

Transmission Impossible? Java5 Enumerations und IIOP

Im September 2006 wird in der Java-to-IDL Revision Task Force Mailing List der Erweiterungsvorschlag für das Mapping von Java5-Enumerationstypen eingereicht [1]. Diesem Vorschlag wird ein Jahr später stattgegeben [2]. Die Java Language Specification für Java5 erscheint jedoch bereits 2005 - und das fast ein Jahr nach der ersten offiziellen Version von Java5 [3][4]. Erste Prototypen von Java5 standen der Allgemeinheit wahrscheinlich noch viel früher zur Verfügung. Damit vergingen mindestens drei Jahre seit der Verfügbarkeit der nativen Enumerationstypen in Java5 und deren Mapping auf CORBAs Interface Definition Language. Diese Verzögerung wirkt noch heute für all diejenigen nach, die den Versuch wagen, Java5 Enumerationsliterals über IIOP zu übertragen. Wie dies dennoch mit Java5-Bordmitteln möglich ist, beschreibt der Artikel im Folgenden.

Serialisierung bedeutet im Allgemeinen, Java Objekte in eine Byte-Repräsentation umzuwandeln und diese Byte-Repräsentation in derselben oder einer anderen VM wieder in einen Objektgraphen zurückverwandeln zu können (Deserialisierung). Die Byte-Repräsentation kann zum Beispiel dazu genutzt werden, um Objekte zwischen Programmaufrufen persistent auf einer Festplatte abzulegen oder um Objekte über das Netzwerk zwischen verschiedenen VM-Instanzen austauschen zu können. Den Serialisierungsmechanismus im Detail beschreibt die Java Object Serialization Specification [5].

Der Austausch von Objekten und Objektgraphen über das Netzwerk (Marshalling/Unmarshalling) erfolgt meist im Rahmen von Remote-Calls. Das dazugehörige Java Protokoll heißt Remote Method Invocation (RMI) [6]. Da RMI zusammen mit Java standardisiert ist, ist auch die Übertragung von Enumerationsliterals hier kein Problem. Clients, die direkt über RMI/JRMP auf beispielsweise einen Application-Server zugreifen (z.B. JBoss AS), können deshalb problemlos mit Enumerationsliterals umgehen. Ebenso funktioniert der Zugriff meist problemlos, wenn statt JRMP ein proprietäres Basisprotokoll (z.B. T3 im Oracle WebLogic Server) im Spiel ist.

Sind auch andere Technologien außer Java Bestandteil des verteilten Systems, so funktioniert der Datenaustausch ausschließlich über RMI nicht mehr. Für diesen Anwendungsfall wurde in den 1990er Jahren von der OMG eine objektorientier-

te Middleware-Architektur namens CORBA spezifiziert [7]. Als Übertragungsprotokoll innerhalb eines CORBA-Systems wird typischerweise das Internet Inter-ORB Protocol (IIOP) verwendet. Erfreulicherweise existiert in der Java-Standard-Edition

```
package com.buschmais.xpl.iiop;

import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.io.Serializable;
import java.rmi.RMISecurityManager;

import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class ClientMain {

    public static void main(String[] args) throws Exception {

        System.setProperty("java.security.policy", "my.policy");
        System.setSecurityManager(new RMISecurityManager());

        InitialContext initialContext = new InitialContext();

        Object object = initialContext.lookup("MyServer");
        MyServer myServer = (MyServer) PortableRemoteObject
            .narrow(object, MyServer.class);

        Serializable bean = myServer.getObject();
        System.out.println(bean);

        PropertyDescriptor[] propertyDescriptors = Introspector
            .getBeanInfo(bean.getClass())
            .getPropertyDescriptors();
        for (PropertyDescriptor pd : propertyDescriptors) {
            System.out.format("%30s %-15.15s = %s\n",
                pd.getPropertyType().getName(),
                pd.getName(),
                pd.getReadMethod().invoke(bean, (Object[]) null));
        }
    }
}
```

Abb. 1: ClientMain.java

ein RMI-IIOP Binding, so dass selbst JavaSE-Anwendungen an einer CORBA-Architektur als Server und Client partizipieren können.

Das Binding ist weitgehend transparent für die Anwendung und unterscheidet sich kaum vom RMI-Programmiermodell. Das sind beste Voraussetzungen, um das eingangs skizzierte Problem verdeutlichen zu können.

Die Basis-Konstellation

Um das Beispiel realistisch zu gestalten, wird auch dynamisches Class-Loading über RMI verwendet. Der Client sieht wie in [Abbildung 1](#) aus (ClientMain.java).

In den Zeilen 15 und 16 wird ein SecurityManager in Stellung gebracht, der für das Laden von fremden Code zwingend erforderlich ist. Der SecurityManager kann über eine Policy-Datei feingranular konfiguriert werden. Der Einfachheit halber wird alles erlaubt (siehe [Abbildung 2](#) my.policy).

In den Zeilen 18 bis 22 wird Verbindung zur einer Object-Registry aufgenommen und dort das Objekt mit dem Namen „MyServer“ erfragt. Die erhaltene Objekt-Referenz, also der RMI-Stub, muss noch gegen das Service-Interface gecastet werden, damit das Objekt für entfernte Funktionsaufrufe genutzt werden kann (Zeilen 21,22). Der entfernte Aufruf „getObject()“ erfolgt schließlich in Zeile 24.

In den Zeilen 27 bis 35 wird mittels Bean-Introspection der Inhalt des zurückgelieferten Objekts untersucht und ausgegeben.

Die Server-Implementierung ist minimalistisch (siehe [Abbildung 3](#) MyServerImpl.java).

```
grant {
    permission java.security.AllPermission;
};
```

Abb. 2: my.policy

```
package com.buschmais.xpl.iiop;

import java.io.Serializable;
import java.rmi.RemoteException;

import javax.rmi.PortableRemoteObject;

public class MyServerImpl
    extends PortableRemoteObject
    implements MyServer {

    protected MyServerImpl() throws RemoteException {
        super();
    }

    public Serializable getObject() throws RemoteException {
        MyJavaBean bean = new MyJavaBean();
        bean.set...(...);
        return bean;
    }
}
```

Abb. 3: MyServerImpl.java

```
package com.buschmais.xpl.iiop;

import java.io.Serializable;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MyServer extends Remote {

    Serializable getObject() throws RemoteException;

}
```

Abb. 4: MyServer.java

```
<rmic base="${basedir}/build/server"
    includes="**/MyServerImpl.class"
    iiop="true">
  <extdirs path="${basedir}/build/api" />
  <classpath>
    <pathelement location="${basedir}/build/api" />
  </classpath>
</rmic>
```

Abb. 5: build.xml

Ebenso minimalistisch ist das zugehörige Service-Interface (siehe [Abbildung 4](#) MyServer.java).

Um den Server zu bauen, ist nach dem normalen Compile-Durchlauf ein weiterer Schritt zur Generierung

der Stub- und Skeleton-Klassen für die IIOP-Aufrufe notwendig. Hierfür wird im Java-SDK das Werkzeug „rmic“ mitgeliefert. Unter Verwendung von Apache Ant sieht der Aufruf wie in [Abbildung 5](#) aus (build.xml).

Der Server wird über eine Hilfsklasse gestartet (siehe [Abbildung 6](#) `ServerMain.java`).

Symmetrisch zum Client erzeugt diese Klasse einen `InitialContext`, um darüber die Service-Bean-Implementierung zu registrieren (Zeilen 17-19).

Zuvor wird noch der `ResourceLocator` der aktuellen Codebase ermittelt und in der Systemvariable „`java.rmi.server.codebase`“ abgelegt. Damit kann jeder Client fehlende Klassen aus dem `ServerContext` automatisch nachladen (Zeilen 11-15).

Um den `InitialContext` erzeugen zu können, ist sowohl beim Client als auch beim Server die Angabe der `InitialContext-Factory` und die URL der Objekt-Registry notwendig. Legt man eine Datei namens „`jndi.properties`“ in den Classpath, so werden die Konfigurationsparameter automatisch daraus geladen (siehe [Abbildung 7](#) `jndi.properties`).

Jetzt ist nur noch die Objekt-Registry zu starten. Dies lässt sich über das SDK-Werkzeug „`orbd`“ realisieren. Als Parameter ist noch die Angabe des Ports notwendig: `-ORBInitialPort 1050`.

Mit diesen Informationen lässt sich das Szenario bereits ausführen. Eine Ausgabe des Clients könnte wie in [Abbildung 8](#) aussehen.

Java5-Enumerationen

Jede Java5-Enumeration leitet sich von `java.lang.Enum` ab. `java.lang.Enum` implementiert das Marker-Interface `java.io.Serializable` mit der Folge, dass alle Enumerationsliterals im Grunde genommen serialisiert werden können. Diese Aussage trifft auf

```
package com.buschmais.xpl.iio;

import java.net.URL;

import javax.naming.InitialContext;

public class ServerMain {

    public static void main(String[] args) throws Exception {

        URL currentCodebase = ServerMain.class
            .getProtectionDomain().getCodeSource()
            .getLocation();
        System.setProperty(„java.rmi.server.codebase“,
            currentCodebase.toString());

        InitialContext initialContext = new InitialContext();
        MyServer server = new MyServerImpl();
        initialContext.rebind(„MyServer“, server);
        System.out.println(„Server is up and running.“);
    }
}
```

Abb. 6: `ServerMain.java`

```
java.naming.factory.initial = com.sun.jndi.cosnaming.CNCTXFactory
java.naming.provider.url = iiop://localhost:1050
```

Abb. 7: `jndi.properties`

```
com.buschmais.xpl.iio.MyJavaBean@384065
java.lang.Class class = class com.buschmais.xpl.iio.MyJavaBean
java.util.Date dateAttribute = Wed Jul 02 07:28:52 CEST 2008
int intAttribute = 2
java.lang.Long longAttribute = 1214976532123
java.lang.String stringAttribute = Java 1.5.0_15
```

Abb. 8: Programm-Ausgabe

Stream-Serialisierung und RMI un- eingeschränkt zu. Nicht so jedoch auf IIOP. Hier tritt mit den Sun JDK 1.5.0_15 die Exception in [Abbildung 9](#) auf.

Mit früheren Versionen von Java 1.5.0 tritt keine Exception auf, ebenso nicht mit Java 1.6.0_06. Dort darf man aber über das Phänomen staunen, dass deserialisierte Enumerationsliterals nicht mehr referenzgleich sind. Schaut man sich die Methode `java.lang.Enum#equals(Object)` an, so ist Referenzgleichheit von Enumerationsliterals aber offenkundig zwingend.

Eine Analyse obiger Deserialisierungs-Exception führt schnell zu ihrem Verursacher. Es ist die Methode `java.lang.Enum#readObject()`, die im Stile einer Notbremse die Exception „`can't deserialize enum`“ wirft. Dass diese Änderung in der Implementierung von `java.lang.Enum` von Java 1.5.0_14 auf Java 1.5.0_15 keinen Eingang in die ReleaseNotes gefunden hat, unterstreicht ihren provisorischen Charakter.

```

02.07.2008 10:17:21 com.sun.corba.se.impl.encoding.CDRInputStream_1_0 read_value
WARNUNG: „IOP00810211: (MARSHAL) Exception from readValue on ValueHandler in CDRInputStream“
org.omg.CORBA.MARSHAL: vmcid: SUN minor code: 211 completed: Maybe
  at com.sun.corba.se.impl.logging.ORBUtilSystemException.valuehandlerReadException(ORBUtilSystemException.
  java:6500)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1045)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_value(CDRInputStream.java:253)
  at com.sun.corba.se.impl.io.IIOPInputStream.inputObjectField(IIOPInputStream.java:1989)
  at com.sun.corba.se.impl.io.IIOPInputStream.inputClassFields(IIOPInputStream.java:2213)
  at com.sun.corba.se.impl.io.IIOPInputStream.inputObject(IIOPInputStream.java:1221)
  at com.sun.corba.se.impl.io.IIOPInputStream.simpleReadObject(IIOPInputStream.java:400)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:327)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:293)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1034)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:879)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_value(CDRInputStream.java:249)
  at com.sun.corba.se.impl.corba.TCUtility.unmarshalIn(TCUtility.java:269)
  at com.sun.corba.se.impl.corba.AnyImpl.read_value(AnyImpl.java:559)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_any(CDRInputStream_1_0.java:739)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_any(CDRInputStream.java:220)
  at com.sun.corba.se.impl.javax.rmi.CORBA.Util.readAny(Util.java:401)
  at javax.rmi.CORBA.Util.readAny(Util.java:92)
  at com.buschmais.xpl.iiop._MyServer_Stub.getObject(Unknown Source)
  at com.buschmais.xpl.iiop.ClientMain.main(ClientMain.java:24)
Caused by: java.io.InvalidObjectException: can't deserialize enum
  at java.lang.Enum.readObject(Enum.java:205)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at com.sun.corba.se.impl.io.IIOPInputStream.invokeObjectReader(IIOPInputStream.java:1694)
  at com.sun.corba.se.impl.io.IIOPInputStream.inputObject(IIOPInputStream.java:1212)
  at com.sun.corba.se.impl.io.IIOPInputStream.simpleReadObject(IIOPInputStream.java:400)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:327)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:293)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1034)
  ... 18 more
Exception in thread „main“ java.rmi.MarshalException: CORBA MARSHAL 1398079699 Maybe; nested exception is:
org.omg.CORBA.MARSHAL: vmcid: SUN minor code: 211 completed: Maybe
  at com.sun.corba.se.impl.javax.rmi.CORBA.Util.mapSystemException(Util.java:197)
  at javax.rmi.CORBA.Util.mapSystemException(Util.java:67)
  at com.buschmais.xpl.iiop._MyServer_Stub.getObject(Unknown Source)
  at com.buschmais.xpl.iiop.ClientMain.main(ClientMain.java:24)
Caused by: org.omg.CORBA.MARSHAL: vmcid: SUN minor code: 211 completed: Maybe
  at com.sun.corba.se.impl.logging.ORBUtilSystemException.valuehandlerReadException(ORBUtilSystemException.
  java:6500)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1045)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_value(CDRInputStream.java:253)
  at com.sun.corba.se.impl.io.IIOPInputStream.inputObjectField(IIOPInputStream.java:1989)
  at com.sun.corba.se.impl.io.IIOPInputStream.inputClassFields(IIOPInputStream.java:2213)
  at com.sun.corba.se.impl.io.IIOPInputStream.inputObject(IIOPInputStream.java:1221)
  at com.sun.corba.se.impl.io.IIOPInputStream.simpleReadObject(IIOPInputStream.java:400)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:327)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:293)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1034)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_value(CDRInputStream.java:249)
  at com.sun.corba.se.impl.corba.TCUtility.unmarshalIn(TCUtility.java:269)
  at com.sun.corba.se.impl.corba.AnyImpl.read_value(AnyImpl.java:559)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_any(CDRInputStream_1_0.java:739)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_any(CDRInputStream.java:220)
  at com.sun.corba.se.impl.javax.rmi.CORBA.Util.readAny(Util.java:401)
  at javax.rmi.CORBA.Util.readAny(Util.java:92)
  ... 2 more
Caused by: java.io.InvalidObjectException: can't deserialize enum
  at java.lang.Enum.readObject(Enum.java:205)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at com.sun.corba.se.impl.io.IIOPInputStream.invokeObjectReader(IIOPInputStream.java:1694)
  at com.sun.corba.se.impl.io.IIOPInputStream.inputObject(IIOPInputStream.java:1212)
  at com.sun.corba.se.impl.io.IIOPInputStream.simpleReadObject(IIOPInputStream.java:400)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:327)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:293)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1034)
  ... 18 more

```

EnumHolder

Da die Implementierung von Java 1.5.0_15 dem Deserialisieren von Enumerationsliteralen einen wirksamen Riegel vorgeschoben hat, muss man sich auf eine List besinnen, um dennoch transparent für Server und Client Enumerationen austauschen zu können. Die List ist so einfach wie wirksam und erschlägt gleichzeitig noch das Problem der fehlerhaften Referenzgleichheit der deserialisierten Enumerationsliterals. Die Javadoc zu `java.io.Serializable` beschreibt einige Möglichkeiten, wie der Serialisierungsmechanismus klassenspezifisch angepasst werden kann. Interessant sind hier die Methoden `writeReplace()` und `readResolve()`. `writeReplace()` ermöglicht, statt des eigentlichen Objektes ein Stellvertreter-Objekt zu serialisieren. `readResolve()` ist das Gegenstück davon und erlaubt, statt des deserialisierten Objektes ein Stellvertreter-Objekt für die Rekonstruktion des Objektgraphen zu verwenden.

Da Enumerationen mit Java 1.5.0_15 per se nicht deserialisiert werden können, wird statt ihrer ein `EnumHolder`-Objekt übertragen (siehe [Abbildung 10](#) `EnumHolder.java`).

Diese Klasse ist serialisierbar und überträgt die Enumerationsklasse und die String-Repräsentation des Enumerationsliterals. Beim Deserialisieren wird aufgrund der Methode `readResolve` das gewünschte Enumerationsliteral als VM-Singleton zurückgeliefert.

Mit Hilfe der `writeReplace` kann der `EnumHolder` statt des Enumerationsliterals über den Draht geschickt werden. Eine Enumeration würde hierzu wie in [Abbildung 11](#) implementiert werden müssen.

```
package com.buschmais.xpl.iiop;

import java.io.ObjectStreamException;
import java.io.Serializable;

public class EnumHolder<E extends Enum<E>> implements
    Serializable {

    private static final long serialVersionUID = -52531617949L;

    private Class<E> clazz;
    private String value;

    public EnumHolder(E enumLiteral) {
        clazz = enumLiteral.getDeclaringClass();
        value = enumLiteral.name();
    }

    private Object readResolve() throws ObjectStreamException {
        return Enum.valueOf(clazz, value);
    }
}
```

Abb. 10: `EnumHolder.java`

```
package com.buschmais.xpl.iiop;

import java.io.ObjectStreamException;

public enum Color {

    BLUE, YELLOW, RED;
    static final long serialVersionUID = 0L;

    private Object writeReplace() throws ObjectStreamException {
        return new EnumHolder<Color>(this);
    }
}
```

Abb. 11: `Color.java`

Wer einen tieferen Blick auf die eingangs beschriebene Java-to-IDL Mapping Spezifikation geworfen hat, wird feststellen, dass Java5-Enumerationen in einer sehr ähnlichen Weise über IIOp verschickt werden: Die Enumerationsklasse und die String-Repräsentation dienen auch hier zur eindeutigen Identifikation der Enumerationsliterals.

Die Tatsache, dass Java5-Enumerationen nicht korrekt über IIOp übertragen werden, ist kein Naturgesetz, sondern eine bedauerliche Auslassung der Java-Standard-Edition. Suns Application-Server GlassFish scheint die Lücke bereits geschlossen zu haben [8].

Ein weiterer, prominenter Kandidat für die Verwendung von IIOp ist Oracles WebLogic Server. IIOp wird hier als Alternative zum T3 Protokoll angeboten, mit dem Vorzug einen schlanken Client zu bekommen.

Spezialfall Oracle WebLogic Server

Wagt man den Versuch, mit dem Oracle WebLogic Server 10.0 MP1 Enumerationsliterals über IIOp zu übertragen, so erhält man clientseitig die Fehlermeldung in [Abbildung 12](#).

```

03.07.2008 10:02:13 com.sun.corba.se.impl.encoding.CDRInputStream_1_0 read_value
WARNUNG: „IOP00810211: (MARSHAL) Exception from readValue on ValueHandler in CDRInputStream“
org.omg.CORBA.MARSHAL: vmcid: SUN minor code: 211 completed: Maybe
  at com.sun.corba.se.impl.logging.ORBUtilSystemException.valueHandlerReadException(ORBUtilSystemException.
    java:6500)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1045)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:879)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_value(CDRInputStream.java:249)
  at com.sun.corba.se.impl.corba.TCUtility.unmarshalIn(TCUtility.java:269)
  at com.sun.corba.se.impl.corba.AnyImpl.read_value(AnyImpl.java:559)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_any(CDRInputStream_1_0.java:739)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_any(CDRInputStream.java:220)
  at com.sun.corba.se.impl.javax.rmi.CORBA.Util.readAny(Util.java:401)
  at javax.rmi.CORBA.Util.readAny(Util.java:92)
  at com.buschmais.xpl.iiop.server._TestBeanImpl_q1qs1o_TestBeanIntf_Stub.getEnum(Unknown Source)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at weblogic.ejb.container.internal.RemoteBusinessIntfProxy.invoke(RemoteBusinessIntfProxy.java:63)
  at $Proxy0.getEnum(Unknown Source)
  at com.buschmais.xpl.iiop.client.ClientMain.main(ClientMain.java:32)
Caused by: java.io.IOException: Mismatched serialization UIDs : Source (Rep. ID RMI:com.buschmais.xpl.iiop.api.
Color:3762E1FD1A6B1BE5:71AF074F34B7BC2E) = 71AF074F34B7BC2E whereas Target (Rep. ID RMI:com.buschmais.xpl.iiop.
api.Color:64D47C1D01980B5E:0000000000000000) = 0000000000000000
  at com.sun.corba.se.impl.util.RepositoryId.useFullValueDescription(RepositoryId.java:577)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.useFullValueDescription(ValueHandlerImpl.java:388)
  at com.sun.corba.se.impl.io.IIOPInputStream.simpleReadObject(IIOPInputStream.java:397)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:327)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:293)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1034)
  ... 16 more
Exception in thread „Main Thread“ javax.ejb.EJBException: nested exception is: org.omg.CORBA.MARSHAL: vmcid:
SUN minor code: 211 completed: Maybe
org.omg.CORBA.MARSHAL: vmcid: SUN minor code: 211 completed: Maybe
  at com.sun.corba.se.impl.logging.ORBUtilSystemException.valueHandlerReadException(ORBUtilSystemException.
    java:6500)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1045)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:879)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_value(CDRInputStream.java:249)
  at com.sun.corba.se.impl.corba.TCUtility.unmarshalIn(TCUtility.java:269)
  at com.sun.corba.se.impl.corba.AnyImpl.read_value(AnyImpl.java:559)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_any(CDRInputStream_1_0.java:739)
  at com.sun.corba.se.impl.encoding.CDRInputStream.read_any(CDRInputStream.java:220)
  at com.sun.corba.se.impl.javax.rmi.CORBA.Util.readAny(Util.java:401)
  at javax.rmi.CORBA.Util.readAny(Util.java:92)
  at com.buschmais.xpl.iiop.server._TestBeanImpl_q1qs1o_TestBeanIntf_Stub.getEnum(Unknown Source)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:585)
  at weblogic.ejb.container.internal.RemoteBusinessIntfProxy.invoke(RemoteBusinessIntfProxy.java:63)
  at $Proxy0.getEnum(Unknown Source)
  at com.buschmais.xpl.iiop.client.ClientMain.main(ClientMain.java:32)
Caused by: java.io.IOException: Mismatched serialization UIDs : Source (Rep. ID RMI:com.buschmais.xpl.iiop.api.
Color:3762E1FD1A6B1BE5:71AF074F34B7BC2E) = 71AF074F34B7BC2E whereas Target (Rep. ID RMI:com.buschmais.xpl.iiop.
api.Color:64D47C1D01980B5E:0000000000000000) = 0000000000000000
  at com.sun.corba.se.impl.util.RepositoryId.useFullValueDescription(RepositoryId.java:577)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.useFullValueDescription(ValueHandlerImpl.java:388)
  at com.sun.corba.se.impl.io.IIOPInputStream.simpleReadObject(IIOPInputStream.java:397)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:327)
  at com.sun.corba.se.impl.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:293)
  at com.sun.corba.se.impl.encoding.CDRInputStream_1_0.read_value(CDRInputStream_1_0.java:1034)
  ... 16 more
javax.ejb.EJBException: nested exception is: org.omg.CORBA.MARSHAL: vmcid: SUN minor code: 211 completed:
Maybe
  at weblogic.ejb.container.internal.RemoteBusinessIntfProxy.invoke(RemoteBusinessIntfProxy.java:78)
  at $Proxy0.getEnum(Unknown Source)
  at com.buschmais.xpl.iiop.client.ClientMain.main(ClientMain.java:32)

```

Abb. 12: Stacktrace: Mismatched serialization UIDs

```

package com.buschmais.xpl.iiop;

import java.io.IOException;
import java.io.ObjectStreamException;

public enum Color {
    RED, BLUE, GREEN;

    transient private String serializedRepresentation;

    private Object writeReplace() throws ObjectStreamException {
        return this.name();
    }

    private void readObject(java.io.ObjectInputStream in) throws
        IOException, ClassNotFoundException {
        serializedRepresentation = (String) in.readObject();
    }

    private Object readResolve() throws ObjectStreamException {
        return Color.valueOf(serializedRepresentation);
    }
}

```

Abb. 13: Color.java für WebLogic Server

Die Fehlermeldung besagt im Wesentlichen, dass die Serialization UUIDs zwischen dem empfangenen Enumerationsliteral und dem erwarteten Enumerationstyp nicht übereinstimmen. Folglich kann das Enumerationsobjekt nicht aus dem IIO-Stream rekonstruiert werden. Möchte man dennoch Java5-Enumerationen übertragen, so kann man sich auf einen ähnlichen Trick besinnen. Die Enumerationsklasse müsste hierzu ebenfalls den Default-Serialisierungsmechanismus von Java überschreiben. Die Implementierung könnte wie in **Abbildung 13** aussehen (Color.java).

Der Mechanismus funktioniert so, dass statt des Enumerationsliterals sein Value-String übertragen wird (Zeilen 11-13). Dieser String muss auf der Empfängerseite wieder in das Enumerationsliteral zurückverwandelt werden. Dieser Prozess ist zweistufig: Zunächst wird in der Methode `readObject(ObjectInputStream)`

der Value-String aus dem `ObjectInputStream` entgegen genommen und in einem privaten Feld zwischengespeichert (Zeilen 15-18). Danach ruft der Deserialisierungsmechanismus die Methode `readResolve()` auf, die schlussendlich dazu benutzt wird mithilfe des zwischengespeicherten Value-Strings das korrekte Enumerationssingleton aufzufinden. Dieses wird zurückgegeben und damit in den Objektgraphen anstelle des übertragenen Strings eingebunden.

Der Unterschied zum ersten Lösungsvorschlag besteht darin, dass der Oracle WebLogic Server offensichtlich trotz der Methode `writeReplace()` die Hülle des Enumerationsliterals überträgt. Nicht anders ist zu erklären, dass der Deserialisierungsmechanismus nicht so wie oben auf zwei Klassen aufgespalten werden muss.

Fazit

Mit diesem Artikel sollte verdeutlicht werden, wie der Java-Serialisierungsmechanismus angepasst werden kann, um Einschränkungen bei der Übertragung von Enumerationen zu umgehen. Als Beispiel wurde hierfür die Übertragung von Java5-Enumerationenliteralen herangezogen. Falls Sie Interesse an ähnlich gelagerten Themen besitzen, lohnt sich ein Blick auf unser Seminarangebot. So beschäftigt sich das Seminar „Java für Fortgeschrittene“ unter anderem mit den Themen „Objekt-Serialisierung“ und „verteilte Systeme“.

Referenzen

- [1] <http://www.omg.org/issues/issue10336.txt>
- [2] <http://www.omg.org/docs/ptc/07-02-07.htm> (Issue 10336)
- [3] http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
- [4] <http://java.sun.com/developer/technicalArticles/releases/j2se15/>
- [5] <http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html>
- [6] <http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html>
- [7] http://www.omg.org/technology/documents/corba_spec_catalog.htm
- [8] <https://glassfish-corba.dev.java>

Kontakt

buschmais GbR
Leipziger Straße 93
01127 Dresden

Tel +49 (0)351 3209230
Fax +49 (0)351 32092329
info@buschmais.de
www.buschmais.de