John Sadd
Fellow and OpenEdge Evangelist
Document Version 1.0
July 2010

DISCLAIMER

This document accompanies the video series on ***Programming with ABL Classes in OpenEdge 10***. The paper summarizes the content of the video sessions, and provides complete code samples used in the videos. It does not duplicate all the spoken text of the videos, which should be viewed to get a more complete understanding of what the code samples show.

The two-part session ***Introduction to Class Syntax in ABL*** starts by showing a simple ABL procedure that can be run as a persistent procedure, with internal procedures and user-defined functions.

```
/*------------------------------------------------------------------------
    File        : SampleProc.p

    Description : Sample procedure to use to compare procedural coding to
                  class-based
  ----------------------------------------------------------------------*/
```

The Definitions section includes definitions for the temp-table, a ProDataSet, a Data-Source for the ProDataSet, a static query, and an ordinary variable.

```
/* ************************  Definitions  ********************** */

DEFINE TEMP-TABLE ttEmployee
    FIELD EmployeeID          AS CHARACTER
    FIELD EmployeeFirstName   AS CHARACTER
    FIELD EmployeeLastName    AS CHARACTER
    FIELD EmployeePosition    AS CHARACTER
    FIELD EmployeeStartDate   AS DATE
    FIELD EmployeeNotes       AS CHARACTER
    FIELD EmployeeBirthCountry AS CHARACTER
    FIELD EmployeeGender      AS CHARACTER.

DEFINE DATASET     dsEmployee  FOR ttEmployee.
DEFINE DATA-SOURCE srcEmployee FOR AutoEdge.Employee.

DEFINE QUERY qEmployee FOR ttEmployee.

DEFINE VARIABLE cAssignedCountry AS CHARACTER NO-UNDO.
```

Since the procedure contains user-defined functions, these require function prototypes at the head of the procedure.

```
/* ***********************  Function Prototypes ********************** */

FUNCTION AssignCountry RETURNS INTEGER
        ( INPUT pcCountry AS CHARACTER ) FORWARD.

FUNCTION GetEmployeeCount RETURNS INTEGER
        (  ) FORWARD.
```

The Main Block contains code that gets executed when the procedure starts up, which fills the temp-table with all the employees in the AutoEdge database.

```
/* ***************************  Main Block  *************************** */

BUFFER ttEmployee:ATTACH-DATA-SOURCE (DATA-SOURCE srcEmployee:HANDLE ).
DATASET dsEmployee:FILL ().   /* All 18 Employees */
OPEN QUERY qEmployee PRESELECT EACH ttEmployee.
```

The Main Block also contains an **ON CLOSE** block of code which will do cleanup if the calling procedure applies the **CLOSE** event to the persistent procedure handle.

```
ON CLOSE OF THIS-PROCEDURE
DO:
   CLOSE QUERY qEmployee.
   EMPTY TEMP-TABLE ttEmployee.
   DELETE PROCEDURE THIS-PROCEDURE.
END.
```

There is one internal procedure, which sets the **EmployeeNotes** value for all employees matching the **EmployeePosition** value passed in.

```
/* **********************  Internal Procedures  *********************** */

PROCEDURE InitializeNotes:

/*------------------------------------------------------------------------------
Purpose:       Assign a value to all empty EmployeeNotes.
Notes:          Done for a specific EmployeePosition.

------------------------------------------------------------------------------*/

DEFINE INPUT PARAMETER pcPosition AS CHARACTER.

    FOR EACH ttEmployee WHERE ttEmployee.EmployeePosition = pcPosition:
        ttEmployee.EmployeeNotes = "Initial note for this " + pcPosition.
    END.
END PROCEDURE.
```

The first user-defined function assigns a **BirthCountry** value to all employees that don't have one, and keeps a count of the number of rows modified.

```
/* ************************  Function Implementations ***************** */

FUNCTION AssignCountry RETURNS INTEGER
            ( INPUT pcCountry AS CHARACTER ):

/*------------------------------------------------------------------------------
Purpose: Assigns a value to all blank EmployeeCountry fields.
Notes:   Returns the number of Employees assigned.

------------------------------------------------------------------------------*/

        DEFINE VARIABLE iCount AS INTEGER NO-UNDO.

        FOR EACH ttEmployee WHERE ttEmployee.EmployeeBirthCountry = "":
            ttEmployee.EmployeeBirthCountry = pcCountry.
            iCount = iCount + 1.
         END.

         cAssignedCountry = pcCountry.

        RETURN iCount.

END FUNCTION.
```

The second function returns the count of modified employee rows to the caller.

```
FUNCTION GetEmployeeCount RETURNS INTEGER
            (  ):

/*------------------------------------------------------------------------------
Purpose:       Return the total number of employee in the temp-table.

------------------------------------------------------------------------------*/

        RETURN QUERY qEmployee:NUM-RESULTS.

END FUNCTION.
```

The persistent procedure is then converted into a class with the equivalent behavior.

```
/*------------------------------------------------------------------------
    File        : SampleClass.cls
    Description : Sample class to use to compare procedural coding to class-based

    Author(s)   : john
    Created     : Wed Feb 17 14:53:32 EST 2010
    Notes       :
  ----------------------------------------------------------------------*/
```

The **CLASS** statement identifies the name of the class and its location – the *package* – to the compiler:

```
CLASS OOSamples.SampleClass:
```

A class can have a Definitions section just as a procedure can. In addition, variables in the Main Block's definitions can be made **PUBLIC**, **PRIVATE**, or **PROTECTED**, and are referred to as *data members* to emphasize these special characteristics.

```
/* ************************  Definitions  ********************** */

DEFINE TEMP-TABLE ttEmployee
    FIELD EmployeeID           AS CHARACTER
    FIELD EmployeeFirstName    AS CHARACTER
    FIELD EmployeeLastName     AS CHARACTER
    FIELD EmployeePosition     AS CHARACTER
    FIELD EmployeeStartDate    AS DATE
    FIELD EmployeeNotes        AS CHARACTER
    FIELD EmployeeBirthCountry AS CHARACTER
    FIELD EmployeeGender       AS CHARACTER.

DEFINE DATASET     dsEmployee  FOR ttEmployee.
DEFINE DATA-SOURCE srcEmployee FOR AutoEdge.Employee.

DEFINE QUERY qEmployee FOR ttEmployee.

DEFINE VARIABLE mcAssignedCountry AS CHARACTER NO-UNDO.
```

A function that returns a value can be turned into a property of a class, which allows it to be referenced from another procedure or class as if it were a simple variable, but with supporting blocks of code to get or set the property value.

```
DEFINE PUBLIC PROPERTY EmployeeCount AS INTEGER
    GET():
        RETURN QUERY qEmployee:NUM-RESULTS.
    END GET.
    PRIVATE SET.
```

The Main Block of a class can't have executable statements. Startup code goes into the class's constructor, a special method with the same name as the class.

```
CONSTRUCTOR PUBLIC SampleClass():
    BUFFER ttEmployee:ATTACH-DATA-SOURCE (DATA-SOURCE srcEmployee:HANDLE ).
    DATASET dsEmployee:FILL ().   /* All 18 Employees */
    OPEN QUERY qEmployee PRESELECT EACH ttEmployee.
END.
```

Cleanup code can go into the destructor, which is reliably executed when the running class instance is deleted.

```
DESTRUCTOR PUBLIC SampleClass():
   CLOSE QUERY qEmployee.
   EMPTY TEMP-TABLE ttEmployee.
END.
```

An internal procedure in a procedure becomes a method in a class with a return type of **VOID**.

```
METHOD PUBLIC VOID InitializeNotes ( INPUT pcPosition AS CHARACTER ):

/*------------------------------------------------------------------------------
Purpose:       Assign a value to all empty EmployeeNotes.
Notes:          Done for a specific EmployeePosition.

------------------------------------------------------------------------------*/

    FOR EACH ttEmployee WHERE ttEmployee.EmployeePosition = pcPosition:
        ttEmployee.EmployeeNotes = "Initial note for this " + pcPosition.
    END.
END METHOD.
```

A user-defined function in a procedure becomes a method with a return type in a class.

```
METHOD PUBLIC INTEGER AssignCountry  ( INPUT pcCountry AS CHARACTER ):

/*------------------------------------------------------------------------------
        Purpose: Assigns a value to all blank EmployeeCountry fields.

        Notes:   Returns the number of Employees assigned.

------------------------------------------------------------------------------*/

        DEFINE VARIABLE iCount AS INTEGER NO-UNDO.

        FOR EACH ttEmployee WHERE ttEmployee.EmployeeBirthCountry = "":
            ttEmployee.EmployeeBirthCountry = pcCountry.
            iCount = iCount + 1.
         END.

         mcAssignedCountry = pcCountry.

        RETURN iCount.

END METHOD.
```

The class ends with an **END CLASS** statement to balance the **CLASS** header statement.

```
END CLASS.
```

The presentation then shows a wrapper procedure to run an instance of the persistent procedure and invoke the internal procedure and the functions it contains. Any errors in the run statements are not detected until runtime, because the compiler does not cross-check the validity of calls to other procedures.

```
/*------------------------------------------------------------------------
    File        : SampleProcRunner.p
    Purpose     : Wrapper procedure to run SampleProc.p

------------------------------------------------------------------------*/

DEFINE VARIABLE iEmployeeCount AS INTEGER NO-UNDO.
DEFINE VARIABLE iCountryCount  AS INTEGER NO-UNDO.
DEFINE VARIABLE hSampleProc    AS HANDLE  NO-UNDO.

RUN OOSamples/SampleProc.p PERSISTENT SET hSampleProc.

RUN InitializeNotes IN hSampleProc (INPUT "Admin").

iEmployeeCount = DYNAMIC-FUNCTION ("GetEmployeeCount" IN hSampleProc).

iCountryCount  = DYNAMIC-FUNCTION ("AssignCountry" IN hSampleProc,
  INPUT "France").

DELETE PROCEDURE hSampleProc.

MESSAGE "There are" iEmployeeCount "employees, of which" SKIP
        iCountryCount "have just been assigned to France."
        VIEW-AS ALERT-BOX.
```

A similar wrapper procedure can create an instance of the class and invoke its methods and reference its properties. Any errors in references to the class are detected by the compiler, because the variable that holds the reference to the class

instance names the class type. This is referred to as *strong typing*, and allows the compiler to cross-check to make sure that references to the other classes are valid at compile time.

```
/*------------------------------------------------------------------------
    File        : SampleClassRunner.p
    Purpose     : Wrapper procedure to run SampleClass.cls

  ----------------------------------------------------------------------*/

DEFINE VARIABLE miEmployeeCount AS INTEGER NO-UNDO.
DEFINE VARIABLE miCountryCount  AS INTEGER NO-UNDO.
DEFINE VARIABLE moSampleClass   AS OOSamples.SampleClass NO-UNDO.

moSampleClass = NEW OOSamples.SampleClass().

moSampleClass:InitializeNotes(INPUT "Admin").

miEmployeeCount = moSampleClass:EmployeeCount.

miCountryCount  = moSampleClass:AssignCountry (INPUT "France").


DELETE OBJECT moSampleClass.

MESSAGE "There are" miEmployeeCount "employees, of which" SKIP
        miCountryCount "have just been assigned to France."
        VIEW-AS ALERT-BOX.
```

The presentation on ***Using Interfaces as a Programming Contract*** creates an interface for a set of classes that do data management.

```
/*------------------------------------------------------------------------
    File        : IDataManager

  ----------------------------------------------------------------------*/

USING Progress.Lang.*.

INTERFACE OOSamples.IDataManager:
    METHOD PUBLIC HANDLE FetchData (INPUT pcFilter AS CHARACTER ).
    DEFINE PUBLIC PROPERTY RowCount AS INTEGER GET.

END INTERFACE.
```

The interface acts as a contract. A class that **IMPLEMENTS** the interface must adhere to the contract, and this is verified by the compiler.

```
 /*------------------------------------------------------------------------
    File        : EmployeeManager

  ----------------------------------------------------------------------*/

USING Progress.Lang.*.
USING OOSamples.IDataManager.

CLASS OOSamples.EmployeeManager IMPLEMENTS IDataManager:
```

EmployeeManager.cls implements the **PUBLIC** property defined in the interface, and extends it to specify that the property cannot be set by another class.

```
        DEFINE PUBLIC PROPERTY RowCount AS INTEGER
        GET.
        PRIVATE SET.
```

It defines a temp-table and a query specific to employees.

```
    DEFINE TEMP-TABLE ttEmployee
        FIELD EmployeeFirstName AS CHARACTER
        FIELD EmployeeLastName  AS CHARACTER
        FIELD EmployeePosition  AS CHARACTER.

    DEFINE QUERY qEmployee FOR AutoEdge.Employee.
```

The default constructor runs a constructor in this class's super class, if there is one.

```
        CONSTRUCTOR PUBLIC EmployeeManager (  ):
               SUPER ().

        END CONSTRUCTOR.
```

EmployeeManager implements **FetchData**, populating its temp-table with rows from the **Employee** table, and setting the **RowCount** property.

```
    METHOD PUBLIC HANDLE FetchData( INPUT pcFilter AS CHARACTER ):

        DEFINE VARIABLE hQuery    AS HANDLE NO-UNDO.

        hQuery = QUERY qEmployee:HANDLE.
        hQuery:QUERY-PREPARE("FOR EACH AutoEdge.Employee WHERE " + pcFilter).
        hQuery:QUERY-OPEN ().
        hQuery:GET-FIRST ().
        DO WHILE NOT hQuery:QUERY-OFF-END:
            CREATE ttEmployee.
            BUFFER-COPY AutoEdge.Employee TO ttEmployee.
            hQuery:GET-NEXT ().
        END.
        RowCount = hQuery:NUM-RESULTS.
        hQuery:QUERY-CLOSE ().
        RETURN TEMP-TABLE ttEmployee:HANDLE.

    END METHOD.
```

EmployeeManager can also have additional methods or properties that are not specified by the interface, such as this one:

```
    METHOD PUBLIC VOID InitializeNotes ( INPUT pcPosition AS CHARACTER ):

    /*------------------------------------------------------------------------------
            Purpose:        Assign a value to all empty EmployeeNotes.
            Notes:          Done for a specific EmployeePosition.
    ------------------------------------------------------------------------------*/

        FOR EACH AutoEdge.Employee WHERE Employee.EmployeePosition = pcPosition:
            Employee.EmployeeNotes = "Initial note for this " + pcPosition.
        END.
    END METHOD.
```

The default destructor and **END CLASS** statement end the class definition for **EmployeeManager**.

```
        DESTRUCTOR PUBLIC EmployeeManager ( ):

        END DESTRUCTOR.

END CLASS.
```

EmployeeClient.cls serves as a test class that creates an instance of **EmployeeManager** and invokes its methods.

```
/*------------------------------------------------------------------------
    File        : EmployeeClient

  ----------------------------------------------------------------------*/

USING Progress.Lang.*.

CLASS OOSamples.EmployeeClient:
```

The variable **oEmpManager** holds the object reference to the instance of **EmployeeManager**. If this class needs to invoke **InitializeNotes**, which is not part of the **IDataManager** interface, then it must define **oEmpManager** as of type **EmployeeManager**. Otherwise it can define the variable as **IDataManager**.

```
    DEFINE PRIVATE VARIABLE oEmpManager AS OOSamples.EmployeeManager.
```

The constructor invokes the class's one method, **FetchEmployees**:

```
    DEFINE PRIVATE VARIABLE cFilter     AS CHARACTER NO-UNDO.
    DEFINE PRIVATE VARIABLE hEmpTable   AS HANDLE    NO-UNDO.

        CONSTRUCTOR PUBLIC EmployeeClient (  ):
                SUPER ().
                FetchEmployees().
        END CONSTRUCTOR.
```

Its **FetchEmployees** method creates an instance of **EmployeeManager**, executes the **InitializeNotes** and **FetchData** methods, and displays a message that uses the **RowCount** property.

```
METHOD PRIVATE VOID FetchEmployees(  ):

        DEFINE VARIABLE hEmpBuffer AS HANDLE NO-UNDO.

        oEmpManager = NEW OOSamples.EmployeeManagerAlt().
        oEmpManager:InitializeNotes("SalesRep").
        cFilter = "EmployeePosition = 'SalesRep' ".
        hEmpTable = oEmpManager:FetchData(INPUT cFilter).
        hEmpBuffer = hEmpTable:DEFAULT-BUFFER-HANDLE.
        hEmpBuffer:FIND-FIRST ().
        MESSAGE " Employee "
           hEmpBuffer:BUFFER-FIELD ("EmployeeFirstName"):BUFFER-VALUE
           hEmpBuffer:BUFFER-FIELD ("EmployeeLastName"):BUFFER-VALUE
           " is the first of " oEmpManager:RowCount " employees with "
           cFilter VIEW-AS ALERT-BOX.
        RETURN.
END METHOD.
```

The two-part session on ***Inheritance and Super Classes*** shows how to refactor the **EmployeeManager** and **CustomerManager** classes to extract common code from them and move it into a common super class, called **DataManager.cls**.

```
/*------------------------------------------------------------------------
    File        : DataManager

  ----------------------------------------------------------------------*/

USING Progress.Lang.*.

CLASS OOSamples.DataManager:
```

The **RowCount** property is common to both employee and customer management, so it is moved to the super class.

```
DEFINE PUBLIC PROPERTY RowCount AS INTEGER
GET.
PRIVATE SET.
```

There are two data values that are used in the super class but must be set in the respective subclass: the buffer handle of the database table that data is retrieved from, and the handle of the temp-table buffer that data is copied into. To make these values available to the super class to use, and to the subclass to set, they are defined in the super class as protected properties.

```
DEFINE PROTECTED PROPERTY DB_BufferHandle AS HANDLE
GET.
SET.

DEFINE PROTECTED PROPERTY TT_BufferHandle AS HANDLE
GET.
SET.
```

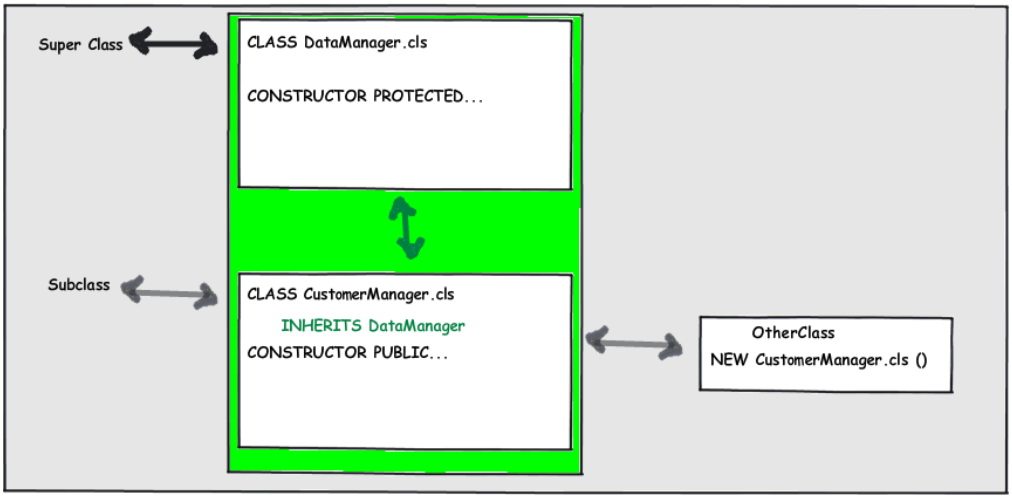
Almost all the code from the **FetchData** method is moved to the super class, with adjustments to make its references to the database and temp-table buffers dynamic:

```
   METHOD PROTECTED HANDLE FetchData( INPUT pcFilter AS CHARACTER ):
        DEFINE VARIABLE hQuery    AS HANDLE NO-UNDO.

        CREATE QUERY hQuery.
        hQuery:SET-BUFFERS (THIS-OBJECT:DB_BufferHandle).
        hQuery:QUERY-PREPARE (pcFilter).
        hQuery:QUERY-OPEN ().
        hQuery:GET-FIRST ().
        DO WHILE NOT hQuery:QUERY-OFF-END:
            TT_BufferHandle:BUFFER-CREATE ().
            TT_BufferHandle:BUFFER-COPY (DB_BufferHandle ).
            hQuery:GET-NEXT ().
        END.
        RowCount = hQuery:NUM-RESULTS.
        hQuery:QUERY-CLOSE ().
        DELETE OBJECT hQuery.
        RETURN TT_BufferHandle.
    END METHOD.
```

Most of the executable code can then be removed from **CustomerManager.cls**, which is now a subclass of **DataManager.cls**. The **INHERITS** phrase tells the compiler that this subclass inherits common behavior from **DataManager.cls**.

```
/*------------------------------------------------------------------------
    File        : CustomerManager

  ----------------------------------------------------------------------*/

USING Progress.Lang.*.
USING OOSamples.IDataManager.

CLASS OOSamples.CustomerManager
    INHERITS OOSamples.DataManager IMPLEMENTS IDataManager:
```

Because the super class constructor is **PROTECTED**, some other class outside its class hierarchy cannot create an instance of **DataManager** directly ("NEW" it). However, when another class creates a new **CustomerManager**, an instance of **DataManager** is also created, and both constructors executed:



The static temp-table definition is specific to the table being managed, so that stays in **CustomerManager.cls**:

```
    DEFINE TEMP-TABLE ttCustomer
        FIELD CustomerFirstName AS CHARACTER
        FIELD CustomerLastName  AS CHARACTER
        FIELD CustomerBirthCountry  AS CHARACTER.
```

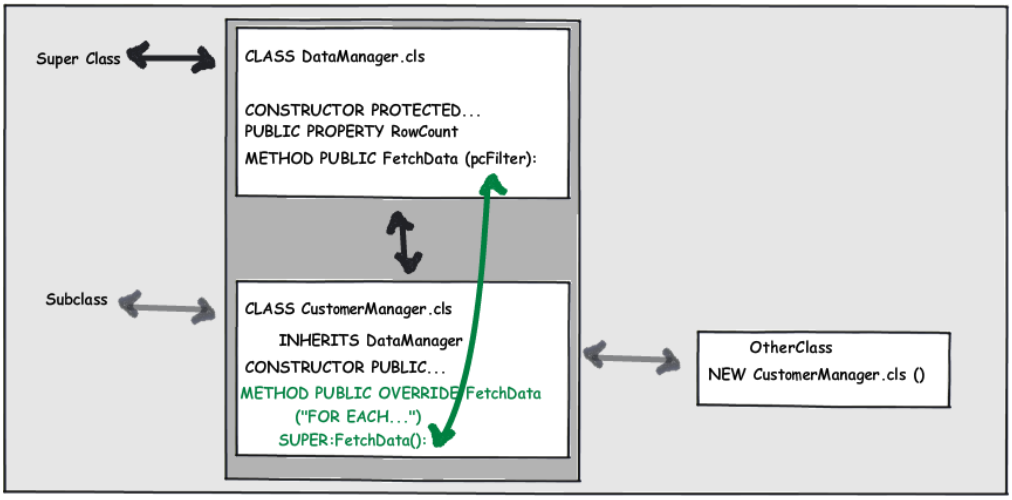
The subclass constructor sets the values of the two protected properties that the super class uses:

```
        CONSTRUCTOR PUBLIC CustomerManager (  ):
                SUPER ().
                DB_BufferHdl = BUFFER AutoEdge.Customer:HANDLE.
                TT_BufferHdl = BUFFER ttCustomer:HANDLE.
        END CONSTRUCTOR.
```

The subclass's **FetchData** method now has almost no code if its own. It defines itself as being an **OVERRIDE** of a method with the same name in its super class, and then invokes the common behavior in the super class.

```
METHOD PUBLIC OVERRIDE HANDLE FetchData( INPUT pcFilter AS CHARACTER ):

    SUPER:FetchData("FOR EACH AutoEdge.Customer WHERE " + pcFilter).
     RETURN TEMP-TABLE ttCustomer:HANDLE.

END METHOD.
```

The subclass implementation of **FetchData** can now invoke the common code in its super class, so that together they provide a complete implementation of its behavior:



The destructor is still there as a place to put any cleanup code required by **CustomerManager.cls**. If there is none, it could be dispensed with.

```
    DESTRUCTOR PUBLIC CustomerManager ( ):

     END DESTRUCTOR.
END CLASS.
```

In the test class **CustomerClient.cls**, which creates an instance of **CustomerManager.cls**, invokes its **FetchData** method, and references its **RowCount** property, nothing changes. It is unaware that most of the code for **FetchData** is now in a super class, and that the property **RowCount** is defined in that super class. Factoring out common code into a class hierarchy is transparent to all other classes, which instantiate and reference only the subclass.

```
    METHOD PRIVATE VOID FetchCustomers(  ):

        DEFINE VARIABLE hBuffer AS HANDLE NO-UNDO.

        oManager = NEW OOSamples.CustomerManager().
        cFilter = "CustomerBirthCountry = 'USA' ".
        hTable = oManager:FetchData(INPUT cFilter).
        hBuffer = hTable:DEFAULT-BUFFER-HANDLE.
        hBuffer:FIND-FIRST ().
        MESSAGE " Customer "
           hBuffer:BUFFER-FIELD ("CustomerFirstName"):BUFFER-VALUE
           hBuffer:BUFFER-FIELD ("CustomerLastName"):BUFFER-VALUE
           " is the first of " oManager:RowCount " Customers with "
           cFilter VIEW-AS ALERT-BOX.
        RETURN.

    END METHOD.
```

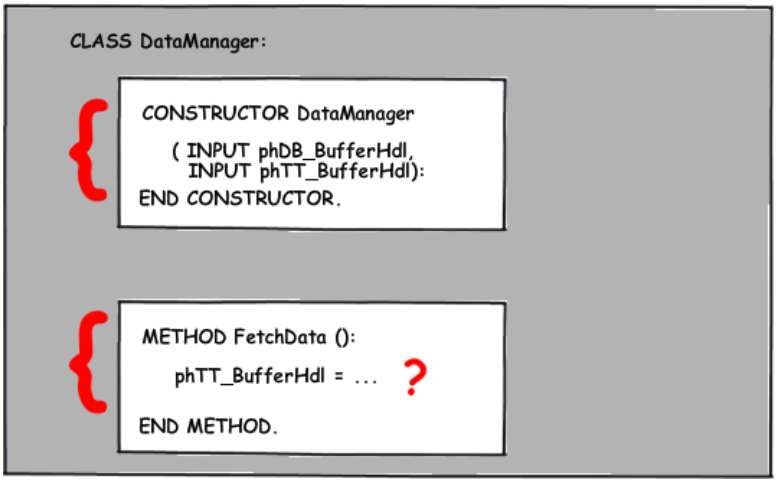
The same changes can be made to **EmployeeManager.cls** so that it now acts as another subclass of **DataManager.cls**. All the common code is now in one place.

```
/*------------------------------------------------------------------------
    File        : EmployeeManager
  ----------------------------------------------------------------------*/

USING Progress.Lang.*.
USING OOSamples.IDataManager.

CLASS OOSamples.EmployeeManager
    INHERITS OOSamples.DataManager IMPLEMENTS IDataManager:

    DEFINE TEMP-TABLE ttEmployee
        FIELD EmployeeFirstName AS CHARACTER
        FIELD EmployeeLastName  AS CHARACTER
        FIELD EmployeePosition  AS CHARACTER.

       CONSTRUCTOR PUBLIC EmployeeManager (  ):
              SUPER ().
              DB_BufferHdl = BUFFER AutoEdge.Employee:HANDLE.
              TT_BufferHdl = BUFFER ttEmployee:HANDLE.
       END CONSTRUCTOR.

       METHOD PUBLIC OVERRIDE HANDLE FetchData( INPUT pcFilter AS CHARACTER ):

           SUPER:FetchData("FOR EACH AutoEdge.Employee WHERE " + pcFilter).
           RETURN TEMP-TABLE ttEmployee:HANDLE.

       END METHOD.

    METHOD PUBLIC VOID InitializeNotes ( INPUT pcPosition AS CHARACTER ):

        FOR EACH AutoEdge.Employee WHERE Employee.EmployeePosition = pcPosition:
            Employee.EmployeeNotes = "Initial note for this " + pcPosition.
        END.
    END METHOD.
```

The presentation on ***Encapsulation and Overloading*** shows an alternative way to make the two buffer handles available to the super class. It defines them as **PRIVATE** variables (data members) in the main block of **DataManager.cls**, and accepts the values from the subclass as new parameters to its constructor:

```
DEFINE PRIVATE VARIABLE hDB_BufferHdl AS HANDLE NO-UNDO.
DEFINE PRIVATE VARIABLE hTT_BufferHdl AS HANDLE NO-UNDO.

        CONSTRUCTOR PROTECTED DataManager
            ( INPUT phDB_BufferHdl AS HANDLE,
              INPUT phTT_BufferHdl AS HANDLE ):
            SUPER ().
            ASSIGN hDB_BufferHdl = phDB_BufferHdl
                   hTT_BufferHdl = phTT_BufferHdl.
        END CONSTRUCTOR.
```

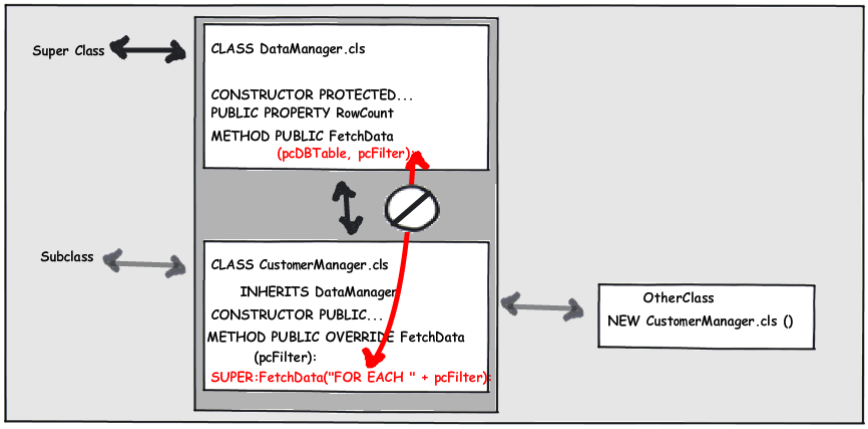
The parameters to the constructor must be assigned to data members in the main block, because just like any other parameters or variables local to a method, the constructor parameters cannot be accessed outside the constructor:



In addition, the **FetchData** method in **DataManager.cls** is changed to turn it into an overloaded method rather than an overridden one. This allows its signature to change so that it accepts two parameters rather than one, which are combined to form the query prepare string. Its return type is changed to **VOID**, since the subclass that invokes it does not need to get back the handle that it previously returned. And because the remaining **FetchData** method in the subclass is the one that must adhere to the **PUBLIC** signature declared in the interface **IDataManager**, the modified version in the super class is made **PROTECTED**:

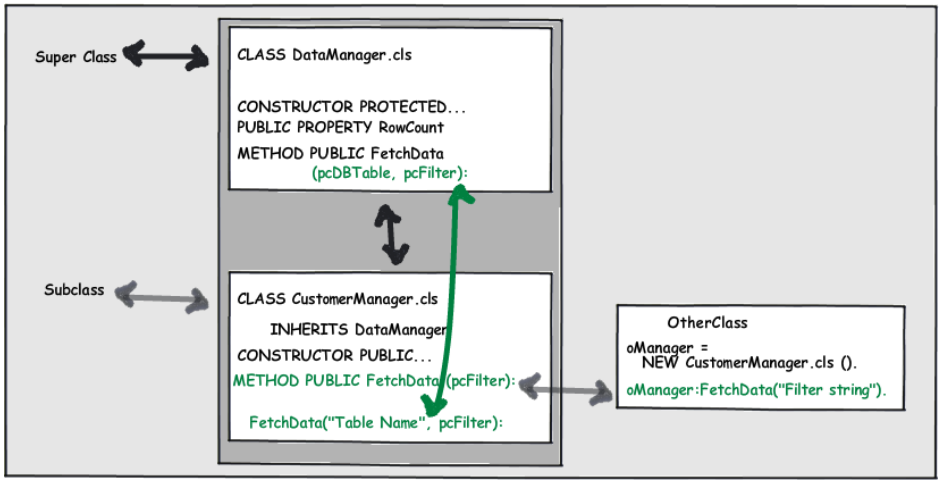
```
    METHOD PROTECTED VOID FetchData
        (INPUT pcDBTable AS CHARACTER ,
         INPUT pcFilter AS CHARACTER ):

        DEFINE VARIABLE hQuery AS HANDLE NO-UNDO.
        CREATE QUERY hQuery.
        hQuery:SET-BUFFERS (hDB_BufferHdl).
        hQuery:QUERY-PREPARE( "FOR EACH " + pcDBTable + " WHERE " + pcFilter).
        hQuery:QUERY-OPEN ().
        hQuery:GET-FIRST ().
        DO WHILE NOT hQuery:QUERY-OFF-END:
            hTT_BufferHdl:BUFFER-CREATE ().
            hTT_BufferHdl:BUFFER-COPY (hDB_BufferHdl).
            hQuery:GET-NEXT ().
        END.
        RowCount = hQuery:NUM-RESULTS.
        hQuery:QUERY-CLOSE ().
        DELETE OBJECT hQuery.
    END METHOD.
```

The subclasses **CustomerManager.cls** (shown here) and **EmployeeManager.cls** are modifed to remove the **OVERRIDE** keyword and the **SUPER** reference when invoking the super class **FetchData**, since it is now an overloaded rather and an overridden method. It would be an error to define a subclass method as an **OVERRIDE** when its signature does not match the signature of the method being overridden.



The compiler distinguishes the two variants now by their differing signatures. The subclass now passes two arguments to the super class version of **FetchData** instead of one:

```
METHOD PUBLIC HANDLE FetchData( INPUT pcFilter AS CHARACTER ):

    FetchData("AutoEdge.Customer", pcFilter).
    RETURN TEMP-TABLE ttCustomer:HANDLE.

END METHOD.
```



In the final presentation on the ***Object Life Cycle***, **CustomerClient.cls** (shown here) and **EmployeeClient.cls** are modified to make their fetch methods **PUBLIC**, and to remove their invocation from the constructor. In this way **FetchCustomers** or **FetchEmployees** can be invoked independently of creating the client object itself.

Also, the filtering value is now passed in as a parameter, rather than being hardcoded, to allow different instances of the objects to retrieve different sets of data.

```
    CONSTRUCTOR PUBLIC CustomerClient (  ):
        SUPER ().
    END CONSTRUCTOR.


    METHOD PUBLIC VOID FetchCustomers(INPUT pcFilterValue AS CHARACTER ).

        DEFINE VARIABLE hBuffer  AS HANDLE NO-UNDO.
        DEFINE VARIABLE oManager AS OOSamples.IDataManager.
        DEFINE VARIABLE cFilter  AS CHARACTER NO-UNDO.
        DEFINE VARIABLE hTable   AS HANDLE    NO-UNDO.

        oManager = NEW OOSamples.CustomerManager().
        cFilter = "CustomerBirthCountry = '" + pcFilterValue + "'".
        hTable = oManager:FetchData(INPUT cFilter).
        hBuffer = hTable:DEFAULT-BUFFER-HANDLE.
        hBuffer:FIND-FIRST ().
        MESSAGE " Customer "
           hBuffer:BUFFER-FIELD ("CustomerFirstName"):BUFFER-VALUE
           hBuffer:BUFFER-FIELD ("CustomerLastName"):BUFFER-VALUE
           " is the first of " oManager:RowCount " Customers with "
           cFilter VIEW-AS ALERT-BOX.
        RETURN.

    END METHOD.
```

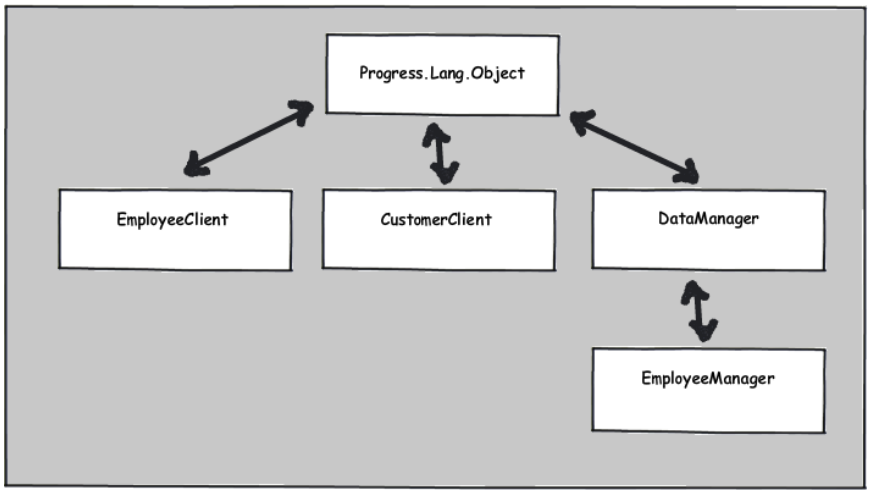
A new class **AutoEdgeManager** now creates instances of both **CustomerClient.cls** and **EmployeeClient.cls**.

```
/*------------------------------------------------------------------------
    File        : AutoEdgeManager

  ----------------------------------------------------------------------*/

USING Progress.Lang.*.

CLASS OOSamples.AutoEdgeManager:
```

The new class defines a temp-table to hold references to objects -- running instances of the other classes that it starts. **ObjectNum** is just a sequence number. **ObjectType** is the name of the client class invoked. **ObjectFilter** is the filter string passed in to qualify the **WHERE** clause applied to the data retrieved into each object. And finally, the **ObjectRef** field holds the object reference to each created (NEW'd) **EmployeeClient** or **CustomerClient** object. As a temp-table field, **ObjectRef** is required to be defined as the type **Progress.Lang.Object**, the virtual parent class for all ABL classes.

```
        DEFINE TEMP-TABLE ttObject
          FIELD ObjectNum AS INTEGER
          FIELD ObjectType AS CHARACTER
          FIELD ObjectFilter AS CHARACTER
          FIELD ObjectRef AS Progress.Lang.Object.
```

All classes are subclasses of **Progress.Lang.Object**, just as **EmployeeManager** is a subclass of **DataManager**:



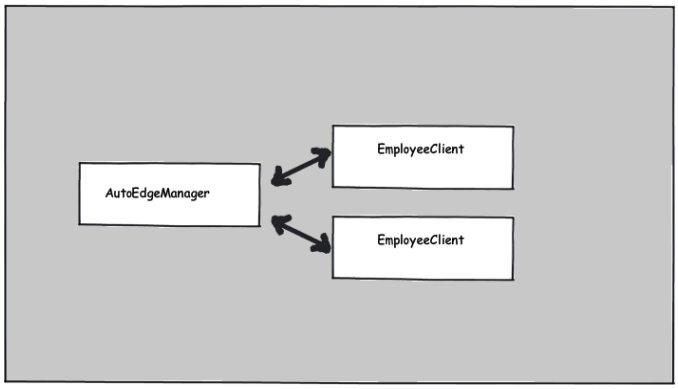
The constructor invokes the class's one method, **StartObjects**.

```
CONSTRUCTOR PUBLIC AutoEdgeManager (  ):
        SUPER ().
        StartObjects().
END CONSTRUCTOR.
```

StartObjects itself creates two instances of **EmployeeClient** and two of **CustomerClient**, which in turn create instances of their respective manager classes to retrieve data. The identifying information for each object, including its object reference, is added to the **ttObject** temp-table.

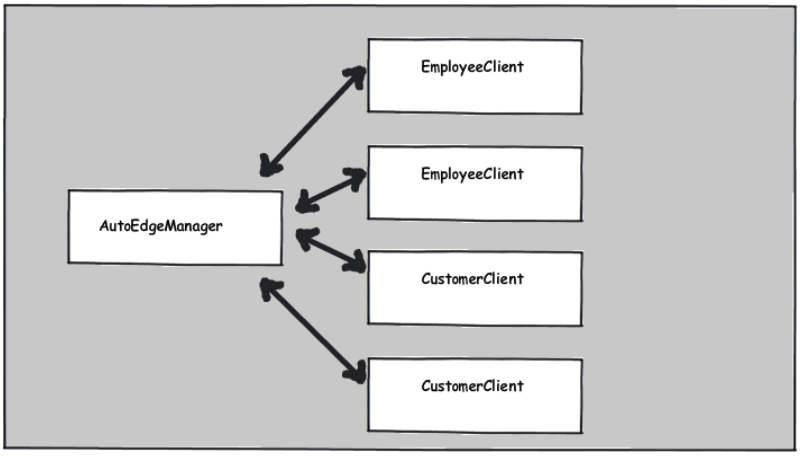
```
METHOD PRIVATE VOID StartObjects(  ):
    CREATE ttObject.
    ASSIGN ttObject.ObjectNum = 1
           ttObject.ObjectType = "EmployeeClient"
           ttObject.ObjectFilter = "SalesRep"
           ttObject.ObjectRef = NEW OOSamples.EmployeeClient().
    CREATE ttObject.
    ASSIGN ttObject.ObjectNum = 2
           ttObject.ObjectType = "EmployeeClient"
           ttObject.ObjectFilter = "Admin"
           ttObject.ObjectRef = NEW OOSamples.EmployeeClient().
```

After these first two groups of statements have been executed, these objects are now in the session:



After the next two sets of statements have been executed, these are now two CustomerClient objects as well as the two EmployeeClient objects:

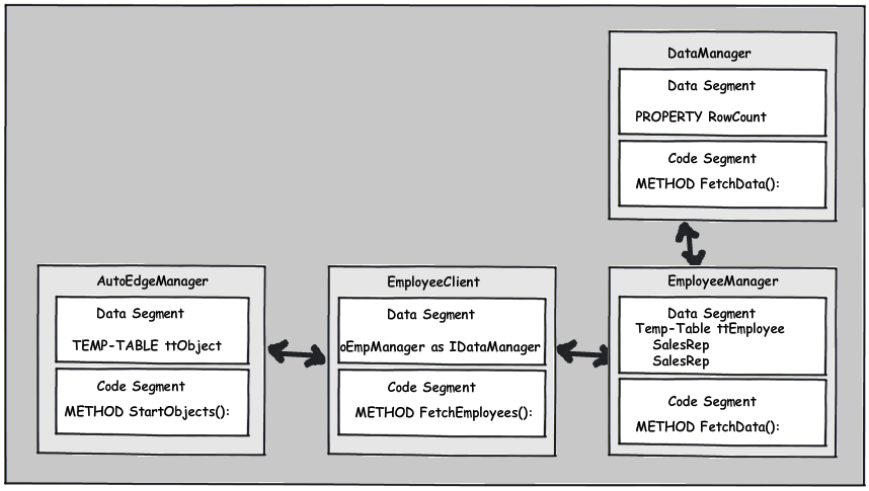
```
CREATE ttObject.
ASSIGN ttObject.ObjectNum = 3
       ttObject.ObjectType = "CustomerClient"
       ttObject.ObjectFilter = "USA"
       ttObject.ObjectRef = NEW OOSamples.CustomerClient().
CREATE ttObject.
ASSIGN ttObject.ObjectNum = 4
       ttObject.ObjectType = "CustomerClient"
       ttObject.ObjectFilter = "Germany"
       ttObject.ObjectRef = NEW OOSamples.CustomerClient().
```



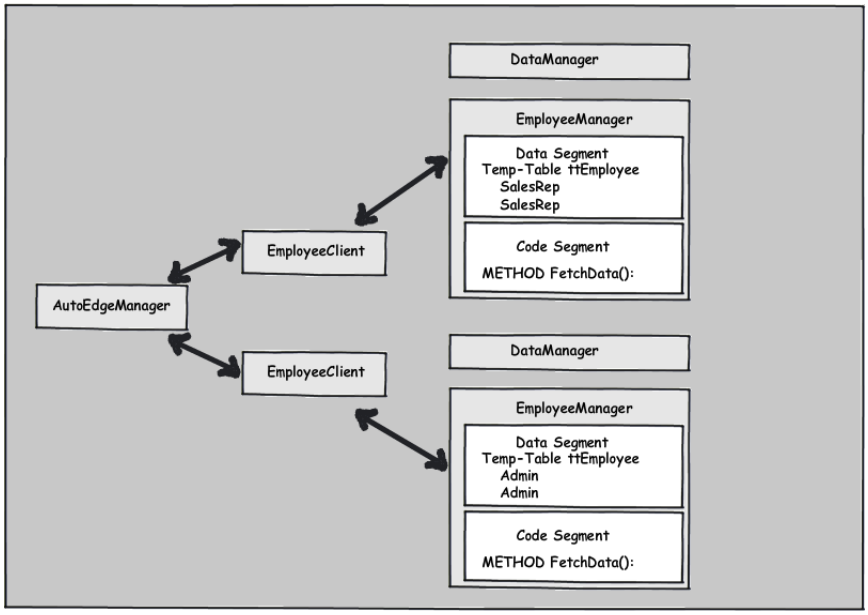
Next, for each row in the temp-table, the method invokes the correct fetch method. Because the **ObjectRef** field is of the generic type **Progress.Lang.Object**, the code must **CAST** the reference to the appropriate type to tell the compiler what type is actually being referenced. This tells the compiler to allow the type-specific methods **FetchEmployees** and **FetchCustomers** to be invoked.

```
FOR EACH ttObject:
    IF ttObject.ObjectType = "EmployeeClient" THEN
        CAST (ttObject.ObjectRef,
           OOSamples.EmployeeClient):FetchEmployees(ttObject.ObjectFilter).
    ELSE IF ttObject.ObjectType = "CustomerClient" THEN
        CAST (ttObject.ObjectRef,
           OOSamples.CustomerClient):FetchCustomers(ttObject.ObjectFilter).
END.
```

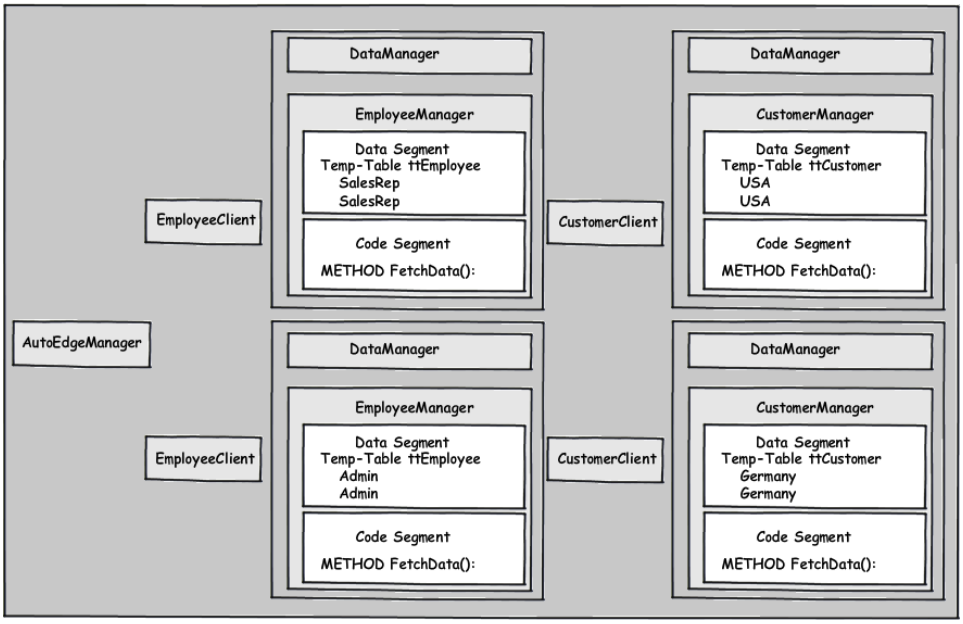
After the first iteration of the **FOR EACH** loop, the **EmployeeClient** object has created a new **EmployeeManager**. The AVM transparently creates an instance of **DataManager** as well, since that is **EmployeeManager's** super class and forms parts of its behavior and data.



After the second iteration of the loop, there are two instances of **EmployeeManager**, each with its own instance of **DataManager** to support it. Runtime optimization may eliminate the duplication of the code segments of these objects, but each has its own data, for instance the different contents of the two **ttEmployee** tables, resulting from two different filter values being passed in to them.



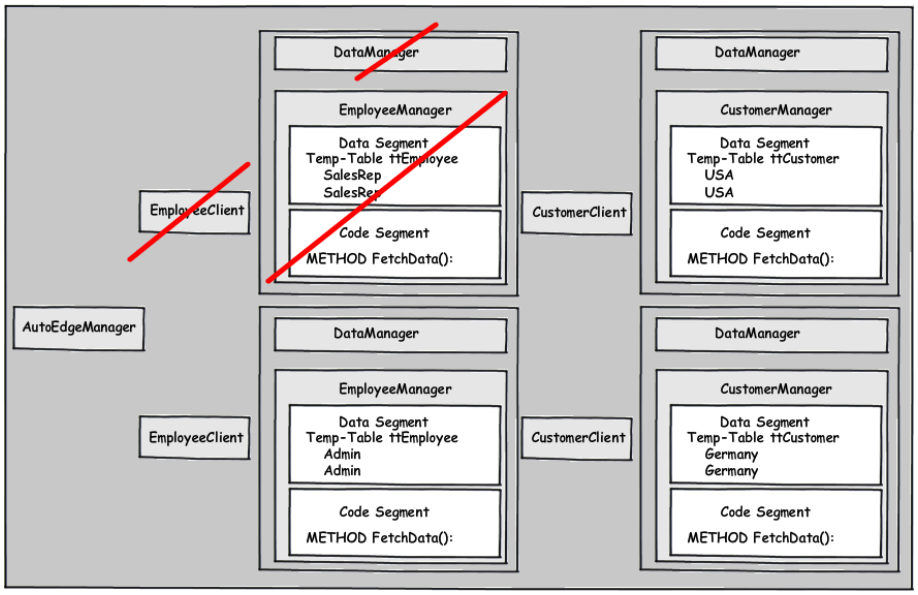
After the final two iterations of the loop, two instances of **CustomerManager** have been created as well, each again with its own **DataManager** object:



The method now deletes each client object it created, which will in turn delete the corresponding manager object. It also deletes each row from the temp-table.

```
        FOR EACH ttObject:
            DELETE OBJECT ttObject.ObjectRef.
            DELETE ttObject.
        END.
        RETURN.

    END METHOD.
```

After the first iteration of the loop, for instance, an **EmployeeClient** object, along with its **EmployeeManager** object and **DataManager** object, will be gone.



Deleting the objects is actually not necessary, because the garbage collection which is a feature of the ABL support for classes will delete objects automatically when they are no longer referenced elsewhere in the session.

For a complete description of the principles of object-oriented programming and the use of classes in ABL that these examples illustrate, please view the videos on PSDN on the Progress Communities website.

For more comprehensive information on programming with classes in OpenEdge 10, please read the manual ***OpenEdge Development: Object-Oriented Programming***, which is part of the OpenEdge documentation set, as well as the detailed descriptions of the ABL syntax that supports classes, in the ABL reference manual.