comp-x value 3.
    05 pic 9(4) comp-x value 1.
    05 pic 9(4) comp-x value 10.
    05 pic 9(4) comp-x value 4.
    05 pic 9(4) comp-x value 2.
    05 pic 9(4) comp-x value 1.
    05 pic 9(4) comp-x value 9.
```

```
* State 3 moves to 4 (on-time, good), 2 (late, poor),
* 10 (missed, bad), and stays at 3 (mediocre)
```

```
    05 pic 9(4) comp-x value 4.
    05 pic 9(4) comp-x value 2.
    05 pic 9(4) comp-x value 10.
    05 pic 9(4) comp-x value 4.
    05 pic 9(4) comp-x value 3.
```

```
    05 pic 9(4) comp-x value 2.
    05 pic 9(4) comp-x value 10.
* State 4
    05 pic 9(4) comp-x value 5.
    05 pic 9(4) comp-x value 3.
    05 pic 9(4) comp-x value 1.
    05 pic 9(4) comp-x value 5.
    05 pic 9(4) comp-x value 3.
    05 pic 9(4) comp-x value 2.
    05 pic 9(4) comp-x value 1.
* State 5
    05 pic 9(4) comp-x value 6.
    05 pic 9(4) comp-x value 4.
    05 pic 9(4) comp-x value 1.
    05 pic 9(4) comp-x value 6.
    05 pic 9(4) comp-x value 4.
    05 pic 9(4) comp-x value 3.
    05 pic 9(4) comp-x value 1.
* State 6
    05 pic 9(4) comp-x value 7.
    05 pic 9(4) comp-x value 5.
    05 pic 9(4) comp-x value 1.
    05 pic 9(4) comp-x value 7.
    05 pic 9(4) comp-x value 5.
    05 pic 9(4) comp-x value 4.
    05 pic 9(4) comp-x value 1.
* State 7
    05 pic 9(4) comp-x value 7.
    05 pic 9(4) comp-x value 6.
    05 pic 9(4) comp-x value 4.
    05 pic 9(4) comp-x value 7.
    05 pic 9(4) comp-x value 6.
    05 pic 9(4) comp-x value 4.
    05 pic 9(4) comp-x value 1.
* State 8
    05 pic 9(4) comp-x value 9.
    05 pic 9(4) comp-x value 8.
    05 pic 9(4) comp-x value 13.
    05 pic 9(4) comp-x value 10.
    05 pic 9(4) comp-x value 9.
    05 pic 9(4) comp-x value 8.
    05 pic 9(4) comp-x value 11.
* State 9
    05 pic 9(4) comp-x value 10.
    05 pic 9(4) comp-x value 8.
    05 pic 9(4) comp-x value 13.
    05 pic 9(4) comp-x value 1.
    05 pic 9(4) comp-x value 9.
    05 pic 9(4) comp-x value 8.
    05 pic 9(4) comp-x value 12.
```

```

* State 10
  05 pic 9(4) comp-x value 1.
  05 pic 9(4) comp-x value 9.
  05 pic 9(4) comp-x value 8.
  05 pic 9(4) comp-x value 2.
  05 pic 9(4) comp-x value 10.
  05 pic 9(4) comp-x value 9.
  05 pic 9(4) comp-x value 13.
* State 11
  05 pic 9(4) comp-x value 12.
  05 pic 9(4) comp-x value 11.
  05 pic 9(4) comp-x value 11.
  05 pic 9(4) comp-x value 8.
  05 pic 9(4) comp-x value 12.
  05 pic 9(4) comp-x value 11.
  05 pic 9(4) comp-x value 11.
* State 12
  05 pic 9(4) comp-x value 13.
  05 pic 9(4) comp-x value 12.
  05 pic 9(4) comp-x value 11.
  05 pic 9(4) comp-x value 8.
  05 pic 9(4) comp-x value 13.
  05 pic 9(4) comp-x value 12.
  05 pic 9(4) comp-x value 11.
* State 13
  05 pic 9(4) comp-x value 8.
  05 pic 9(4) comp-x value 13.
  05 pic 9(4) comp-x value 12.
  05 pic 9(4) comp-x value 9.
  05 pic 9(4) comp-x value 8.
  05 pic 9(4) comp-x value 13.
  05 pic 9(4) comp-x value 11.

01 w-state-transition-table redefines
                                w-state-transitions.
  05 w-new-state-list occurs 13 times.
  10 w-next-state pic 9(4) comp-x occurs 7 times.
* State transitions can be accomplished with lots of IF
* statements; the appeal of the table method is that
* selecting the next state is simply a matter of looking up
* w-next-state(w-state, w-action).
* Method nextState uses this table lookup method.

method-id. nextState.
linkage Section.
01 l-action pic 9(4) comp-x.

procedure division using l-action.
  if l-action >= 1 and l-action <= 7 then
    move w-next-state(w-current-state, l-action) to

```

```

        w-current-state.
exit method.
end method nextState.

method-id. getState.
linkage section.
01 l-state pic 9(4) comp-x.
procedure division returning l-state.
    move w-current-state to l-state.
    exit method.
end method getState.

method-id. setState.
linkage section.
01 l-state pic 9(4) comp-x.
procedure division using l-state.
    if l-state <= 0 and l-state > 13
        move 1 to w-current-state
    else
        move l-state to w-current-state.
    exit method.
end method setState.

method-id. clearToStart .
procedure division.
    move 1 to w-current-state.
    exit method.
end method clearToStart.

end object.

end class CreditState.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. TEST-CREDITSTATE.

ENVIRONMENT DIVISION.
REPOSITORY.

CLASS CreditState AS "creditstate".

* This separate program references CreditState; COBOL is
* informed that this class actually exists by placing it in
* the repository.

INPUT-OUTPUT SECTION.

FILE-CONTROL.

SELECT CUST-FILE ASSIGN TO "E:\CLASS\336\CUST.IND"
ORGANIZATION IS INDEXED
ACCESS IS RANDOM
RECORD KEY IS CF-ID
ALTERNATE KEY IS CF-NAME WITH DUPLICATES.
SELECT TRANS-FILE ASSIGN TO "E:\CLASS\336\TRANS.TXT"
ORGANIZATION IS LINE SEQUENTIAL.

DATA DIVISION.

FILE SECTION.

FD CUST-FILE.

01 CUST-REC.

05 CF-ID PIC X(7).
05 CF-NAME PIC X(20).
05 CF-BALANCE PIC 9(7)V99 COMP-3.
05 CF-CREDIT PIC 9(4) COMP-X.

FD TRANS-FILE.

01 TRANS-REC.

05 TF-ID PIC X(7).
05 TF-OBSERVATION PIC X(73).

WORKING-STORAGE SECTION.

01 W-CUST-REC.

05 W-ID PIC X(7).
05 W-NAME PIC X(20).
05 W-BALANCE PIC 9(7)V99 COMP-3.
05 W-CREDIT OBJECT REFERENCE.

* W-CREDIT will refer to an instance of class CreditState.

01 W-EOF PIC X VALUE 'N'.
88 EOF VALUE 'Y'.
01 w-current-id pic x(7) value spaces.
01 W-MSG PIC X(20).
01 W-ACTION PIC 9(4) COMP-X.

PROCEDURE DIVISION.

PROCESS-CHANGES.

```
OPEN INPUT TRANS-FILE.
OPEN I-O CUST-FILE.
READ TRANS-FILE, AT END SET EOF TO TRUE.
PERFORM PROCESS-TRANSACTION UNTIL EOF.
CLOSE CUST-FILE, TRANS-FILE.
STOP RUN.
```

```
PROCESS-TRANSACTION.
```

```
MOVE TF-ID TO CF-ID, W-CURRENT-ID.
```

```
READ CUST-FILE,
```

```
INVALID KEY
```

```
DISPLAY "Unable to find ", CF-ID
```

```
NOT INVALID KEY
```

```
MOVE CUST-REC TO W-CUST-REC
```

```
* Create a CreditState object with its new method, then
* invoke object methods by referring to the object
* reference in w-credit.
```

```
invoke CreditState "new" returning w-credit
```

```
invoke w-credit "setState" using cf-credit
```

```
perform handle-specifics
```

```
until TF-ID NOT = W-CURRENT-ID.
```

```
HANDLE-SPECIFICS.
```

```
invoke CreditState "encode-action"
```

```
using TF-OBSERVATION
```

```
RETURNING W-ACTION.
```

```
INVOKE w-credit "nextState" USING W-ACTION.
```

```
READ TRANS-FILE, AT END SET EOF TO TRUE
```

```
NOT AT END
```

```
IF W-CURRENT-ID NOT = TF-ID THEN
```

```
PERFORM UPDATE-CUSTOMER.
```

```
UPDATE-CUSTOMER.
```

```
MOVE W-CUST-REC TO CUST-REC.
```

```
INVOKE W-CREDIT "getState" returning CF-CREDIT.
```

```
invoke CreditState "get-state-description"
```

```
using cf-credit, w-msg.
```

```
display cf-id, " ", cf-name, " ", w-msg.
```