Drag & Drop - Dealing with transient Objects

Dragging and dropping is fairly simple with CaptainCasa Enterprise Client:

- The component you want to drag from provides a DRAGSEND attribute in which you feed in the information to drag from. Example: by specifying “ARTICLE:4711” you define that an information of type “ARTICLE” is available for dragging, the concrete instance being “4711”.
- The component you want to drop information onto defines the types of information that is accepted by using the attribute DROPRECEIVE. Example: by specifying “ARTICLE;ORDER” a component defines that the user may drop information of type “ARTICLE” or of type “ORDER” onto the component.
- Once a user drops information onto a component, the component triggers the corresponding action listener on server side – passing the information about what was dropped.

Persistent <=> Transient Objects

The mechanism described in the introduction works without any problems as long “real” objects are dragged and dropped: articles, orders, ... - these objects typically hold an id and these objects typically are persistent objects: the receiver exactly knows how to obtain an object instance out of an id when processing the drop action listener.

When it comes to transient objects that you want to transfer between different parts of you user interface things become more difficult:

- You have to somehow define a string identifying the object.
- The receiver (drop processing) has to find the object instance based on this string.

Centrally registering Objects...

The solution – of course – is to somehow register transient objects in a central hash table after having associated the object with a key. This key can be used in the drag-drop processing so that the receiving drop processing may use this key to obtain the object from the central hash table.

But: such central registration instances have a strong tendency to collect object by object, without properly releasing the objects when not required any longer.

Weak references!...!!

The solution is simple: the usage of weak references. The central registration needs to register such instances in a weak way: if they are not “really” referenced anymore within your object graph, then they should disappear from the central hash table as well.

The reference manager that is shown in the next chapter exactly does so – and can be either used directly (it is part of the CaptainCasa delivery) or you can write your own one and copy from it.

Reference Manager

The following reference manager is a central point of registration for transient objcets.

```
package org.eclnt.jsfserver.util.references;

import java.lang.ref.ReferenceQueue;
import java.lang.ref.WeakReference;
import java.util.Hashtable;

import org.eclnt.util.log.CLog;

/**
 * This object holds weak references to any object that is added. Per object a
 * textual id is passed back, that can be used for referencing the object.
 */
public class ReferenceMgr
{
    // ------------------------------------------------------------------------
    // inner classes
    // ------------------------------------------------------------------------

    class MyCleanUpThread extends Thread
    {
        @Override
        public void run()
        {
            while (true)
            {
                try { sleep(m_cleanUpThreadInterval); }
                    catch (Throwable t) { break; }
                cleanUpReferences();
            }
        }
    }

    class MyWeakReference extends WeakReference<Object>
    {
        long i_key;
        public MyWeakReference(long key, Object o)
        {
            super(o,m_referenceQueue);
            i_key = key;
        }
    }

    // ------------------------------------------------------------------------
    // members
    // ------------------------------------------------------------------------

    static ReferenceMgr s_instance = new ReferenceMgr();

    long m_referenceCounter = 0;
    Hashtable<Long,MyWeakReference> m_objects =
                          newHashtable<Long,MyWeakReference>();
    ReferenceQueue<Object> m_referenceQueue;
    long m_cleanUpThreadInterval = 5000;

    // ------------------------------------------------------------------------
    // constructors
    // ------------------------------------------------------------------------

    private ReferenceMgr()
    {
        m_referenceQueue = new ReferenceQueue<Object>();
        (new MyCleanUpThread()).start();
    }

    // ------------------------------------------------------------------------
    // public usage
    // ------------------------------------------------------------------------

    public static ReferenceMgr getInstance() { return s_instance; }

    public long getCleanUpThreadInterval() { return m_cleanUpThreadInterval; }
    public void setCleanUpThreadInterval(long cleanUpThreadInterval)
        { m_cleanUpThreadInterval = cleanUpThreadInterval; }

    public long registerObject(Object o)
    {
        long key = increaseReferenceCounter();
        MyWeakReference ref = new MyWeakReference(key,o);
```

CaptainCasa Enterprise Client

```
        m_objects.put(key,ref);
        return key;
    }

    public Object getRegisteredObject(long key)
    {
        MyWeakReference ref = m_objects.get(key);
        if (ref == null)
            return null;
        return ref.get();
    }

    // ------------------------------------------------------------------------
    // private usage
    // ------------------------------------------------------------------------

    private synchronized void cleanUpReferences()
    {
        int cleanedReferences = 0;
        while (true)
        {
            MyWeakReference ref = (MyWeakReference)m_referenceQueue.poll();
            if (ref == null)
                break;
            m_objects.remove(ref.i_key);
            cleanedReferences++;
        }
        CLog.L.log(CLog.LL_INF,"ReferenceMgr - number of cleaned references: " +
cleanedReferences);
    }

    private synchronized long increaseReferenceCounter()
    {
        m_referenceCounter++;
        return m_referenceCounter;
    }

}
```

Usage

The usage is:

- By calling the static method “ReferenceMgr.getInstance()” you receive the central instance of the reference manager.
- You may register objects using the “registerObject(...)” method and receive back an id that can be “externalized” and that can be used within a DRAGSEND definition.
- Vice versa you may obtain objects back by calling “getRegisteredObject(...)”, now passing the id as argument and receiving the object that was registered before.

The reference manager is cleaning up its references by a thread that by default operates all 5 seconds. Using weak reference queues these objects are identified that are no longer references and removed from the central hash table of the reference manager.

“Pay Attention”-Issues

The way of transferring transient objects into “temporary long identifiers” has some advantages – but also some issues to pay attention to.

Session Migration

In a servlet based environment you may have prepared your application to support session state migration from one server to the next. Because the coding above is based on static members (s_instance) it is not supporting session migration. Extending the coding above to also support session migration would be still simple . but we did not do so, in order to

not overload this technical documentation.

Thread Management

There is a thread required that constantly cleans up weak references after the corresponding objects were removed by the garbage collector.

Opening an own thread must be in sync with your application server environment. There might be security policies that restrict the opening of own threads by the application. Maybe there are already exisiting observer-threads in which you can plug in the clean up function instead of opening a thread on your own.

Central Services...

Please be aware of the fact that in the code of the section above all sessions within an server environment access one and the same static instance of “ReferenceMgr”. This means:

When extending the ReferenceMgr on your own, pay attention to multi-threading issues!

And: the number of items that are internally kept in the hashtable might become quite high – depending on the way the “ReferenceMgr” is used by the application. This is not a problem at all, but we want to explicitly point out! ;-)

CaptainCasa Enterprise Client