

# 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 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.
  - cErrorMsg = “Human friendly description”
  - iErrorCode = {Z}



# The Door Object

- Recap of variables associated with the object.
  - iErrorCode
  - cErrorMsg
  - IsLocked
  - IsOpen

# 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 its 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 "persistantly."

# 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

- 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. Progress does not enforce privacy.
  - 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.
```

# 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.
```

# Create an instance of an object

- 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").
```

# Create an instance of an object

- Once an object has been allocated and its constructor has been called, one can begin working its 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.").
```

# Creating 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.
```

# Creating an object instance

- When you are finished with an object, you should call its “destructor.” Note you need to delete its persistence handle.

```
/* Call their destructors */
```

```
RUN Destroy IN hobjlog_SystemLogFile.  
DELETE WIDGET hobjlog_SystemLogFile.
```

```
RUN Destroy IN hobjlog_ProgrammerLogFile.  
DELETE WIDGET 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.

# Some things to think about

- Objectfying LT2K
  - An Application Object
    - SetApplicant(hperson\_main, “Main”)
  - A Person Object
    - SetName(FirstName, LastName)
  - A Property Object
    - GetFees()

Questions?