Table of Contents

A server-side action is a simple Java method defined in a life cycle object (LCO). A standard JVx application has an application global LCO, a session LCO, and a LCO for every screen.

An action can be defined in every LCO, but it makes sense to define the actions in the right “context” and for the right “scope”. But what is the right scope?

Start defining your actions in screen LCOs and if an action is reusable, move it to the session LCO. The application LCO is not a relevant context for actions because it makes no difference if you define an action in the application LCO or session LCO. The application LCO exists exactly once for an application. The session LCO will be created for every master connection (user login). The application LCO can be used to store “application global” objects or states.

The general rule is that the screen LCO should be the first place for your actions, and the session LCO should be used for reusable actions. However, be careful because there are some things to know!

You application usually has the following classes:

```java
// Application LCO
public class Application extends GenericBean
{
}

// Session LCO
public class Session extends Application
{
}

// A Screen LCO
public class Students extends Session
{
}
```

The Students LCO extends the Session LCO, and the Session LCO extends the Application LCO. This class hierarchy makes it possible to call actions from the super class, e.g.,

Session.java

```java
// Session LCO
public class Session extends Application
{
    public CachedObject getContacts()
    {
        CachedObject obj = (CachedObject)get("contacts");

        if (obj == null)
        {
            obj = new CachedObject();
            obj.setId("contacts");

            put("contacts", obj);
        }
        return obj;
    }

    public void sendAlert(String pMessage)
    {
        Mail.send("noreply@sibvisions.com", "alert@customer.com",
                   null, "Alert from EG", pMessage);
    }
}

// A Screen LCO
public class Students extends Session
{
}
```

It's possible to call the sendAlert action through the Students LCO because the method was defined in the super class - Session - and it's a public method. This is possible because of the class hierarchy.

But if you call the action, it will be called in the Session LCO instance and not in the Students LCO instance. *What?*

The Students LCO has a parent LCO, and this is the Session LCO. In turn, the Session LCO has the Application LCO as parent. The Application LCO doesn't have a parent:

The parents will be set automatically from the ObjectProvider, but only if your LCOs are an instance of GenericBean, which is preferred. It would be possible to create your LCOs as instance of map interface, but you would lose many advantages like object caching. The LCOs in our example are GenericBean instances!

Why Does JVx Set a Parent LCO?

Simple answer: to save memory and object instances.

All objects in the Application LCO exist only once - like singletons per application. All objects in the Session LCO exist only once per user. All objects in the Screen LCOs, like Students, exist for every screen instance.

If you close a screen, all Screen LCO instances will be discarded, and if you logout from the application, all Session LCO instances will be discarded. This mechanism helps to reduce the memory consumption but it adds complexity to your LCOs.

The Students LCO extends the Session LCO and all public/protected/package private methods of the Session LCO are available in the Students LCO. However, if you access an object like contacts via getContacts, you will get the instance from the Session LCO and not a new instance from the Students LCO. If you overwrite the method in the Students LCO, you will receive a new instance of the object for your LCO and not the instance from the Session LCO. This is a built-in security mechanism and a sort of object scope.

The only thing you should really know is that a call or an object access will use the LCO where the method was declared.

In most cases, the standard mechanism works like a charm, but sometimes you need more control. Imagine you have three screens and every screen has the same actions. You would define all actions in the Session LCO because they are reusable. But all actions are stateful and save a timestamp in a field property:

```java
// Application LCO
public class Application extends GenericBean
{
}

// Session LCO
public class Session extends Application
{
    private Date date;

    public String openAccount()
    {
        date = new Date();

        return UUID.randomUUID().toString();
    }

    public Date getOpenDate()
    {
        return date;
    }

    public void releaseAccount()
    {
        date = null;
    }
}

// A Screen LCO
public class Students extends Session
{
}

// A Screen LCO
public class Teacher extends Session
{
}
```

If you call the openAccount action via Students LCO, the Session LCO will be used, and the date field will be saved in Session LCO. If you call the same action via Teacher LCO, the Session LCO will be used, and the same date field will be used. This is not good because the Screen LCOs are not independent of each other!

One possible solution could be that you move all reusable screen actions to a placeholder class:

```
// Session LCO
public class Session extends Application
{
}

// placeholder class for actions
public class AccountSession extends Session
{
    private Date date;

    public String openAccount()
    {
        date = new Date();

        return UUID.randomUUID().toString();
    }

    public Date getOpenDate()
    {
        return date;
    }

    public void releaseAccount()
    {
        date = null;
    }
}

// A Screen LCO
public class Students extends AccountSession
{
}

// A Screen LCO
public class Teacher extends AccountSession
{
}
```

If you call the openAccount action, the date field will be saved in Students or Teacher LCO, because the placeholder class is not the Session LCO, and everything will work as expected. The Screen LCOs are independent from each other.

This solution needs an extra class to work around the built-in call mechanism of JVx. But there's a built-in solution without placeholder class:

```
// Session LCO
public class Session extends Application
```

```
{
    private Date date;

    @Inherit
    public String openAccount()
    {
        date = new Date();

        return UUID.randomUUID().toString();
    }

    @Inherit
    public Date getOpenDate()
    {
        return date;
    }

    @Inherit
    public void releaseAccount()
    {
        date = null;
    }
}

// A Screen LCO
public class Students extends Session
{
}

// A Screen LCO
public class Teacher extends Session
{
}
```

All action methods are annotated with @Inherit. This annotation defines that the action call will be done in the context of the caller LCO. The annotation doesn't really move the scope from Session to Screen, but it has the same effect.