Object Oriented Programming with Progress 4GL

Scott Auge

Definition of OOP

- A type of programming in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. In this way, the data structure becomes an *object* that includes both data and functions. In addition, programmers can create relationships between one object and another. For example, objects can *inherit* characteristics from other objects.
- One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify.

Example of an object

- Lets examine what would be taken to make an object representing a door.

The Door Object

- We know that a door has certain states:
  - Open
  - Closed
  - Locked
  - Unlocked

The Door Object

- These states would be stored in an attribute of the object, a variable of the object that is part of the object.
  - IsLocked = {Yes, No}
  - IsOpen = {Yes, No}

The Door Object

- We can do certain things with the door, certain actions.

The Door Object

- These actions are called methods. These are the things one can do with the door object.
  - Lock()
  - UnLock()
  - Open()
  - Close()

The Door Object

- Sometimes what we want to happen to the door needs to send some feedback. For example, trying to open() an already opened door. We create some attributes for that.
  - cErrorMsg = “Human friendly description”
  - iErrorCode = {Z}

The Door Object

- Recap of variables associated with the object.
  - iErrorCode
  - cErrorMsg
  - lsLocked
  - lsOpen

The Door Object

- Recap of the methods available
  - GetError()
  - Lock()
  - Unlock()
  - Open()
  - Close()

The Door Object

- What if we want to make two door objects? Do we need to make two sets of variables?
  - NO! An object has all it's variables pre-defined and scoped only to that object. All we need to do is make another "instance" of the object.

Implementing an object

- An object in the 4GL needs to be defined in a procedure file.
- The object's attributes would be local variables to the file.
- The object's methods would be internal procedures to the file.
- To make an "instance" of the object, one would run the file "persistently."

Implementing an object

- Object oriented files should begin with obj_ so that programmers know off the bat the file is object oriented.
  - obj_log.p is an object oriented log file management system.

Implementing an object

- The logging object is stored in a file
  - obj_log.p
- The log management object has some attributes associated to it:
  - The file name of the file it manages
  - The error messaging subsystem.
  - A variable to determine if the file open

Implementing an object

- The logging object has some methods available to it:
  - GetError()
  - GetLogFileName()
  - Init() [A “constructor”]
  - InitAppend() [A “constructor”]
  - Destroy() [A “destructor”]
  - DoWrtLogFile()
  - DoEraseLogFile()
  - DoCopyLogFile()
  - DoCloseFile()

Implementing an object

- A constructor is a method that is run on creation of an object. In more OO languages, this is automatically done based on the arguments provided.

Implementing an object

- The log management object has some “private” methods available to it. Private methods are methods that should only be called by the object it’s self:
  - These should be pre-pended with prv so programmers know they are private. Note the use of the PRIVATE keyword on the procedure definition.
  - prvAssignError()
    - Helps to ease management of the errors from other procedures.

Implementing an object

- Local variables to the object (attributes), will need their own Set*() and Get*() methods because the 4GL does not allow external programs to access them directly.

Creating an instance of the object

- Now that an object is defined, how do we use it?

Create an instance of an object

- Objects need to be referred through Progress 4GL handles. The code below says we are expecting to work with two objects:

```
/* Object instance handles */

DEFINE VARIABLE hobjlog_SystemLogFile AS HANDLE NO-UNDO.
DEFINE VARIABLE hobjlog_ProgrammerLogFile AS HANDLE NO-UNDO.

/* C++ Version */

ObjLog *hobjlog_SystemLogFilex;
ObjLog *hobjlog_ProgrammerLogFile;
```

Create an instance of an object.

- Once we have handles to manage our instances, we go ahead and make instances:

```
/* Create two instances of the Logging object */

RUN obj_log.p PERSISTENT SET hobjlog_SystemLogFile.
RUN obj_log.p PERSISTENT SET hobjlog_ProgrammerLogFile.

/* C++ Version */
Not needed!
```

Create an instance of an object

- Polymorphism is weak in the language (er, actually kind of non-existent.) You will need to choose the constructor manually.
- The 4GL does not let us call the constructor automatically like some languages do (C++ for example.) We need to call those separately:

```
/* Call their constructors */

RUN InitAppend IN hobjlog_SystemLogFile ("/tmp/systemlogfile.txt").
RUN InitAppend IN hobjlog_ProgrammerLogFile ("/tmp/programmerlogfile.txt").

/* C++ Version */
Hobjlog_ProgrammerLogFile = new ObjLog ("/tmp/programmerlogfile.txt");
```

Using an instance of an object

- Once an object has been allocated and it's constructor has been called, one can begin working it's methods:

```
/* Call into their "activity" methods */

RUN DoWrtLogFile IN hobjlog_SystemLogFile ("This should be in system log file.").
RUN DoWrtLogFile IN hobjlog_ProgrammerLogFile ("This should be in programmer log
    file.").

/* C++ Version */

hobjlog_SystemLogFile->DoWrtLogFile("This should be in system log file.");
hobjlog_ProgrammerLogFile->DoWrtLogFile("This should be in programmer log file.");
```

Using an object instance

- One of the good things about objects is that they can know about themselves. Here is a method that returns the log file the object is writing to:

```
/* Display the log files these objects are using */

RUN GetLogFileName IN hobjlog_SystemLogFile (OUTPUT c1).
RUN GetLogFileName IN hobjlog_ProgrammerLogFile (OUTPUT c2).

disp c1 c2.

/* C++ Version */

cout << hobjlog_SystemLogFile->GetLogFileName();
```

Destroying an object instance

- When you are finished with an object, you should call it's "destructor." Note you need to delete it's persistence handle.

```
/* Call their destructors */

RUN Destroy IN hobjlog_SystemLogFile.
DELETE WIDGET hobjlog_SystemLogFile.

RUN Destroy IN hobjlog_ProgrammerLogFile.
DELETE WIDGET hobjlog_ProgrammerLogFile.

/* C++ Version */

delete hobjlog_ProgrammerLogFile;
```

Passing Objects around

- One can pass objects to other routines by passing their handles to them.
- Handles can be converted into strings with the STRING() function, and returned to a handle type with the WIDGET-HANDLE() function.

Inheritance

There are two ways for inheritance to work.

3. Use super-procedures.
4. Create an instance of the base object inherited and call into it directly.

Each has it’s pro’s and con’s.

Inherit by calls

- Lets explore number 2. It is easier to understand.

Inherit by calls

- The Good
- Pretty easy to determine the flow of control.
- Pretty easy to manage different instances of parent objects because they are scoped to the child object by handle and managed by constructor and destructor calls.
- Pretty easy to write “over-ride” method. Just have your child do what it does, with/without running the parent’s method.

- The Bad
- If the parent has it’s parameters changed, all children need to be found and have their parameters changed.

Inherit by calls

- In your constructors, you create parent object(s) instance(s) of the child object, then call it's constructors in your child's constructor:

```
DEFINE VARIABLE hobj_Parent AS HANDLE NO-UNDO.  /* Parent object's handle */

/* Calling your init (constructor) will automatically instance a parent */

PROCEDURE Init:

  RUN obj_base.p PERSISTENT SET hobj_Parent. /* Create instance */
  RUN Init IN hobj_Parent.                   /* Call it's constructor */

END.
```

Inherit by calls

- Destroying parent objects of child object. In your child's destructor, be sure to call the destructors of inherited objects and clean up memory.

```
DEFINE VARIABLE hobj_Parent AS HANDLE NO-UNDO.

/* Calling your Destroy (destructor) will automatically delete instance a parent */
/* Call other instances if inheriting more than one object.                      */

PROCEDURE Destroy:

  RUN Destroy IN hobj_Parent.
  DELETE OBJECT hobj_Parent.

END.
```

Inherit by calls

- Using parent objects of child object. You will need to write a “wrapper” method that passes arguments from your child’s procedure into the parent’s procedure.

```
DEFINE VARIABLE hobj_Parent AS HANDLE NO-UNDO.

/* Call into parent object directly.  No over-riding code is included. */

PROCEDURE DoThis:

  RUN DoThis IN hobj_Parent.

END.
```

Inherit by calls

- This is an example of over-riding the method of a parent object in the child object. Simply don't call the parent's method (unless you need to!)

```
/* Call into parent object directly.  This is an over-ride of an existing */
/* parent function.                                                       */

PROCEDURE DoThat:

  /* Your own code.  Call parent object for what ever or not call the parent. */

END.
```

Inherit by Super-procedures

- The Good
- No need for “wrapper” procedures. Parent methods automatically made available.
- You can inherit functions.
- Multiple “vertical” object inheritance works great!

- The Bad
- One needs to be careful how Supers are stacked. Less control over which parent object’s method is called if two parents with the same method. No “switcharoo’s” determining order of parent methods called.
- Doesn’t handle “horizontal” objects very well, that is an object of two base objects with the same method. MethodA might need one super-procedure stack order while MethodB needs another.
- Gotta start up all those parents and know you need to start em up.

Inherit by Super Procedures

- Very similar to the inherit by call method.
- See the progress Expert Series book:

OpenEdge
Development: Progress 4GL Handbook, b
Sadd

Questions?

Corrections and contributions

- Tim Townsend, www.tttechno.com
- Colin Stutley, ws.com.au

Possible way to automate destructors

```
/* ... Main Block */
 ON CLOSE OF THIS-PROCEDURE RUN ipDispose IN THIS-PROCEDURE.

IF p_OptionalParameter = ? THEN RUN ipConstructor IN THIS-PROCEDURE().
ELSE RUN ipConstructor2 IN THIS-PROCEDURE(p_OptionalParameter).
RETURN.

Then in any invoking procedure ...

RUN xyz.r PERSISTENT SET vObjectHandle(?). ...

... APPLY "CLOSE":U TO vObjectHandle. /* defacto garbage collection */
```