Table of Contents

Die Lifecycle Objekte von JVx sind einerseits Container für Objekte bzw. Methoden und andererseits ermöglichen sie die Wiederverwendung von Funktionalitäten durch Vererbung. Als Lifecycle Objekt kann auf Wunsch jedes POJO verwendet werden. Dadurch würde man aber auf den großen Vorteil der Wiederverwendung verzichten.

Wir empfehlen eine spezielle Verwendung und eine vordefinierte Klassenhierarchie um alle Vorteile ohne Einschränkungen zu nutzen!

Konfiguration

Im Idealfall wird je ein Lifecycle Objekt für die Applikation und für die MasterSession definiert.

Die Definition wird in der Applikationskonfiguration durchgeführt:

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

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

</application>
```

Die Klassenbezeichnung für die MasterSession kann auch beim Erstellen der MasterConnection durch setLifeCycleName definiert werden.

Jede MasterConnection (Client) benötigt eine MasterSession (Server) um auf den Server zugreifen zu können.

Das Lifecycle Objekt für die Applikation ist optional und wird nur für Applikationsübergreifende Aufgaben benötigt.

Klassenhierarchie

Wir erklären die Klassenhierarchie anhand der Showcase Applikation.

Das Applikations Lifecycle Objekt:

Application.java

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

...
...

/**
 * Application object for Showcase application.
```

```
 */
public class Application extends GenericBean
{
}   // Application
```

Die Klasse GenericBean übernimmt die Objektverwaltung. Aus diesem Grund leiten wir davon ab.

Das MasterSession Lifecycle Objekt:

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
```

Wir leiten von Application ab um vollen Zugriff auf alle Methoden und Objekte des Vorgängers zu erhalten. Durch die generelle Ableitung von GenericBean wird die Verfügbarkeit der Objekte gewährleistet.

Jede SubConnection (Client), sprich jeder WorkScreen erhält ein eigenes Lifecycle Objekt:

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
```

Das Lifecycle Objekt wird abgeleitet von Session um auch hier den vollen Zugriff auf alle Methoden und Objekte der Vorgänger zu erhalten.

Anhand des Aufrufs von getDBAccess sehen wir bereits den Vorteil dieser Technik. Wir öffnen die Datenbankverbindung an zentraler Stelle und alle Ableitungen haben Zugriff auf die Verbindung.

Durch dieses Vorgehen werden Änderungen an zentraler Stelle durchgeführt, wir sparen Zeit und lösen Abhängigkeiten auf.

GenericBean

Anhand des vorherigen Beispiels ist zwar gut zu erkennen, das aufgrund der Ableitungen auch alle Methoden vererbt werden, doch üblicherweise würde doch jede Instanz ihre eigenen Objekte verwalten. Wir würden also erwarten, daß jede Instanz von Educations durch den Aufruf von getDBAccess auch eine neue Datenbankverbindung herstellt!

Und genau darin liegt der Unterschied zwischen POJO und GenericBean.

Wenn von GenericBean abgeleitet wird, sorgt der Server dafür, daß Instanzen wiederverwendet werden. In unserem konkreten Beispiel wird bei der Instanzierung von Educations die Session Instanz als Parent gesetzt. Die Session Instanz selbst erhält als Parent die Application Instanz. Dadurch liefert unser Aufruf von getDBAccess immer ein und dieselbe Datenbankverbindung.

Eine weitere Besonderheit von GenericBean ist der Zugriff auf die verwalteten Objekte durch deren Namen:

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

Wir können also entweder getEducations() oder get(“educations”) aufrufen und bekommen in beiden Fällen dieselbe Instanz geliefert. Damit dies funktioniert muss das gewünschte Objekt instanziert und hinterlegt werden:

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

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

Ein eher ungewöhnlicher jedoch sparsamer Ansatz des GenericBean ist die Objektinitialisierung ohne get Methoden. In diesem Fall wird nur per Name auf die Objekte zugegriffen.

Unsere Educations Lifecycle Objekt würde wie folgt implementiert werden:

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
```

Die Initialisierung wird beim ersten Zugriff mit get(“educations”) automatisch durchgeführt. Der Name der Methode muss beachtet werden: “init” + “Educations” (richtet sich nach dem Objektnamen).

Der Nachteil dieser Methode ist, daß abgeleitete Klassen keinen Überblick über die verwalteten Objekte bekommen!