JBossJTS 4.15.0

JBossJTS ORB Portability Guide

Mark Little

Jonathan Halliday

Andrew Dinn

Kevin Connor

JBossJTS 4.15.0 JBossJTS ORB Portability Guide

Author	Mark Little	mlittle@redhat.com
Author	Jonathan Halliday	jhallida@redhat.com
Author	Andrew Dinn	adinn@redhat.com
Author	Kevin Connor	kconnor@redhat.com

Preface

1. Prerequisites

JBossJTS works in conjunction with the rest of the JBoss Transactions suite. In addition to the documentation here, consult the JBossJTS documentation, which ships as part of JBossJTS and is also available on the JBoss Transaction Service website at *http://www.jboss.org/jbosstm* .

2. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

2.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

[1] https://fedorahosted.org/liberation-fonts/

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.
>
> The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.
>
> To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

2.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

2.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

3. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in the JBoss Issue Tracker: *https://jira.jboss.org/* against the product **JBossJTS.**

When submitting a bug report, be sure to mention the manual's identifier:

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

About This Guide

The Programmer's Guide contains information on how to use the ORB Portability Layer. Although the CORBA specification is a standard, it is written in such a way that allows for a wide variety of implementations. Unless writing extremely simple applications, differences between ORB implementations tend to produce code which cannot easily be moved between ORBs. This is especially true for server-side code, which suffers from the widest variation between ORBs. There have also been a number of revisions of the Java language mapping for IDL and for CORBA itself. Many ORBs currently in use support different versions of CORBA and/or the Java language mapping.

The JBossJTS only supports the new Portable Object Adapter (POA) architecture described in the CORBA 2.3 specification as a replacement for the Basic Object Adapter (BOA). Unlike the BOA, which was weakly specified and led to a number of different (and often conflicting) implementations, the POA was deliberately designed to reduce the differences between ORB implementations, and thus minimize the amount of re-coding that would need to be done when porting applications from one ORB to another. However, there is still scope for slight differences between ORB implementations, notably in the area of threading. Note, instead of talking about the POA, this manual will consider the Object Adapter (OA).

Because the JBossJTS must be able to run on a number of different ORBs, we have developed an ORB portability interface which allows entire applications to be moved between ORBs with little or no modifications. This portability interface is available to the application programmer in the form of several Java classes. Note, the classes to be described in this document are located in the **`com.arjuna.orbportability`** package.

1.1. Audience

This document provides a detailed look at the ORB Portability layer and how it can be used to facilitate the implementation of ORB portable applications. This guide provides a guide as to the best practices of using the ORB portability layer.

1.2. Prerequisites

Familiarity with ORBs.

ORB Portability API

2.1. Using the ORB and OA

The **ORB** class shown below provides a uniform way of using the ORB. There are methods for obtaining a reference to the ORB, and for placing the application into a mode where it listens for incoming connections. There are also methods for registering application specific classes to be invoked before or after ORB initialisation. Note, some of the methods are not supported on all ORBs, and in this situation, a suitable exception will be thrown. The ORB class is a factory class which has no public constructor. To create an instance of an ORB you must call the getInstance method passing a unique name as a parameter. If this unique name has not been passed in a previous call to getInstance you will be returned a new ORB instance. Two invocations of getInstance made with the same unique name, within the same JVM, will return the same ORB instance.

Example 2.1. **ORB.java**

```
public class ORB {
public static ORB getInstance(String uniqueId);

public synchronized void initORB()
  throws SystemException;

public synchronized void initORB(Applet a, Properties p)
  throws SystemException;

public synchronized void initORB(String[] s,
  Properties p) throws SystemException;

public synchronized org.omg.CORBA.ORB orb();

public synchronized boolean setOrb(
  org.omg.CORBA.ORB theORB);

public synchronized void shutdown();

public synchronized boolean addAttribute(Attribute p);

public synchronized void addPreShutdown(PreShutdown c);

public synchronized void addPostShutdown(PostShutdown c);

public synchronized void destroy()
  throws SystemException;

public void run();

public void run(String name);
}
```

We shall now describe the various methods of the ORB class.

- **initORB** : given the various parameters, this method initialises the ORB and retains a reference to it within the ORB class. This method should be used in preference to the raw ORB interface since the *JBoss Transaction Service* requires a reference to the ORB. If this method is not used, setOrb must be called prior to using *JBoss Transaction Service* .
- **orb** : this method returns a reference to the ORB. After shutdown is called this reference may be null.

- **`shutdown`** : where supported, this method cleanly shuts down the ORB. Any pre- and post-ORB shutdown classes which have been registered will also be called. See the section titled ORB and OA Initialisation. This method must be called prior to application termination. It is the application programmer's responsibility to ensure that no objects or threads continue to exist which require access to the ORB. It is ORB implementation dependant as to whether or not outstanding references to the ORB remain useable after this call.
- **`addAttribute`** : this method allows the application to register classes with JBoss Transaction Service which will be called either before, or after the ORB has been initialised. See the section titled ORB and OA Initialisation. If the ORB has already been initialised then the attribute object will not be added, and false will be returned.
- **`run`** : these methods place the ORB into a listening mode, where it waits for incoming invocations.

The OA classes shown below provide a uniform way of using Object Adapters (OA). There are methods for obtaining a reference to the OA. There are also methods for registering application specific classes to be invoked before or after OA initialisation. Note, some of the methods are not supported on all ORBs, and in this situation, a suitable exception will be thrown. The OA class is an abstract class and provides the basic interface to an Object Adapter. It has two sub-classes RootOA and ChildOA, these classes expose the interfaces specific to the root Object Adapter and a child Object Adapter respectively. From the RootOA you can obtain a reference to the RootOA for a given ORB by using the static method getRootOA. To create a ChildOA instance use the createPOA method on the RootOA.

Example 2.2. **`OA.java`**

```
public abstract class OA {
public synchronized static RootOA getRootOA(
  ORB associatedORB);

public synchronized void initPOA()
  throws SystemException;

public synchronized void initPOA(String[] args)
  throws SystemException;

public synchronized void initOA()
  throws SystemException;

public synchronized void initOA(String[] args)
  throws SystemException;

public synchronized ChildOA createPOA(
  String adapterName, PolicyList policies)
  throws AdapterAlreadyExists, InvalidPolicy;

public synchronized org.omg.PortableServer.POA rootPoa();

public synchronized boolean setPoa(
  org.omg.PortableServer.POA thePOA);

public synchronized org.omg.PortableServer.POA poa(
  String adapterName);

public synchronized boolean setPoa(String adapterName,
  org.omg.PortableServer.POA thePOA);

public synchronized boolean addAttribute(OAAttribute p);

public synchronized void addPreShutdown(OAPreShutdown c);
```

```
 public synchronized void addPostShutdown(
   OAPostShutdown c);
}

public class RootOA extends OA {
 public synchronized void destroy()
   throws SystemException;

 public org.omg.CORBA.Object corbaReference(Servant obj);

 public boolean objectIsReady(Servant obj, byte[] id);

 public boolean objectIsReady(Servant obj);

 public boolean shutdownObject(org.omg.CORBA.Object obj);

 public boolean shutdownObject(Servant obj);
}

public class ChildOA extends OA {
 public synchronized boolean setRootPoa(POA thePOA);

 public synchronized void destroy()
   throws SystemException;

 public org.omg.CORBA.Object corbaReference(Servant obj);

 public boolean objectIsReady(Servant obj, byte[] id)
   throws SystemException;

 public boolean objectIsReady(Servant obj)
   throws SystemException;

 public boolean shutdownObject(org.omg.CORBA.Object obj);

 public boolean shutdownObject(Servant obj);
}
```

We shall now describe the various methods of the OA class.

- **`initPOA`** : this method activates the POA, if this method is called on the RootPOA the POA with the name RootPOA will be activated.
- **`createPOA`** : if a child POA with the specified name for the current POA has not already been created then this method will create and activate one, otherwise AdapterAlreadyExists will be thrown. This method returns a ChildOA object.
- **`initOA`** : this method calls the initPOA method and has been retained for backwards compatibility.
- **`rootPoa`** : this method returns a reference to the root POA. After destroy is called on the root POA this reference may be null.
- **`poa`** : this method returns a reference to the POA. After destroy is called this reference may be null.
- **`destroy`** : this method destroys the current POA, if this method is called on a RootPOA instance then the root POA will be destroyed along with its children.
- **`shutdown`** : this method shuts down the POA.
- **`addAttribute`** : this method allows the application to register classes with JBoss Transaction Service which will be called either before or after the OA has been initialised. See below. If the OA has already been initialised then the attribute object will not be added, and false will be returned.

2.1.1. ORB and OA Initialisation

It is possible to register application specific code with the ORB portability library which can be executed either before or after the ORB or OA are initialised. Application programs can inherit from either **`com.arjuna.orbportability.orb.Attribute`** or **`com.arjuna.orbportability.oa.Attribute`** and pass these instances to the addAttribute method of the ORB/OA classes respectively:

Example 2.3. **`Attribute.java`**

```
 package com.arjuna.orbportability.orb;
public abstract class Attribute {
 public abstract void initialise(String[] params);

 public boolean postORBInit();
};

package com.arjuna.orbportability.oa;
public abstract class OAAttribute {
 public abstract void initialise(String[] params);

 public boolean postOAInit();
};
```

By default, the **`postORBInit/postOAInit`** methods return true, which means that any instances of derived classes will be invoked after either the ORB or OA have been initialised. By redefining this to return false, a particular instance will be invoked before either the ORB or OA have been initialised.

When invoked, each registered instance will be provided with the exact String parameters passed to the initialise method for the ORB/OA.

2.1.2. ORB and OA shutdown

It is possible to register application specific code (via the **`addPreShutdown/addPostShutdown`** methods) with the ORB portability library which will be executed prior to, or after, shutting down the ORB. The pre/post interfaces which are to be registered have a single work method, taking no parameters and returning no results. When the ORB and OA are being shut down (using **`shutdown/destroy`**), each registered class will have its work method invoked.

Example 2.4. **`Shutdown.java`**

```
 public abstract class PreShutdown {
 public abstract void work();
}

public abstract class PostShutdown {
 public abstract void work();
}
```

2.1.3. Specifying the ORB to use

JDK releases from 1.2.2 onwards include a minimum ORB implementation from Sun. If using such a JDK in conjunction with another ORB it is necessary to tell the JVM which ORB to use. This happens by specifying the **`org.omg.CORBA.ORBClass`** and **`org.omg.CORBA.ORBSingletonClass`** properties. The ORB Portability classes will ensure that these properties are automatically set when required, i.e., during ORB initialisation. Of course it is still possible to specify these values explicitly (and necessary if not using the ORB initialisation methods). Note: if you do not use the ORB

Portability classes for ORB initialisation then it will still be necessary to set these properties. The ORB portability library attempts to detect which ORB is in use, it does this by looking for the ORB implementation class for each ORB it supports. This means that if there are classes for more than one ORB in the classpath the wrong ORB can be detected. Therefore it is best to only have one ORB in your classpath. If it is necessary to have multiple ORBs in the classpath then the property **`OrbPortabilityEnvironmentBean.orbImplementation`** must be set to the value specified in the table below.

ORB	Property Value
JacORB v2	**`com.arjuna.orbportability.internal.orbspecific`**
JDK miniORB	**`com.arjuna.orbportability.internal.orbspecific`**

2.1.4. Initialisation code

The *JBoss Transaction Service* requires specialised code to be instantiated before and after the ORB and the OA are initialised. This code can be provided at runtime through the use of OrbPortabilityEnvironmentBean.orbInitializationProperties This mechanism is also available to programmers who can register arbitrary code which the ORB Portability will guarantee to be instantiated either before or after ORB/OA initialisation. For each application (and each execution of the same application) the programmer can simultaneously provide multiple Java classes which are instantiated before and after the ORB and or OA is initialised. There are few restrictions on the types and numbers of classes which can be passed to an application at execution time. All classes which are to be instantiated must have a public default constructor, i.e., a constructor which takes no parameters. The classes can have any name. The property names used must follow the format specified below:

- *com..orbportability.orb.PreInit* – this property is used to specify a global pre-initialisation routine which will be run before any ORB is initialised.
- *com..orbportability.orb.PostInit* – this property is used to specify a global post-initialisation routine which will be run after any ORB is initialised.
- *com..orbportability.orb.<ORB NAME>.PreInit* – this property is used to specify a pre-initialisation routine which will be run when an ORB with the given name is initialised.
- *com..orbportability.orb.<ORB NAME>.PostInit* – this property is used to specify a post-initialisation routine which will be run after an ORB with the given name is initialised.
- *com..orbportability.oa.PreInit* – this property is used to specify a global pre-initialisation routine which will be run before any OA is initialised.
- *com..orbportability.oa.PostInit* – this property is used to specify a global post-initialisation routine which will be run after any OA is initialised,
- *com..orbportability.oa.<ORB NAME>.PreInit* – this property is used to specify a pre-initialisation routine which will be run before an OA with the given name is initialised
- *com..orbportability.oa.<ORB NAME>.PostInit* – this property is used to specify a pre-initialisation routine which will be run after an OA with the given name is initialised

Pre and post initialisation can be arbitrarily combined, for example:

```
java –
DorbPortabilityEnvironmentBean.orbInitializationProperties=”com..orbportability.orb.PreInit=org.foo.AllO
```

```
com..orbportability.orb.MyORB.PostInit=org.foo.MyOrbPostInit
com..orbportability.oa.PostInit=orb.foo.AllOAPostInit” org.foo.MyMainClass
```

2.1.5. Locating Objects and Services

Locating and binding to distributed objects within CORBA can be ORB specific. For example, many ORBs provide implementations of the naming service, whereas some others may rely upon proprietary mechanisms. Having to deal with the many possible ways of binding to objects can be a difficult task, especially if portable applications are to be constructed. ORB Portability provides the Services class in order to provide a more manageable, and portable binding mechanism. The implementation of this class takes care of any ORB specific locations mechanisms, and provides a single interface to a range of different object location implementations.

Example 2.5. **`Services.java`**

```
public class Services {
/**
 * The various means used to locate a service.
 */

public static final int RESOLVE_INITIAL_REFERENCES = 0;
public static final int NAME_SERVICE = 1;
public static final int CONFIGURATION_FILE = 2;
public static final int FILE = 3;
public static final int NAMED_CONNECT = 4;
public static final int BIND_CONNECT = 5;

public static org.omg.CORBA.Object getService(
  String serviceName, Object[] params,
  int mechanism) throws InvalidName,
  CannotProceed, NotFound, IOException;

public static org.omg.CORBA.Object getService(
  String serviceName, Object[] params)
  throws InvalidName, CannotProceed, NotFound,
  IOException;

public static void registerService(
  org.omg.CORBA.Object objRef,
  String serviceName, Object[] params,
  int mechanism) throws InvalidName, IOException,
  CannotProceed, NotFound;

public static void registerService(
  org.omg.CORBA.Object objRef,
  String serviceName, Object[] params)
  throws InvalidName, IOException, CannotProceed,
  NotFound;
}
```

There are currently several different object location and binding mechanisms supported by Services (not all of which are supported by all ORBs, in which case a suitable exception will be thrown):

1. *RESOLVE_INITIAL_REFERENCES* : if the ORB supported resolve_initial_references, then Services will attempt to use this to locate the object.

2. *NAME_SERVICE* : Services will contact the name service for the object. The name service will be located using **`resolve_initial_references`** .

3. *CONFIGURATION_FILE* : as described in the Using the OTS Manual, the JBoss Transaction Service supports an initial reference file where references for specific services and objects can be stored and used at runtime. The file, CosServices.cfg, consists of two columns: the service name (in the case of the OTS server TransactionService) and the IOR, separated by a single space. CosServices.cfg is located at runtime by the OrbPortabilityEnvironmentBean properties initialReferencesRoot (a directory, defaulting to the current working directory) and initialReferencesFile (a name relative to the directory,'CosServices.cfg' by default).

4. *FILE* : object IORs can be read from, and written to, application specific files. The service name is used as the file name.

5. *NAMED_CONNECT* : some ORBs support proprietary location and binding mechanisms.

6. *BIND_CONNECT* : some ORBs support the bind operation for locating services.

We shall now describe the various methods supported by the Services class:

- *getService* : given the name of the object or service to be located (serviceName), and the type of mechanism to be used (mechanism), the programmer must also supply location mechanism specific parameters in the form of params. If the name service is being used, then params[0] should be the String kind field.

- *getService* : the second form of this method does not require a location mechanism to be supplied, and will use an ORB specific default. The default for each ORB is shown in Table 2.

- *registerService* : given the object to be registered, the name it should be registered with, and the mechanism to use to register it, the application programmer must specify location mechanism specific parameters in the form of params. If the name service is being used, then params[0] should be the String kind field.

2.1.6. ORB location mechanisms

The following table summarises the different location mechanisms that ORB Portability supports for each ORB via the Services class:

Location Mechanism	ORB
CONFIGURATION_FILE	All available ORBs
FILE	All available ORBs
BIND_CONNECT	None

If a location mechanism isn’t specified then the default is the configuration file.

Appendix A. Revision History

Revision 1 **Wed Apr 13 2010** **Tom Jenkinson**
tom.jenkinson@redhat.com

Initial converstion to docbook