Table of Contents

JVx's Life Cycle Objects are not only containers for objects and methods, they also allow the reuse of functionality through inheritance. Any POJO can potentially be used as a lifecycle object, although this would mean that we forego the advantage of reuse.

We recommend a special use and a predefined class hierarchy to exploit all the advantages without restrictions!

Configuration

Ideally, life cycle objects are defined for the application and for the MasterSession.

The objects are defined in the application configuration:

```
<application>
  ...
  ...

  <!-- predefined life-cycle object names -->
  <lifecycle>
    <application>com.sibvisions.apps.showcase.Application</application>
    <mastersession>com.sibvisions.apps.showcase.Session</mastersession>
  </lifecycle>

</application>
```

The class name for the MasterSession can also be defined during the creation of MasterConnection using setLifeCycleName.

Each MasterConnection (Client) requires a MasterSession (Server) to access the server.

The life cycle object for the application is optional and is only required for tasks spanning multiple applications.

Class Hierarchy

We will explain the class hierarchy based on a showcase application.

The applications life cycle object:

Application.java

```
package com.sibvisions.apps.showcase;

...
...

/**
 * Application object for Showcase application.
 */
public class Application extends GenericBean
```

```
{
}   // Application
```

The class GenericBean handles the object administration, which is why we derive from it.

The MasterSession life cycle object:

Session.java

```
package com.sibvisions.apps.showcase;

...
...

/**
 * Session object for Showcase application.
 */
public class Session extends Application
{
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   // User-defined methods
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   /**
    * Returns access to the database.
    *
    * @return the access to the database
    * @throws Exception if the datasource can not be opened
    */
   public DBAccess getDBAccess() throws Exception
   {
      DBAccess dba = (DBAccess)get("dBAccess");

      if (dba == null)
      {
         IConfiguration cfgSession =
SessionContext.getCurrentSessionConfig();

         dba = new HSQLDBAccess();

         //read the configuration from the config file
         dba.setConnection(cfgSession.getProperty(
"/application/securitymanager/database/url"));
         dba.setUser(cfgSession.getProperty(
"/application/securitymanager/database/username"));
         dba.setPassword(cfgSession.getProperty(
"/application/securitymanager/database/password"));
         dba.open();

         put("dBAccess", dba);
```

```
      }

      return dba;
   }

   /**
    * Gets the source code access object.
    *
    * @return the source access object
    */
   public SourceCode getSourceCode()
   {
      SourceCode sc = (SourceCode)get("sourceCode");

      if (sc == null)
      {
         sc = new SourceCode();

         put("sourceCode", sc);
      }

      return sc;
   }

}   // Session
```

We derive from Application to receive full access to the methods and objects of the super class. The derivation of GenericBean ensures the availability of objects.

Each SubConnection (Client), and, therefore, each workscreen, receives its own life cycle object:

Educations.java

```
package com.sibvisions.apps.showcase.frames;

...
...

/**
 * The <code>Educations</code> class is the life cycle object for
<code>EducationsFrame</code>.
 */
public class Educations extends Session
{
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   // User-defined methods
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   /**
    * Returns the educations storage.
```

```
     *
     * @return the Educations storage
     * @throws Exception if the initialization throws an error
     */
    public DBStorage getEducations() throws Exception
    {
       DBStorage dbsEducations = (DBStorage)get("educations");

       if (dbsEducations == null)
       {
          dbsEducations = new DBStorage();
          dbsEducations.setDBAccess(getDBAccess());
          dbsEducations.setFromClause("V_EDUCATIONS");
          dbsEducations.setWritebackTable("EDUCATIONS");
          dbsEducations.open();

          put("educations", dbsEducations);
       }

       return dbsEducations;
    }

}   // Educations
```

The life cycle object is derived from session to also receive full access to all methods and objects of the super class.

By calling getDBAccess we can see the advantage of this technique. We open the database connection at a central location and all derivations have access to the connection.

By using this procedure we perform changes at a central location; we can save time and solve dependencies.

GenericBean

The previous example shows that, due to the derivations, all methods are inherited but each instance would usually manage its own objects. We would, therefore, expect that each instance of Educations creates a new database connection through the call of getDBAccess!

This is the difference between POJO and GenericBean.

Because of the derivation of GenericBean, the server makes sure that all instances are reused. In our example, the session Instance is set as the parent at the instantiation of Educations and the Application instance is set as parent of the Session instance. Because of this definition, the method getDBAccess always returns the same database connection.

Another feature of GenericBean is the access to managed objects by their names:

```
DBStorage dbsEducations = (DBStorage)get("educations");
```

We can, therefore, call either getEducations() or get(“educations”) and, in both cases, the same instance is delivered. For this process to work, the desired object has to be instantiated and put to the cache:

```
dbsEducations = new DBStorage();
dbsEducations.setDBAccess(getDBAccess());
dbsEducations.setFromClause("V_EDUCATIONS");
dbsEducations.setWritebackTable("EDUCATIONS");
dbsEducations.open();

put("educations", dbsEducations);
```

A rather unusual, but economical, approach of the GenericBean is the initialization of objects without get methods. In this case objects are accessed via name only.

Our Educations life cycle object would be implemented as follows:

Educations.java

```
public class Educations extends Session
{
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   // User-defined methods
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

   /**
    * Initializes the educations storage.
    *
    * @return the educations storage
    * @throws Exception if the initialization throws an error
    */
   @SuppressWarnings("unused")
   private DBStorage initEducations() throws Throwable
   {
       dbsEducations = new DBStorage();
       dbsEducations.setDBAccess(getDBAccess());
       dbsEducations.setFromClause("V_EDUCATIONS");
       dbsEducations.setWritebackTable("EDUCATIONS");
       dbsEducations.open();

       return dbsEducations;
   }

}   // Educations
```

Initialization occurs automatically at first access using get(“educations”). The name of the method has to be considered: “init” + “Educations” (depends on the object name).

A disadvantage of this method is that derived classes have no overview of the managed objects!