# BEA WebLogic Enterprise

# Using CORBA Server-to-Server Communication

WebLogic Enterprise 5.0
Document Edition 5.0
December 1999

**Using CORBA Server-to-Server Communication**

| Document Edition | Date | Software Version |
|---|---|---|
| 5.0 | December 1999 | BEA WebLogic Enterprise 5.0 |

# Contents

# About This Document

This document describes using the CORBA server-to-server functionality in the BEA WebLogic Enterprise (WLE) product. This document defines concepts associated with using server-to-server communication and describes the development process for Java and C++ joint client/server applications. In addition, instructions for building and running the Chat Room and Callback sample applications are included in this document.

This document covers the following topics:

- Chapter 1, "Understanding CORBA Server-to-Server Communication," explains the concepts you need to understand to use server-to-server communication and build joint client/server applications.
- Chapter 2, "Developing C++ Joint Client/Server Applications," describes building C++ joint client/server applications and how to build and run the Chat Room sample application.
- Chapter 3, "Developing Java Joint Client/Server Applications,"describes building Java joint client/server applications and how to build and run the Callback sample application.

## What You Need to Know

This document is intended for programmers who are interested in implementing CORBA server-to-server communication in their WLE applications.

## e-docs Web Site

The BEA WebLogic Enterprise product documentation is available on the BEA corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the "e-docs" Product Documentation page at http://e-docs.beasys.com.

## How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the WebLogic Enterprise documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Enterprise documentation Home page, click the PDF Files button, and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

## Related Information

For more information about CORBA, Java 2 Enterprise Edition (J2EE), BEA TUXEDO, distributed object computing, transaction processing, C++ programming, and Java programming, see the WLE Bibliography in the WebLogic Enterprise online documentation.

## Contact Us!

Your feedback on the BEA WebLogic Enterprise documentation is important to us. Send us e-mail at **docsupport@beasys.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Enterprise documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Enterprise 5.0 release.

If you have any questions about this version of BEA WebLogic Enterprise, or if you have problems installing and running BEA WebLogic Enterprise, contact BEA Customer Support through BEA WebSupport at www.beasys.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

## Documentation Conventions

The following documentation conventions are used throughout this document.

| Convention | Item |
|---|---|
| **boldface text** | Indicates terms defined in the glossary. |
| Ctrl+Tab | Indicates that you must press two or more keys simultaneously. |

| Convention | Item |
|---|---|
| *italics* | Indicates emphasis or book titles. |
| `monospace text` | Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.<br>*Examples*:<br>`#include <iostream.h> void main ( ) the pointer psz`<br>`chmod u+w *`<br>`\tux\data\ap`<br>`.doc`<br>`tux.doc`<br>`BITMAP`<br>`float` |
| **`monospace boldface text`** | Identifies significant words in code.<br>*Example*:<br>`void` **`commit`** `( )` |
| *`monospace italic text`* | Identifies variables in code.<br>*Example*:<br>`String` *`expr`* |
| UPPERCASE TEXT | Indicates device names, environment variables, and logical operators.<br>*Examples*:<br>LPT1<br>SIGNON<br>OR |
| `{ }` | Indicates a set of choices in a syntax line. The braces themselves should never be typed. |
| `[ ]` | Indicates optional items in a syntax line. The brackets themselves should never be typed.<br>*Example*:<br>`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`<br>`[-l` *`file-list`*`]...` |

| Convention | Item |
|---|---|
| \| | Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed. |
| `...` | Indicates one of the following in a command line:<br>■ That an argument can be repeated several times in a command line<br>■ That the statement omits additional optional arguments<br>■ That you can enter additional parameters, values, or other information<br>The ellipsis itself should never be typed.<br>*Example*:<br>`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`<br>`[-l` *`file-list`*`]...` |
| .<br>.<br>. | Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed. |

CHAPTER

# 1 Understanding CORBA Server-to-Server Communication

This chapter contains the following topics:

- Overview of CORBA Server-to-Server Communication
- Joint Client/Server Applications
- Object Policies for Callback Objects

## Overview of CORBA Server-to-Server Communication

CORBA Server-to-server communication allows WebLogic Enterprise (WLE) applications to invoke CORBA objects and handle invocations from those CORBA objects (referred to as callback objects). The CORBA objects can be either inside or outside of a WLE domain.

The WLE product offers an implementation of the Internet Inter-ORB Protocol (IIOP) Version 1.2, which provides inbound and outbound communication with the CORBA objects. Server-to-server communication provides more efficient use of network resources and provides integration with third-party Object Request Brokers (ORBs). In addition, server-to-server communication is supported with CORBA objects that are implemented using IIOP versions 1.0 and 1.1.

## Joint Client/Server Applications

In previous versions of the WLE product, client applications invoked operations defined in Object Management Group (OMG) Interface Definition Language (IDL) on a CORBA object. The server applications implemented the operations of the CORBA object. The CORBA objects in the server application used WLE TP Framework and environmental objects to implement state management, security, and transactions. These CORBA objects as referred to as WLE objects. Server applications could act as client applications for other server applications; however, client applications could not *act* as server applications for other client applications.

Server-to-server communication allows client applications to now act as server applications for requests from other client applications. In addition, server-to-server communication allows WLE server applications to invoke objects on other ORBs.

The server-to-server communication functionality is available through a callback object. A callback object has two purposes:

- It invokes operations on either WLE or CORBA objects.
- It implements the operations of a CORBA object.

Callback objects do not use the TP Framework and are not subject to WLE administration, they should be used when transactional behavior, security, reliability, and scalability are not important.

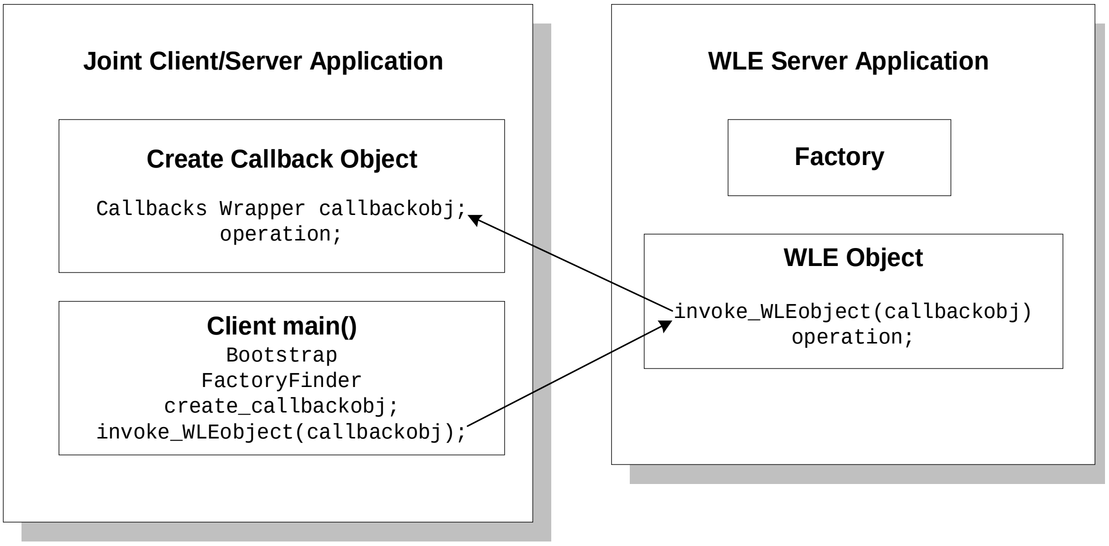
Callback objects are implemented in joint client/server applications. A joint client/server application consists of the following:

- A portion that performs WLE client application functions, such as initializing the ORB, using the WLE environmental objects to establish connections, resolving initial references to the FactoryFinder object, and using factories to create WLE objects

- A portion that creates a servant for a callback object and activates the callback object using an object ID

Figure 1-1 shows the structure of a joint client/server application.

**Figure 1-1 Structure of a Joint Client/Server Application**



C++ and Java joint client/server applications are supported.

Use of callback objects in Java applets is limited due to Java applet security mechanisms. Any Java applet run-time environment that allows a Java applet to create and listen on sockets (via the proprietary environment or protocol of the Java applet) can act as a joint client/server application. However, if the Java applet run-time environment restricts socket communication, the Java applet cannot act as a joint client/server application.

**Note:** The ActiveX client software that is included in the WLE V4.2 kit does not support callback objects, and, therefore cannot be used to develop joint client/server applications.

Joint client/server applications use IIOP to communicate with the WLE server applications. IIOP can work in the following ways, depending on the version of the IIOP protocol you are using:

- Bidirectional

  Joint client/server applications are always connected to the same IIOP Server Handler (ISH) in the WLE domain. That ISH reuses the same connection to send requests to and receive requests from the joint client/server application.

- Dual-paired connection

  Joint client/server applications use the `register_callback_port` method of the Bootstrap object to register the listening port of the joint client/server application in the ISH. Invocations from server applications on the callback object in the joint client/server application are routed through the ISH connected to the joint client/server application. This ISH uses a second outbound connection to send requests to and receive replies from the connected joint client/server application. The outbound connection is paired with the incoming connection. This differs from bidirectional IIOP, which uses only one connection.

- Asymmetric

  Joint client/server applications can invoke on any callback object, and are not restricted to invoking callback objects implemented in joint client/server applications connected to an ISH. Asymmetric IIOP forces the ORB infrastructure to search for an available ISH to handle the invocation.

For a more detailed description of bidirectional, dual-paired connnection, and asymmetric IIOP, see the *C++ Programming Reference* or the *Java Programming Reference*.

## Object Policies for Callback Objects

Callback objects are assigned policies that control how long an object reference is valid and how an object ID is assigned to the object. Object policies are defined when the reference to the callback object is created. In addition, they can be defined in the Callbacks Wrapper object, which simplifies the development of joint client/server applications.

The following object policies are supported for callback objects:

- Transient/System ID—The object reference for this type of callback object is valid only for the life of the joint client/server application. The object ID is assigned by the WLE system. This type of object is used for invocations that a joint client/server application wants to receive only until it terminates.
- Persistent/System ID—The object reference for this type of callback object is valid across multiple invocations in a joint client/server application. The object ID is assigned by the WLE system. This type of object is useful in joint client/server applications that stop and restart over a period of time. When the Joint client/server application is up, it can receive requests on a particular callback object with that object reference. Typically, the joint client/server application creates the object reference once, saves it in its own permanent storage area, and reactivates the servant for the object every time the joint client/server application comes up.
- Persistent/User ID—This object policy is the same as Persistent/System ID, except that the object ID is assigned by the joint client/server application.

When creating a callback object with an object policy of transient, the object reference is valid only until the joint client/server application is terminated or until the `stop_all_objects` method is called.

When creating a callback object with an object policy of persistent, the object reference is valid even after the termination of the joint client/server application. If the joint client/server application terminates, restarts, and activates a servant for the same object ID, the servant accepts requests made on that object reference.

**Note:** If you are creating a native joint client/server application (that is, a joint client/server application that is located in the same WLE domain as the WLE server applications that invokes it), you cannot use the Persistent/System ID or Persistent/User ID object policies.

*CHAPTER*

# 2 Developing C++ Joint Client/Server Applications

This chapter contains the following topics:

- Development Process
- Chat Room Sample Application
- Step 1: Writing the OMG IDL
- Step 2: Generating Skeletons and Client Stubs
- Step 3: Writing the Methods That Implement Each Object’s Operations
- Step 4: Writing the Client Portion of the Joint Client/Server Application
- Step 5: Creating a Callback Object Using the Callbacks Wrapper Object
- Step 6: Invoking Operations on a WLE Object By Passing a Reference to the Callback Object
- Step 7: Specifying Configuration Information
- Step 8: Compiling Joint Client/Server Applications
- Using the POA to Create a Callback Object
- Threading Considerations for C++ Joint Client/Server Applications
- Building and Running the Chat Room Sample Application

## Development Process

Table 2-1 outlines the development process for C++ joint client/server applications.

**Table 2-1 Development Process for C++ Joint Client/Server Applications**

| Step | Description |
|---|---|
| 1 | Write the OMG IDL for the callback interface and for the CORBA interfaces you want to use in your WLE application. |
| 2 | Generate the skeletons and client stubs. |
| 3 | Write the methods that implement each object's operations. |
| 4 | Write the client portion of the joint client/server application. |
| 5 | Create a callback object using the Callbacks Wrapper object. |
| 6 | Invoke operations on a WLE object by passing the object reference for the callback object. |
| 7 | Specifying configuration information. |
| 8 | Compile the joint client/server application. |

These steps are explained in detail in subsequent topics.

Because the callback object in a joint client/server application is not transactional and has no object management capabilities, you do not need to create an Implementation Configuration File (`*filename*.icf`) for it. However, you still need to create an ICF file for the WLE objects in your WLE application. For information about writing an ICF file, see *Creating CORBA C++ Server Applications*.

# Chat Room Sample Application

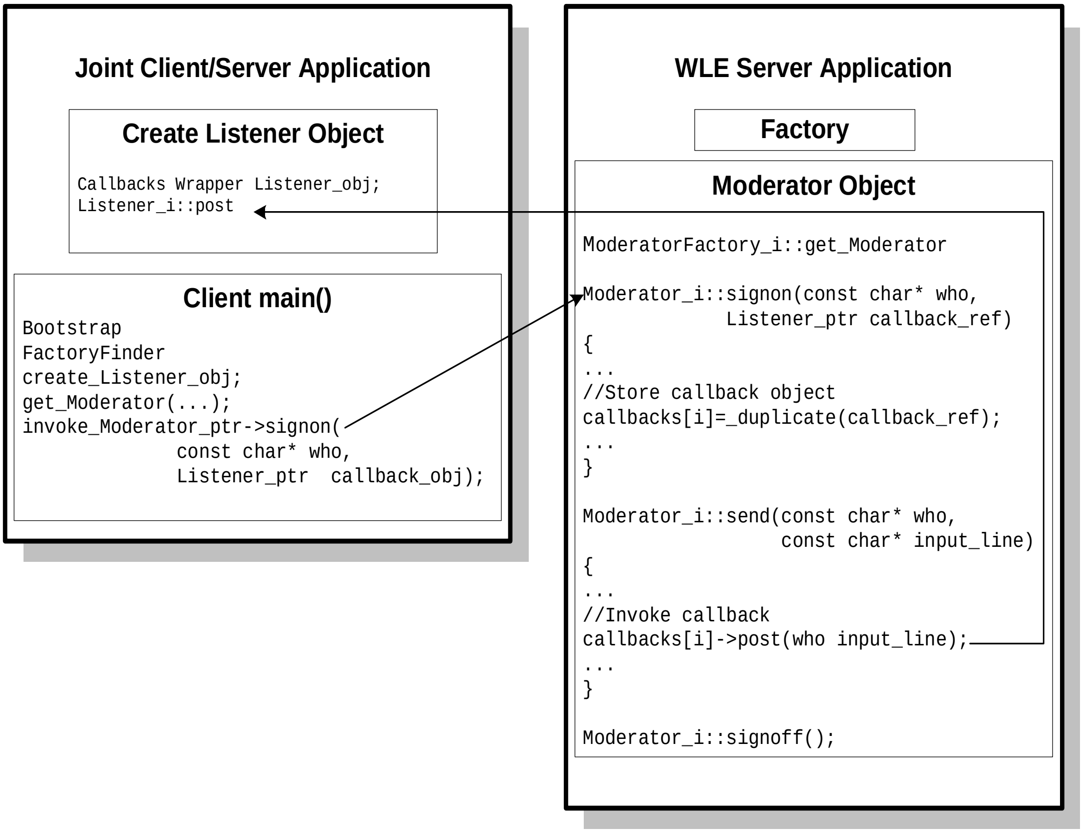
Throughout this topic, the Chat Room sample application is used to demonstrate the development steps. A chat room is an application that allows several people at different locations to communicate with each other. The chat room can be thought of as a moderator whose job it is to keep track of client applications that have logged in, and to distribute messages to those client applications.

A client application logs in to the moderator, supplying a user name. When messages are entered at the keyboard, the client application invokes the moderator, and passes the messages to the moderator. The moderator then distributes the messages to all the other client applications by making an invocation on the callback object.

The Chat Room sample application consists of a C++ joint client/server application and a WLE server application. The joint client/server application receives keyboard input and makes invocations on the moderator. The joint client/server application also sets up the callback object to listen for messages from the moderator (that is, to receive invocations from the moderator). The WLE server application in the Chat Room sample application implements the moderator.

Figure 2-1 illustrates how the Chat Room sample application works.

**Figure 2-1 How the Chat Room Sample Application Works**



```
Callbacks Wrapper Listener_obj;
Listener_i::post
```

```
Bootstrap
FactoryFinder
create_Listener_obj;
get_Moderator(...);
invoke_Moderator_ptr->signon(
              const char* who,
              Listener_ptr  callback_obj);
```

```
ModeratorFactory_i::get_Moderator

Moderator_i::signon(const char* who,
             Listener_ptr callback_ref)
{
...
//Store callback object
callbacks[i]=_duplicate(callback_ref);
...
}

Moderator_i::send(const char* who,
                  const char* input_line)
{
...
//Invoke callback
callbacks[i]->post(who input_line);
...
}

Moderator_i::signoff();
```

The Chat Room sample application works as follows:

1. The joint client/server application implements the logic for the callback object (the Listener object), creates a servant for the Listener object, and activates the Listener object.
2. The joint client/server application creates an object reference for the Listener object and passes it to the Moderator object as part of the `signon` operation.
3. The server application in the Chat Room sample application checks the keyboard for messages.
4. When messages are generated at the keyboard, the Chat Room sample application sends the messages to the Moderator object via the `send` operation.

5. The Chat Room sample application temporarily passes control over to the ORB to allow the Listener object in the joint client/server application to receive `post` invocations from the Moderator object.

6. The Listener object in the joint client/server application saves the posted messages until a client application requests them.

The source files for the Chat Room sample application are located in the `WLEdir\samples\corba\chatroom` directory in the WLE software directory. See "Building and Running the Chat Room Sample Application" on page 2-24 for more information.

## Step 1: Writing the OMG IDL

You use Object Management Group (OMG) Interface Definition Language (IDL) to describe available CORBA interfaces to client applications. An interface definition written in OMG IDL completely defines the CORBA interface and fully specifies each operation's arguments. OMG IDL is a purely declarative language. This means that it contains no implementation details. For more information about OMG IDL, see *Creating CORBA Client Applications*.

The Chat Room sample application implements the CORBA interfaces listed in Table 2-2.

**Table 2-2 CORBA Interfaces for the Chat Room Sample Application**

| Interface | Description | Operation |
|---|---|---|
| `Listener` | The callback object | `post()` |
| `Moderator` | Receives input from client applications and uses the callback object to forward messages back to the joint client/server application | `signon()`<br>`send()`<br>`signoff()` |
| `ModeratorFactory` | Creates object references to the Moderator object | `get_moderator()` |

Listing 2-1 shows the `chatclient.idl` that defines the Listener interface.

**Listing 2-1 OMG IDL for the Listener Interface**

```
module ChatClient{
       interface Listener  {
              oneway void post (in string from,
                                in string output_line);
       };
};
```

Listing 2-2 shows the `chatroom.idl` that defines the Moderator and ModeratorFactory interfaces for the Chat Room sample application. The `#include` is used to resolve references to interfaces in another OMG IDL file. In the Chat Room sample application, the `signon` method requires a Listener object as a parameter and, therefore, must use the `#include` to reference the OMG IDL file that defines the Listener interface.

**Listing 2-2 OMG IDL for the Moderator and ModeratorFactory Interfaces**

```
#include "ChatClient.idl"

module ChatRoom {

       interface Moderator {
          exception IdAlreadyUsed{};
          exception NoRoomLeft{};
          exception IdNotKnown{};

          void signon(in string                 who,
                      in ChatClient::Listener   callback_ref )
                           raises( IdAlreadyUsed, NoRoomLeft );

          void send  (in string                 who,
                     in string                  input_line )
                           raises( IdNotKnown );

          void signoff(in string                who )
                           raises( IdNotKnown );
       };
```

```
        interface ModeratorFactory {
           Moderator get_moderator( in string chatroom_name );
        };
};
```

## Step 2: Generating Skeletons and Client Stubs

The interface specification defined in OMG IDL is used by the IDL compiler to generate skeletons and client stubs. Note that a joint client/sever application uses the client stub for the WLE object and the skeleton and client stub for the callback object.

For example, in the Chat Room sample application, the joint client/server application uses the skeleton and client stub for the Listener object (that is, the callback object) to implement the object. The joint client/server application also uses the client stubs for for the Moderator and ModeratorFactory to invoke operations on the objects. The WLE server application uses the skeletons for the Moderator and ModeratorFactory objects to implement the objects and the client stub for the Listener object to invoke operations on the object.

During the development process, use the `idl` command with the `-P` and `-i` options to compile the OMG IDL file that defines the callback object (for example, the `chatclient.idl` file in the Chat Room sample application). The options work as follows:

- The `-P` option creates a skeleton class that inherits directly from the `PortableServer::ServantBase` class. Inheriting from `PortableServer::ServantBase` means the joint client/server application must explicitly create a servant for the callback object and initialize the servant's state. The servant for the callback object cannot use the `activate_object` and `deactivate_object` methods as they are members of the `PortableServer::ServantBase` class.
- The `-i` option results in an implementation template file being generated. This file is a template for the code that implements the interfaces defined in OMG IDL for the Listener object.

You then need to compile the OMG IDL file that defines the interfaces in the WLE server application (for example, the `chatroom.idl` file in the Chat Room sample application). Use the `idl` command with only the `-i` option to compile that OMG IDL file.

Table 2-3 lists the files that are created by the `idl` command.

**Note:** In the Chat Room sample application, the generated template files for the `ChatClient.idl` and `ChatRoom.idl` files have been renamed to reflect the objects (Listener and Moderator) they implement. In addition, the template file for the Moderator object includes the implementation for the ModeratorFactory object.

**Table 2-3 Files Produced by the `idl` Command**

| File | File in the Chat Room Sample Application Created by the `idl` Command | Description |
|---|---|---|
| Client stub file | `Listener_c.cpp`<br>`Listener_c.h`<br>`Moderator_c.cpp`<br>`Moderator_c.h` | Contains client stubs for each interface specified in the OMG IDL file. The client stubs are used to send a request to an object. |
| Implementation file | `Listener_i.cpp`<br>`Moderator_i.cpp` | Contains signatures for the methods that implement the operations of the `Listener`, `Moderator`, and `ModeratorFactory` interfaces specified in the OMG IDL file. The `Listener_i.h` file contains implementation files that inherit from the `POA_ChatClient::Listener` class. |
| Skeleton file | `Listener_s.cpp`<br>`Listener_s.h`<br>`Moderator_s.cpp`<br>`Moderator_s.h` | Contains skeletons for each interface specified in the OMG IDL file. During run time, the skeleton maps client requests to the appropriate operation in the server application. The `Listener_s.h` file contains `POA_`*`skeleton`* class definitions (for example, `POA_ChatClient::Listener`). |

## Step 3: Writing the Methods That Implement Each Object's Operations

After you compile each of the OMG IDL files, you need to write methods that implement the operations for each object. In a joint client/server application, you write the implementation file for the callback object (that is, the Listener object). You write the implementation for a callback object as you would write the implementation for any other CORBA object, except that you use the POA instead of the TP Framework. You also write implementation files for the WLE objects (that is, the Moderator and ModeratorFactory objects) in the WLE server application.

An implementation file contains the following:

- Method declarations for each operation specified in the OMG IDL file
- Your application's business logic
- Constructors for each interface implementation (implementing these is optional)
- Optionally, for WLE objects, the `com.beasys.Tobj_Servant.activate_object` and `com.beasys.Tobj_Servant.deactivate_object` methods

  Within the `activate_object` and `deactivate_object` methods, you write code that performs any particular steps related to activating or deactivating an object.

Listing 2-3 includes the implemention file for the Listener object, and Listing 2-4 includes the implementation file for the Moderator and ModeratorFactory objects.

**Note:** Additional methods and data were added to the implementation file for the Moderator and ModeratorFactory objects. The template for the implementation file was created by the `idl -i` command.

**Listing 2-3 Implementation File for the Listener Object**

```
//This module contains the definition of the implementation class
//Listener_i

#ifndef _Listener_i_h
#define _Listener_i_h

#include "ChatClient_s.h

class Listener_i : public POA_ChatClient::Listener {
       public:

              Listener_i ();
              virtual ~Listener_i();

              void post (
                     const char * from,
                     const char * output_line);
...
};

#endif
```

**Listing 2-4 Implementation File for Moderator and ModeratorFactory Objects**

```
//This module contains the definition of the implementation class
//Moderator and ModeratorFactory

#ifndef _Moderator_i_h
#define _Moderator_i_h

#include "ChatRoom_s.h"

const int CHATTER_LIMIT = 5;          // the most chatters allowed

class Moderator_i : public POA_ChatRoom::Moderator {
       public:

//Define the operations

       void signon ( const char*      who,
                     ChatClient::Listener_ptr   callback_ref);

       void send ( const char *       who,
                   const char *       input_line);
       void signoff ( const char * who);

//Define the Framework functions

       virtual void activate_object  ( const char* stroid );
       virtual void deactivate_object( const char* stroid,
                                   TobjS::DeactivateReasonValue
                                   reason);
       private:

//Define function to find name on list
       int find( const char * handle );

//Define name of the chat room overseen by the Moderator
       char* m_chatroom_name;

//Data for maintaining list

//Chatter[n] id
       CORBA::String            chatters[CHATTER_LIMIT];

//Chatter[n] callback ref
       ChatClient::Listener_var callbacks[CHATTER_LIMIT];
};

class ModeratorFactory_i : public POA_ChatRoom::ModeratorFactory {
       public:
        ChatRoom::Moderator_ptr get_moderator ( const char*
                                                chatroom_name );
};
#endif
```

## Step 4: Writing the Client Portion of the Joint Client/Server Application

During development of a joint client/server application, you write the client portion of the joint client/server application as you would write any WLE client application. The client application needs to include code that does the following:

1. Initializes the ORB. The WLE system activates an ORB using the correct protocol (in this case, IIOP).

2. Uses the Bootstrap object to establish communication with the WLE domain.
3. Resolves initial references to the FactoryFinder object.
4. Uses a factory to get an object reference for the desired WLE object (that is, the Moderator object).

The client development steps are illustrated in Listing 2-5, which includes code from the Chat Room sample application. In the Chat Room sample application, the client portion of the joint client/server application uses a factory to get an object reference to the Moderator object, and then invokes the `sign_on()`, `send()`, and `sign_off()` methods on the Moderator object.

**Listing 2-5 Client Portion of the Chat Room Joint Client/Server Application**

```
...

//Initialize the ORB

orb_ptr = CORBA::ORB_init(argc, argv, “BEA_IIOP”);

//Create a Bootstrap object to establish communication with the
//WLE domain

bootstrap = new Tobj_Bootstrap(orb_ptr,"");

//Get a FactoryFinder object, use it to find a Moderator factory,
//and get a Moderator.

//Use the Bootstrap object to find the FactoryFinder object

CORBA::Object_var var_factory_finder_oref =
           bootstrap->resolve_initial_references("FactoryFinder");

//Narrow the FactoryFinder object

Tobj::FactoryFinder_var var_factory_finder =
       Tobj::FactoryFinder::_narrow(var_factory_finder_oref.in());

//Use the FactoryFinder object to find a factory for the Moderator

CORBA::Object_var var_moderator_factory_oref =
       var_factory_finder->find_one_factory_by_id(
       "ModeratorFactory" );

//Narrow the Moderator Factory
```

```
ChatRoom::ModeratorFactory_var var_moderator_factory =
       ChatRoom::ModeratorFactory::_narrow(
       var_moderator_factory_oref.in() );

//Get a Moderator
//The chatroom name is passed as a command line parameter

var_moderator_oref =
            var_moderator_factory->get_moderator
            (var_chat_room_name.in() );

...
```

## Step 5: Creating a Callback Object Using the Callbacks Wrapper Object

Since the basic steps for creating a callback object are always the same, the WLE product provides a Callbacks Wrapper object that simplifies the development of callback objects.

The Callbacks Wrapper object does the following:

- Defines the object policy for the callback object. The following object policies are supported:
  - Transient/SystemID (`_transient`)
  - Persistent/SystemId (`_persistent/systemid`)
  - Persistent/UserId (`_persistent/userid`)

  For a complete description of the object policies for callback objects, see "Object Policies for Callback Objects" on page 1-4.
- Creates a servant for the callback object.
- Sets the ORB and the POA to the state in which they will accept requests on the callback object.

- Returns an object reference to the activated callback object. The object Id can be generated by the system or supplied by the user.
- Tells the ORB to stop accepting requests on either a single servant or all the active servants.

For a complete description of the Callbacks Wrapper object and its methods, see the *CORBA C++ Programming Reference*.

Listing 2-6 shows how a Callbacks Wrapper object is used in the Chat Room sample application.

**Listing 2-6 Using the Callbacks Wrapper Object in the Chat Room Sample Application**

```
...

//Use the Callbacks object to create a servant for the
//Listener object, activate the Listener object, and create an
//object reference for the Listener object.

BEAWrapper::Callbacks* callbacks =
                       new BEAWrapper::Callbacks( orb_ptr );
Listener_i * listener_callback_servant = new Listener_i();
CORBA::Object_var v_listener_oref=callbacks->start_transient(
                                listener_callback_servant,
                                ChatClient::_tc_Listener->id());
ChatClient::Listener_var v_listener_callback_oref =
                                ChatClient::Listener::_narrow(
                                var_listener_oref.in());
...
```

## Step 6: Invoking Operations on a WLE Object By Passing a Reference to the Callback Object

Once you have an object reference to a callback object, you can pass the callback object reference as a parameter to a method of a WLE object. In the Chat Room sample application, the Moderator object uses an object reference to the Listener object as a parameter to the `sign_on` method. Listing 2-7 illustrates this step.

**Listing 2-7 Invoking the signon Method**

```
//Sign on to the Chat room using a user-defined handle and a
//reference to the Listener object (the callback object) to receive
//input from other client applications logged into the Chat room.

var_moderator_reference->signon(handle,
                                var_listener_callback_oref.in() );
```

## Step 7: Specifying Configuration Information

When running remote joint client/server applications that use IIOP, the object references for the callback object must contain a host and port number, as follows.

- For transient callback objects, any port is sufficient and can be obtained dynamically by the ORB.
- For persistent callback objects, the ORB must be configured to accept requests for the callback object on the same port on which the object reference for the callback object was created.

The user specifies the port number from the user range of port numbers, rather than from the dynamic range. Assigning port numbers from the user range prevents joint client/server applications from using conflicting ports. To specify a particular port for the joint client/server application to use, include the following on the command line that starts the process for the joint client/server application:

```
-ORBport nnn
```

where *nnn* is the number of the port to be used by the ORB when creating invocations and listening for invocations on the callback object in the joint client/server application.

Use this command when you want the object reference for the callback object in a joint client/server application to be persistent and when you want to stop and restart the joint client/server application. If this command is not used, the ORB uses a random port. If the joint client/server application is stopped and then started, invocations to callback objects in the the joint client/server application will fail.

The port number is part of the input to the `argv` argument of the `CORBA::orb_init` member function. When the `argv` argument is passed, the ORB reads that information, establishing the port for any object references created in that process. You can also use the Bootstrap object's `register_callback_port` operation for the same purpose.

For a joint client/server application to communicate with a WLE object in the same WLE domain, a configuration file for the WLE server application is needed. The configuration file should be written as part of the development of the WLE server application. The binary version of the configuration file, the `TUXCONFIG` file, must exist before the joint client/server application is started. The `TUXCONFIG` file is created using the `tmloadcf` command. For information about creating a `TUXCONFIG` file, see *Getting Started* and the *Administration Guide*.

If you are using a joint client/server application that uses IIOP version 1.0 or 1.1, the administrator needs to boot the IIOP Server Listener (ISL) with startup parameters that enable outbound IIOP to invoke callback objects not connected to an IIOP Server Handler (ISH). The `-O` option of the ISL command enables outbound IIOP. Additional parameters allow administrators to obtain the optimum configuration for their WLE application. For more information about the ISL command, see the *Administration Guide*.

## Step 8: Compiling Joint Client/Server Applications

The final step in the development of a joint client/server application is to produce the executable. To do this, you need to compile the code and link against the skeleton and client stub.

Use the `buildobjclient` command with the `-P` option to construct a joint client/server application executable. To form an executable, the command combines the client stub for the WLE object, the client stub for the callback object, the skeleton for the callback object, and the implementation for the callback object with the appropriate POA libraries.

**Note:** To use the `-P` option of the `buildobjclient` command, you need to have used the `-P` option of the `idl` command when you created the skeleton and client stub for the callback object.

## Using the POA to Create a Callback Object

You can use the POA directly to create a callback object. You would use the POA directly when you want to use POA features and object policies not available through the Callbacks Wrapper object. For example, if you want to use the POA optimization features, you need to use the POA directly. The following topics describe how to use the POA to create callback objects with the supported object policies.

**Note:** Only a subset of the POA interfaces are supported in WLE version 4.2. For a list of support interfaces, see the *CORBA C++ Programming Reference*.

## Creating a Callback Object with a Transient Object Policy

To use the POA to create a callback object with a transient object policy, you need to write code that does the following:

1. Establishes a connection with a POA.
2. Creates a child POA.

   Since the root POA does not allow use of bidirectional IIOP, you need to create a child POA. The child POA can use the defaults for `LifespanPolicy` (`TRANSIENT`) and `IDAssignmentPolicy` (`SYSTEM`). You need to specify a `BiDirPolicy` policy of `BOTH`.

   IIOP version 1.2 supports reuse of the TCP/IP connection for both incoming and outgoing requests. Allowing reuse of a TCP/IP connection is the choice of the ORB. To allow reuse, you create an ORB policy object that allows reuse of a TCP/IP connection, and you use that policy object in the list of policies when initializing an ORB. The policy object is created using the `CORBA::ORB::create_policy` operation. For more information about the `CORBA::ORB::create_policy` operation, see the *CORBA C++ Programming Reference*.
3. Creates a servant for the callback object.
4. Informs the POA that the servant is ready to accept requests for the callback object.

   In this step, the joint client/server application activates the callback object in the POA using an object ID.
5. Activates the POA.
6. Creates an object reference for the callback object.
7. Makes an invocation on a WLE object using the object reference for the callback object as a parameter to one of the methods of the WLE object.

Listing 2-8 shows the portion of the Chat Room sample application that uses the POA to create the Listener object.

**Listing 2-8 Using the POA to Create the Listener Object**

```
//Establish communication with the POA

orb_ptr = CORBA::ORB_init(argc, argv, "BEA_IIOP");
CORBA::PolicyListpolicy_list(1);
CORBA::Any val;

CORBA::Object_ptr o_init_poa;
o_init_poa = orb_ptr->resolve_initial_references("RootPOA");

// Narrow to get the Root POA

root_poa_ptr = PortableServer::POA::_narrow(o_init_poa);
CORBA::release(o_init_poa);


//Specify an IIOP Policy of Bidirectional for the POA

val <<= BiDirPolicy::BOTH;
CORBA::Policy_ptr bidir_pol_ptr = orb_ptr->create_policy(
                     BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE, val);
policy_list.length ( 1 );
policy_list[0] = bidir_pol_ptr;

//Create the BiDirectional POA

bidir_poa_ptr = root_poa_ptr->create_POA("BiDirPOA",
                                         root_poa_ptr->
                                         the_POAManager(),
                                         policy_list);
//Activate the POA

root_poa_ptr->the_POAManager()->activate();

//Create the Listener object

ChatClient::Listener_var v_listener_callback_ref;

//Create a servant for Listener object and activate it

listener_callback_servant = new Listener_i();
  CORBA::Object_var           v_listener_oref;
  PortableServer::ObjectId_var temp_OId =
     bidir_poa_ptr ->activate_object(listener_callback_servant );

//Create object reference for the Listener object with a
```

```
//system generated Object Id

v_listener_oref = bidir_poa_ptr->create_reference_with_id
                                (temp_OId,
                                 ChatClient::_tc_Listener->id() );
v_listener_callback_ref = ChatClient::Listener::_narrow
                                ( v_listener_oref.in() );
```

## Creating a Callback Object with a Persistent/User ID Object Policy

To use the POA to create a callback object with a Persistent/User ID object policy, you need to write code that does the following:

1. Uses a string to store the user ID and converts the string to the object ID.
2. Creates a child POA with a `LifespanPolicy` set to `PERSISTENT` and `IDAssignmentPolicy` set to `USERID`.
3. Creates a servant for the Listener object.
4. Creates an object reference for the Listener object using the stringified object ID and the repository Id of the Listener object.
5. Activates the Listener object.

**Note:** The Persistent/User ID object policy is only used with remote joint client/server applications (that is, a joint client/server application that is not in a WLE domain).

Listing 2-9 shows code that performs these steps.

**Note:** The code example does not use bidirectional IIOP.

**Listing 2-9 Example Code for Listener Object with Persistent/User ID Object Policy**

```
//Declare a string and convert it to an object Id.
const char* oid_string = “783”;
PortableServer::ObjectID_var oid=
PortableServer::string_to_ObjectId(oid_string);

//Find the root POA
CORBA::Object_var oref =
orb_ptr->resolve_initial_references(“RootPOA”);
PortableServer::POA_var root_poa =
PortableServer::POA::_narrow(oref);

//Create and activate a Persistent/UserID POA
CORBA::PolicyList policies(2);
policies.length(2);
policies[0] = root_poa->create_lifespan_policy(
              PortableServer::PERSISTENT);
policies[1] = root_poa->create_id_assignment_policy(
              PortableServer::USER_ID );
PortableServer::POA_var poa_ref =
              root_poa->create_POA(“poa_ref”,
              root_poa->the_POAManager(),policies);
root_poa->the_POAManager()->activate();

//Create object reference for the Listener object.
oref = poa_ref->create_reference_with_id(oid,
              ChatClient::_tc_Listener->id());
ChatClient::Listener_ptr Listener_oref =
              ChatClient::Listener::_narrow( oref );

//Create Listener_i servant and activate the Listener object
Listener_i* my_Listener_i = new Listener_i();
poa_ref->activate_object_with_id( oid, my_Listener_i);

//Make call passing the reference to the Listener object
v_moderator_ref->signon( handle, Listener_oref);
```

## Creating a Callback Object with a Persistent/System ID Object Policy

To use the POA to create a callback object with a Persistent/System ID object policy, you need to write code that does the following:

1. Creates a child POA with a `LifespanPolicy` set to `PERSISTENT` and `IDAssignmentPolicy` set to the default.
2. Creates a servant for the Listener object.
3. Creates an object reference for the Listener object using a system generated object Id (the repository Id of the Listener object).
4. Activates the Listener object.

**Note:** The Persistent/System ID object policy is only used with remote joint client/server applications (that is, a joint client/server application that is not in a WLE domain).

Listing 2-10 shows code that performs these steps.

**Listing 2-10 Example Code for Listener Object with Persistent/System ID Object Policy**

```
//Find the root POA
CORBA::Object_var oref=
       orb_ptr->resolve_initial_references(“RootPOA”)
PortableServer::POA_var root_poa =
PortableServer::POA::_narrow(oref);

//Create and activate a Persistent/System ID POA
CORBA::PolicyList policies(1);
policies.length(1);
policies[0] = root_poa->create_lifespan_policy(
PortableServer::PERSISTENT);
//IDAssignmentPolicy is the default so you do not need to specify it
PortableServer::POA_var poa_ref = root_poa->create_POA(
       “poa_ref”, root_poa->the_POAManager(), policies);
root_poa->the_POAManager()->activate();

//Create Listener_i servant and activate the Listener object
```

```
Listener_i* my_Listener_i = new Listener_i();
PortableServer::ObjectId_var temp_OId =
       poa_ref->activate_object ( my_Listener_i );

//Create object reference for Listener object with returned
//system object Id
oref =
poa_ref->create_reference_with_id(
     temp_OId, ChatClient::_tc_Listener->id() );
ChatClient::Listener_var Listener_oref =
                            ChatClient::Listener::_narrow(oref);

//Make the call passing the reference to the Listener object
v_moderator_ref->signon( handle, Listener_oref );
```

## Threading Considerations for C++ Joint Client/Server Applications

A joint client/server application may first function as a client application and then switch to functioning as a server application. To do this, the joint client/server application turns complete control of the thread to the ORB by making the following invocation:

```
orb -> run();
```

If a method in the server portion of a joint client/server application invokes `ORB::shutdown()`, all server activity stops and control is returned to the statement after `ORB::run()` is invoked in the server portion of the joint client/server application. Only under this condition does control return to the client functionality of the joint client/server application.

Since a client application has only a single thread, the client functionality of the joint client/server application must share the central processing unit (CPU) with the server functionality of the joint client/server application. This sharing is accomplished by occasionally checking with the ORB to see if the joint client/server application has server application work to perform. Use the following code to perform the check with the ORB:

```
if ( orb->work_pending() ) orb->perform_work();
```

After the ORB completes the server application work, the ORB returns to the joint client/server application, which then performs client application functions. The joint client/server application must remember to occasionally check with the ORB; otherwise, the joint client/server application will never process any invocations.

You should be aware that the ORB cannot service callbacks while the joint client/server application is blocking on a request. If a joint client/server application invokes an object in another WLE server application, the ORB blocks while it waits for the response. While the ORB is blocking, it cannot service any callbacks, so the callbacks are queued until the request is completed.

## Building and Running the Chat Room Sample Application

Perform the following steps to build and run the Chat Room sample application:

1. Copy the files for the Chat Room sample application into a work directory.
2. Change the protection attribute on the files for the Chat Room sample application.
3. Verify the environment variables.
4. Execute the `ChatSetup` command.

The following sections describe these steps.

## Copying the Files for the Chat Room Sample Application into a Work Directory

You need to copy the files for the Chat Room sample application into a work directory on your local machine. The files for the Chat Room sample application are located in the following directories:

**Windows NT**

`drive:\WLEdir\samples\corba\chatroom`

**UNIX**

`/usr/local/WLEdir/samples/corba/chatroom`

You will use the files listed in Table 2-4 to build and run the Chat Room sample application.

**Table 2-4 Files Included in the Chat Room Sample Application**

| File | Description |
|---|---|
| `ChatRoom.idl` | The OMG IDL code that declares the `Moderator` and `ModeratorFactory` interfaces. |
| `ChatClient.idl` | The OMG IDL code that declares the `Listener` interface. |
| `Listener_i.h`<br>`Listener_i.cpp` | The C++ source code for method implementations of the Listener object in the joint client/server application. |
| `Moderator_i.h`<br>`Moderator_i.cpp` | The C++ source code for method implementations of the Moderator and ModeratorFactory objects in the WLE server application. |
| `ChatClientMain.cpp` | The C++ source code for the joint client/server application. |
| `ChatRoomServer.cpp` | The C++ source code for the WLE server application. |

**Table 2-4 Files Included in the Chat Room Sample Application**

| File | Description |
|---|---|
| `KeyboardManager.h`<br>`KeyboardManager.cpp` | The C++ source code that handles input from the keyboard in the Chat Room sample application. This code is used by `ChatClientMain.cpp`. |
| `ChatRoom.icf` | The Implementation Configuration File (ICF) for the Moderator and ModeratorFactory objects in the WLE server application in the Chat Room sample application. |
| `ChatRoom.ksh` | A UNIX script that sets the environment variables and builds the Chat Room sample application. |
| `ChatRoom.cmd` | An MS-DOS command procedure that sets the environment variables and builds the Chat Room sample application. |
| `ChatRoom.mk` | The UNIX operating system `makefile` for the Chat Room sample application. |
| `ChatRoom.nt` | The Windows NT operating system `makefile` for the Chat Room sample application. |
| `Readme.txt` | The file that provides the latest information about building and running the Chat Room sample application. |

## Changing the Protection Attribute on the Files for the Chat Room Sample Application

During the installation of the WLE software, the sample application files are marked read-only. Before you can edit or build the files in the Chat Room sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

**Windows NT**

```
prompt>attrib -r drive:\workdirectory\*.*
```

**UNIX**

```
prompt>/bin/ksh
```

```
ksh prompt>chmod u+w /workdirectory/*.*
```

On the UNIX operating system platform, you also need to change the permission of `ChatRoom.ksh` to give execute permission to the file, as follows:

```
ksh prompt>chmod +x ChatRoom.ksh
```

## Verifying the Setting of the TUXDIR Environment Variable

Before building and running the Chat Room sample application, you need to ensure that the `TUXDIR` environment variable is set on your system. In most cases, this environment variable is set as part of the installation procedure. The `TUXDIR` environment variable defines the directory path where you installed the WLE software. For example:

**Windows NT**

```
TUXDIR=c:\WLEDir
```

**UNIX**

```
TUXDIR=/usr/local/WLEDir
```

To verify that the information for the environment variables defined during installation is correct, perform the following steps:

**Windows NT**

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.

   The Control Panel appears.
3. Click the System icon.

   The System Properties window appears.
4. Click the Environment tab.

The Environment page appears.

5. Check the setting for `TUXDIR`.

**UNIX**

```
ksh prompt>printenv TUXDIR
```

To change the settings, perform the following steps:

**Windows NT**

1. On the Environment page in the System Properties window, click the `TUXDIR` environment variable.
2. Enter the correct information for the environment variable in the Value field.
3. Click OK to save the changes.

**UNIX**

```
ksh prompt>export TUXDIR=directorypath
```

## Executing the ChatSetup Command

The `ChatSetup` command automates the following steps:

1. Setting the system environment variables
2. Creating and loading the configuration file
3. Compiling the code for the client application
4. Compiling the code for the server application

Before running the `ChatSetup` command, you need to check the following:

- Ensure that you have the appropriate administrative privileges to build and run applications.
- On Windows NT, make sure `nmake` is in the path of your machine.
- On UNIX, make sure `make` is in the path of your machine.

To build and run the sample application, enter the `ChatSetup` command, as follows:

**Windows NT**

```
prompt>cd workdirectory
```

```
prompt>ChatSetup.cmd
```

**UNIX**

```
ksh prompt>cd workdirectory
```

```
ksh prompt>./ChatSetup.ksh
```

## Starting the Server Application

Start the server application and the system server processes in the Chat Room sample application by entering the following command:

```
prompt>tmboot -y
```

This command starts the following server processes:

- `TMSYSEVT`

  The system event broker. This server process is used only by the WLE system.

- `TMFFNAME`

  The following three `TMFFNAME` server processes are started:

  - The `TMFFNAME` server process started with the `-N` and `-M` options is the Master NameManager service. The NameManager service maintains a mapping of the application-supplied names to object references. This server process is used only by the WLE system.
  - The `TMFFNAME` server process started with only the `-N` option is the Slave NameManager service.
  - The `TMFFNAME` server process started with the `-F` option contains the FactoryFinder object.

- `ChatRoom`

  The server application process for the Chat Room sample application.

- `ISL`

  The IIOP Listener/Handler process.

## Starting the Client Application

Start the client application in the Chat Room sample application by entering the following command:

```
prompt>ChatClient chatroom_name -ORBport nnn
```

where *chatroom_name* is the name of a chat room to which you want to connect. You can enter any value. You will be prompted for a handle to identify yourself. You can enter any value. If the handle you chose is in use, you will be prompted for another handle.

To optimize the usefulness of the Chat Room sample application, you should run a second client application using the same chat room name.

To exit the client application, enter `\`.

## Stopping the Chat Room Sample Application

Before using another sample application, enter the following commands to stop the Chat Room sample application and to remove unnecessary files from the work directory:

**Windows NT**

```
prompt>tmshutdown -y

prompt>Admin\setenv

prompt>nmake -f ChatRoom.nt superclean

prompt>nmake -f ChatRoom.nt adminclean
```

**UNIX**

```
ksh prompt>tmshutdown -y

ksh prompt>. ./Admin/setenv.ksh

ksh prompt>make -f ChatRoom.mk superclean

ksh prompt>make -f ChatRoom.nt adminclean
```

*CHAPTER*

# 3 Developing Java Joint Client/Server Applications

This chapter contains the following topics:

- Building and Running the Callback Sample Application
- Using the Callback Sample Application

## Development Process

Table 3-1 outlines the development process for Java joint client/server applications.

**Table 3-1 Development Process for Java Joint Client/Server Applications**

| Step | Description |
|---|---|
| 1 | Write the OMG IDL for the callback interface and the CORBA interfaces you want to use in your WLE application. |
| 2 | Generate the skeletons and client stubs. |
| 3 | Write the methods that implement each interface’s operations. |
| 4 | Initialize the ORB. |
| 5 | Write the client main portion of the joint client/server application. |
| 6 | Create a callback object using the Callbacks Wrapper object. |
| 7 | Establish communication with an ISH. |
| 8 | Invoke operations on the WLE object by passing an object reference for the callback object. |
| 9 | Specify configuration information. |
| 10 | Compile the joint client/server application. |

These steps are explained in detail in subsequent topics.

Because the callback object in a joint client/server application is not transactional and has no object management capabilities, you do not need to create a Server Description File (`filename.xml`) for it. However, you still need to create a Server Description File for the WLE objects in your WLE application. For information about writing a Server Description File, see *Creating CORBA Java Server Applications*.

## Software Requirements

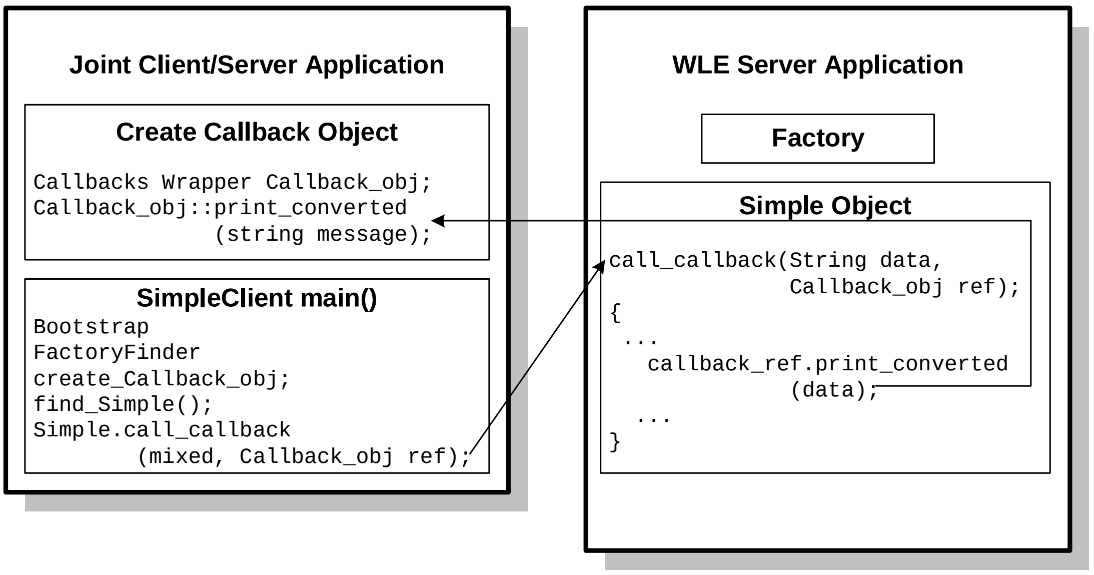
You need the Java JDK version 1.2.1 to create Java joint client/server applications.

## The Callback Sample Application

Throughout this topic, the Callback sample application is used to demonstrate the development steps. The callback object in the joint client/server application has a `print_converted` method, which accepts a string from the `Simple` object in the WLE server application and prints the string in uppercase and lowercase letters.

Figure 3-1 illustrates how the Callback sample application works.

**Figure 3-1 How the Callback Sample Application Works**



The source files for the Callback sample application are located in the `WLEdir\samples\corba\callback_java` directory of the WLE software. See "“Building and Running the Callback Sample Application” on page 3-19" for more information.

## Step 1: Writing the OMG IDL

You use OMG IDL to describe available CORBA interfaces to client applications. An interface definition written in OMG IDL completely defines the CORBA interface and fully specifies each operation’s arguments. OMG IDL is a purely declarative language. This means that it contains no implementation details. For more information about OMG IDL, see *Creating CORBA Client Applications*.

The Callback sample application implements the CORBA interfaces listed in Table 3-2.

**Table 3-2 CORBA Interfaces for the Callback Sample Application**

| Interface | Description | Operation |
|---|---|---|
| `Callback` | Accepts a string from the `Simple` object in the WLE server application and prints the string in uppercase and lowercase letters | `print_converted()` |
| `Simple` | Calls the Callback object in the joint client/server application | Calls the Callback object in the joint client/server application |
| `SimpleFactory` | Creates object references to the `Simple` object | `find_simple()` |

Listing 3-1 shows the `simple.idl` file that defines the Callback, Simple, and SimpleFactory interfaces in the Callback sample application.

**Listing 3-1 OMG IDL for the Callback Sample Application**

```
#pragma prefix "beasys.com"

interface Callback

//This method prints the passed data in uppercase and lowercase
//letters.

{
       void print_converted(in string message);
};

interface Simple

//Call the callback object in the joint client/server application

{
              void call_callback(in string val, in Callback
                                   callback_ref);
};
```

```
interface SimpleFactory
{
     Simple find_simple();
};
```

## Step 2: Generating Skeletons and Client Stubs

The interface specification defined in OMG IDL is used by the IDL compiler to generate skeletons and client stubs. Note that a joint client/server application uses the client stub for the WLE objects and the skeleton and client stub for the callback object.

For example, in the Callback sample application, the joint client/server application uses the skeleton and the client stub for the Callback object to implement the object. The joint client/server application also uses the client stubs for for the Simple and SimpleFactory to invoke operations on the objects. The WLE server application uses the skeletons for the Simple and SimpleFactory objects to implement the objects and the client stub for the Callback object to invoke operations on the object.

During the development process, you use the following compilers to build client stubs and skeletons.

- You use the `idltojava` command supplied with the JDK version 1.2.1 to compile the OMG IDL file and generate client stubs and skeletons to be used by the joint client/server application.
- You use the `m3idltojava` command to compile the OMG IDL file and generate client stubs and skeletons to be used by the WLE server application.

The names of the files generated by the `idltojava` and `m3idltojava` commands are the same; however, the content is different. When developing a WLE application that contains a joint client/server application, it is recommended that you create two separate directories for each set of client stubs and skeletons. For the Callback sample application, the files generated by the `idltojava` command are located in the `client` directory and the files generated by the `m3idltojava` command are located in the `server` directory.

Table 3-3 lists the files that are generated by the `idltojava` and the `m3idltojava` commands.

**Table 3-3 Files Created by the `idltojava` and `m3idltojava` Commands**

| File | Description |
|---|---|
| `Callback.java` | The Java version of the `Callback` OMG IDL interface. It extends `org.omg.CORBA.Object.` |
| `CallbackHelper.java` | The Java class that provides auxiliary functionality, notably the `narrow` method. |
| `CallbackHolder.java` | The Java class that provides operations for `out` and `inout` arguments that are included in CORBA, but that do not map exactly to Java. |
| `_CallbackStub.java` | The client stub that implements the `Callback.java` interface. |
| `_CallbackImplBase.java` | The skeleton that implements the `Callback.java` interface. The class `CallbackImpl` extends `_CallbackImplBase`. |
| `Simple.java` | The Java version of the `Simple` OMG IDL interface. It extends `org.omg.CORBA.Object`. |
| `SimpleHelper.java` | The Java class that provides auxiliary functionality, notably the `narrow` method. |
| `SimpleHolder.java` | The Java class that provides operations for `out` and `inout` arguments that CORBA has but that do not match exactly to Java. |
| `_SimpleStub.java` | The client stub that implements the `Simple.java` interface. |
| `_SimpleImplBase.java` | The skeleton that implements the `Simple.java` interface. The class `SimpleImpl` extends `_SimpleImplBase`. |

| File | Description |
|---|---|
| `SimpleFactory.java` | The Java version of the `SimpleFactory` OMG IDL interface. It extends `org.omg.CORBA.Object`. |
| `SimpleFactoryHelper.java` | The Java class that provides auxiliary functionality, notably the `narrow` method. |
| `SimpleFactoryHolder.java` | The Java class that provides operations for `out` and `inout` arguments that are included in CORBA, but that do not map exactly to Java. |
| `_SimpleFactoryImplBase.java` | The skeleton that implements the `SimpleFactory.java` interface. The class `SimpleFactoryImpl` extends `_SimpleFactoryImplBase`. |
| `_SimpleFactoryStub.java` | The client stub that implements the `SimpleFactory.java` interface. |

The skeleton class that is created by the `idltojava` command does not inherit from the TP Framework `com.beasys.Tobj_Servant` class. Instead, the skeleton class inherits directly from the `org.omg.CORBA.DynamicImplementation` class. Inheriting from `com.beasys.Tobj_Servant` means the joint client/server application must explicitly create a servant for the callback object and initialize the servant's state. The servant for the callback object cannot use the `activate_object` and `deactivate_object` methods as they are members of the `com.beasys.Tobj_Servant` class.

## Step 3: Writing the Methods That Implement Each Interface's Operations

After you compile the OMG IDL, you need to write methods that implement the operations of each object. In a joint client/server application, you write the implementation file for the callback object. You write the implementation file for a callback object as you would write the implementation file for any other CORBA object. You also write the implementation file for the WLE object in your WLE application.

An implementation file consists of the following:

- Method declarations for each operation specified in the OMG IDL file
- Your application's business logic
- Constructors for each interface implementation (optional)

Listing 3-2 includes the implementation file for the Callback object.

**Listing 3-2 Implementation File for the Callback Object**

```
//The implementation file for the Callback object. The Callback
//object implements the print_converted method.

class CallbackImpl extends _CallbackImplBase {

//Prints a string in upper and lower case

       public void print_converted(String data) {
         if (data == null)
              System.out.println("Null String");
       else
       {
              //Print input data in uppercase
              System.out.println(data.toUpperCase());

              //Print input data in lowercase
              System.out.println(data.toLowerCase());
       }
   }
}
```

Listing 3-3 includes the implementation file for the Simple object.

**Listing 3-3 Implementation File for the Simple Object**

```
import com.beasys.Tobj.TP;

//The implementation file for the Simple interface. The Simple
```

```
//interface implements the call_callback method of the Callback
//object.

public class SimpleImpl extends _SimpleImplBase
{
       public void call_callback(String data, Callback
                                 callback_ref)

//Call the print_converted method on the reference to the Callback
//object
       {
         callback_ref.print_converted(data);
         return;
       }
}
```

Listing 3-4 includes the implementation file for the SimpleFactory object.

**Listing 3-4 Implementation File for the SimpleFactory Object**

```
import com.beasys.Tobj.TP;

//The implementation file for the SimpleFactory object. The
//SimpleFactory object provides methods to create a Simple object.


public class SimpleFactoryImpl extends _SimpleFactoryImplBase
{

//Create an object reference to a Simple object.

       public Simple find_simple()
         {
              try {
                     org.omg.CORBA.Object simple_oref =
                         TP.create_object_reference(
                            SimpleHelper.id(),  // Repository id
                            "simple_callback",  // object id
                            null               // routing criteria
                            );

       // Send back the narrowed reference.

                     return SimpleHelper.narrow(simple_oref);
```

```
          } catch (Exception e){
             TP.userlog("Cannot create Simple: " +e.getMessage());
             e.printStackTrace();
             return null;
          }
    }
}
```

## Step 4: Initializing the ORB

In previous versions of the WLE product, Java client applications used the JDK ORB without modifications. Version 4.2 of the WLE product provides a value-added implementation of the JDK ORB. The modifications to the JDK ORB include classes and methods that support callback objects. The classes and methods for the callback objects are in the `wleclient.jar` file located in the following directories:

**UNIX**

*`$wledir`*`/udataobj/java/jdk`

**Window NT**

*`%wledir%`*`\udataobj\java\jdk`

To use this modified JDK ORB, Java joint client/server applications must set certain properties. Listing 3-5 contains the command to initialize the JDK ORB with the correct properties. For more information about the properties used to initialize the JDK ORB, see the *CORBA Java Programming Reference*.

**Listing 3-5 Initializing the ORB in the Callback Sample Application**

```
properties prop = new Properties(System.getProperties());
prop.put(“org.omg.CORBA.ORBclass”,
    “com.beasys.CORBA.iiop.ORB”);
prop.put(“org.omg.CORBA.ORBSingletonclass”,
    “com.beasys.CORBA.idl.ORBSingleton”);
System.setProperties(prop);
```

```
//Initialize the ORB

ORB orb = ORB.init(args, prop);
```

## Step 5: Writing the Client Portion of the Joint Client/Server Application

During development of a joint client/server application, you write the client portion of the joint client/server application as you would write any WLE client application. The client application needs to include code that does the following:

1. Uses the Bootstrap object to establish communication with the WLE domain
2. Resolves initial references to the FactoryFinder object
3. Uses a factory to get an object reference for the desired WLE object

The client development steps are illustrated in Listing 3-6, which includes code from the Callback sample application. In the Callback sample application, the client portion of the joint client/server application uses a factory to get an object reference to the Simple object.

**Listing 3-6 The Client Portion of the Callback Sample Application**

```
//Create a Bootstrap object
       Tobj_Bootstrap bootstrap = new Tobj_Bootstrap(orb,"");

//Create the Bootstrap object. The TOBJADDR system property
//defines the host and port.

Tobj_Bootstrap bootstrap = new Tobj_Bootstrap(orb, "");

//Use the Bootstrap object to find the FactoryFinder object.

org.omg.CORBA.Object fact_finder_oref =
       bootstrap.resolve_initial_references("FactoryFinder");

//Narrow the FactoryFinder object.
```

```
FactoryFinder fact_finder_oref =
       FactoryFinderHelper.narrow(fact_finder_oref);

//Use the FactoryFinder object to locate a factory for the
//Simple object.

org.omg.CORBA.Object simple_fact_oref =
     fact_finder_oref.find_one_factory_by_id
     (SimpleFactoryHelper.id());

//Narrow the factory.

SimpleFactory simple_factory_oref =
     SimpleFactoryHelper.narrow(simple_fact_oref);

//Find the Simple object.

Simple simple = simple_factory_oref.find_simple();
```

## Step 6: Creating a Callback Object Using the Callbacks Wrapper Object

To allow the use of outbound IIOP in Java joint client/server applications, the JDK ORB has been extended to implement certain POA functionality. The POA functionality is implemented through the Callbacks Wrapper object.

The Callbacks Wrapper object does the following:

- Defines the object policy for the callback object. The following object policies are supported:
  - Transient/SystemID (`_transient`)
  - Persistent/SystemID (`_persistent/systemid`)
  - Persistent/UserID (`_persistent/userid`)

  For a complete description of the object policies for callback objects, see "Object Policies for Callback Objects" on page 1-4.

- Creates a servant for the callback object.

- Sets the ORB to the state in which it will accept requests on the callback object.
- Returns an object reference to the activated callback object. The object Id can be generated by the system or supplied by the user.
- Tells the ORB to stop accepting requests on either a single servant or all the active servants.

For a complete description of the Callbacks Wrapper object, see the *CORBA Java Programming Reference*.

Listing 3-7 shows how the Callbacks object is used in the Callback sample application.

**Listing 3-7 Using the Callbacks Wrapper Object in the Callback Sample Application**

```
import java.io.*;
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CORBA.portable.ObjectImpl;
import com.beasys.*;
import com.beasys.Tobj.*;
import com.beasys.BEAWrapper.Callbacks;
...
//Create the servant for the Callback object

CallbackImpl callback_ref = new CallbackImpl();

//Use the Callbacks Wrapper object to create the callback object

Callbacks callbacks = new Callbacks(orb);

//Activate the servant and allow the ORB to accept
//callback requests.

callbacks.start_persistent_userid(callback_ref,
       ((ObjectImpl)callback_ref)._ids() [0],
       “myID”);
...
```

## Step 7: Establishing a Connection to an ISH

To support IIOP more efficiently in Java joint client/server applications, the Bootstrap object supports a `register_callback_port` method. This method registers the callback object in a joint client/server application with the listening port of an ISH, causing invocations to the callback object to be routed through the specified ISH.

In this situation, the joint client/server application is using dual-pair connection IIOP. A joint client/server application that does not perform this registration will force server applications that invoke the callback object in the joint client/server application to use asymmetric IIOP, which uses the ORB infrastructure to locate an available ISH.

**Note:** The callback object must be activated before the `register_callback_port` method is called.

Listing 3-8 shows how the `register_callback_port` method is used in the Callback sample application.

**Listing 3-8 The register_callback_port Method in the Callback Sample Application**

```
...
//Register the callback port are specified in org.omg.CORBA.ORBport

bootstrap.register_callback_port(callback_ref);
...
```

## Step 8: Invoking Operations on the Callback Object

Once you have an object reference to a callback object, you pass the callback object reference as a parameter to a method of a WLE object. In the Callback sample application, the Simple object (the WLE object) uses an object reference to the Callback object as a parameter to the `call_callback` method. Listing 3-9 illustrates this step.

**Listing 3-9 Invoking the call_callback Method**

```
...
//Call the call_callback method which invokes the Callback object

simple.call_callback(mixed, callback_ref);
...
```

## Step 9: Specifying Configuration Information

When using joint client/server applications, the object references for the callback object must contain a host and port number, as follows:

- For transient callback objects, any port is sufficient and can be obtained dynamically by the ORB.
- For persistent callback objects, the ORB must be configured to accept requests for the callback object on the same port on which the object reference for the callback object was created.

The ORB is configured by setting the `org.omg.CORBA.ORBPort` system property. Every time you run the joint client/server application, you must enter the following commands to set the `org.omg.CORBA.ORBPort` system property:

**UNIX**

```
java -DTOBJADDR=//Host:Port
     -Dorg.omg.CORBA.ORBport=portnumber
     -classpath=$CLASSPATH JointClientServerApplication
```

**Window NT**

```
java -DTOBJADDR=//Host:Port
     -Dorg.omg.CORBA.ORBport=portnumber
     -classpath=%CLASSPATH% JointClientServerApplication
```

The administrator assigns the port number for the joint client/server application from the user range of port numbers, rather than from the dynamic range. Assigning port numbers from the user range prevents joint client/server applications from using conflicting ports.

For Java joint client/server applications, the administrator needs to boot the IIOP Server Listener (ISL) with startup parameters that enable outbound IIOP to invoke callback objects not connected to an IIOP Server Handler (ISH). The `-O` option of the ISL command enables outbound IIOP. The ISL parameter is defined in the configuration file. Additional parameters allow administrators to obtain the optimum configuration for their WLE application. For more information about the ISL command, see the *Reference*.

**Note:** The Callback sample application does not demonstrate using asymmetric IIOP. Therefore, the `-O` option is not used in the configuration file.

## Step 10: Compiling Java Joint Client/Server Applications

When creating joint client/server applications, use the `javac` command provided with the JDK 1.2.1 to construct an executable for the joint client/server application. The command compiles the java source code of the joint client/server application.

When compiling joint client/server applications, you need to include the following Java ARchive (JAR) files in your `CLASSPATH`:

- The `m3envobj.jar` file, which contains Java versions of the WLE environmental objects

**Note:** The `m3envobj.jar` file is in this directory: `wledir\udataobj\java\jdk`.

- The `wleclient.jar` file, which contains the classes and methods for the Callbacks Wrapper object

For the syntax of the `javac` command, see the *CORBA Java Programming Reference*.

You use the `buildjavaserver` command to build the WLE server application that invokes the callback object. For information about compiling WLE server applications, see *Getting Started* and *Creating CORBA Java Server Applications*.

## Threading Considerations for Java Joint Client/Server Applications

**Note:** The Callback sample application does not use multiple threads.

Since Java as an execution environment is multithreaded, there is no need to implement the ORB `org.omg.CORBA.orb.work_pending` and `org.omg.CORBA.orb.perform_work` methods. These methods throw a `NO_IMPLEMENT` exception when a user tries to invoke them. In addition, the `org.omg.CORBA.orb.run` method does not need to be called. Be aware that any code that executes concurrently must be written to be thread-safe.

When using multiple threads in Java, the client functionality of the joint client/server application starts up in the main thread. The joint client/server application then activates the callback object using one of the start methods of the Callbacks Wrapper object. The Callbacks Wrapper object registers the servant for the callback object, and its associated object ID, in the ORB’s object manager. The joint client/server application is then free to pass the object reference for the callback object to any application that may need to invoke the callback object.

**Note:** The BEA version of the JDK ORB requires an explicit call to one of the start methods of the Callbacks Wrapper object to initialize the servant for the callback object and create a valid object ID. This requirement differs from the base JDK ORB, which allows implicit creation of object references through the `orb.connect` method when marshaling an object reference when an application has not already done so.

Invocations on the callback object are handled by the ORB. As each request is received, the ORB validates the request against the object manager and spawns a thread for that request. Multiple requests can be made simultaneously to the same callback object, since the ORB creates a new thread for each request.

As each request terminates, the thread that runs the servant for the callback object terminates. The main thread that controls the client functionality of the joint client/server application can make as many client invocations as it needs. There is no restriction to prevent other servants defined in the joint client/server application to act as client applications and invoke on WLE objects. A call to `stop_all_objects()` merely takes the callback object out of the object manager's list, thus preventing any further invocations on the callback object. Any invocation to a stopped callback object fails as if it never existed.

If the client functionality of a joint client/server application needs to retrieve the results of a callback from another thread, the client functionality must use normal thread synchronization techniques.

If any thread in the joint client/server application invokes an `exit` method, all activity is stopped and the Java execution environment terminates. It is recommended to only call `return()` to terminate a thread.

## Building and Running the Callback Sample Application

Perform the following steps to build and run the Callback sample application:

1. Copy the files for the Callback sample application into a work directory.
2. Change the protection attribute on the files for the Callback sample application.

3. Verify the environment variables.
4. Execute the `runme` command.

The following sections describe these steps.

## Copying the Files for the Callback Sample Application into a Work Directory

You need to copy the files for the Callback sample application into a work directory on your local machine. The files for the Callback sample application are located in the following directories:

**Windows NT**

*`drive:`*`\`*`WLEdir`*`\samples\corba\callback_java`

**UNIX**

*`/usr/local/WLEdir/`*`samples/corba/callback_java`

You will use the files listed in Table 3-4 to build and run the Callback sample application.

**Table 3-4 Files Included in the Callback Sample Application**

| File | Description |
|---|---|
| `Simple.idl` | The OMG IDL code that declares the `Callback`, `Simple`, and `SimpleFactory` interfaces.This file is copied from the sample application directory by the `runme` command file. |
| `ServerImpl.java` | The Java source code that implements the `Server.initialize` and `Server.release` methods. |
| `SimpleJCS.java` | The Java source code for the joint client/server application in the Callback sample application. |
| `SimpleFactoryImpl.java` | The Java source code that implements the methods of the `SimpleFactory` object . |

**Table 3-4 Files Included in the Callback Sample Application**

| File | Description |
|---|---|
| `SimpleImpl.java` | The Java source code that implements the methods of the `Simple` object. |
| `CallbackImpl.java` | The Java source code that implements the methods of the `Callback` object. |
| `Simple.xml` | The Server Description File used to associate activation and transaction policy values with CORBA interfaces. For the Callback sample application, the `Simple` and `SimpleFactory` interfaces have an activation policy of `method` and a transaction policy of `never`. |
| `Readme.txt` | The file that provides the latest information about building and running the Callback sample application. |
| `runme.cmd` | The Windows NT batch file that builds and runs the Callback sample application. |
| `runme.ksh` | The UNIX Korn shell script that builds and executes the Callback sample application. |
| `makefile.mk` | The UNIX Korn `make` file for the Callback sample application. This file is used to manually build the Callback sample application. Refer to the `Readme.txt` file for information about manually building the Callback sample application. The UNIX `make` command needs to be in the path of your machine. |
| `makefile.nt` | The Windows NT `make` file for the Callback sample application. This `make` file can be used directly by the Visual C++ `nmake` command. This file is used to manually build the Callback sample application. Refer to the `Readme.txt` file for information about manually building the Callback sample application. The Windows NT `nmake` command needs to be in the path of your machine. |

**Table 3-4 Files Included in the Callback Sample Application**

| File | Description |
|---|---|
| `smakefile.nt` | The `make` file that is used with the Visual Cafe `smake` command for the Callback sample application.<br>**Note:** `makefile.nt` is included by `smakefile.nt`. |

**Note:** When running the Callback sample application on the UNIX operating system, you need to make sure the `makefile` is in the path of your machine.

## Changing the Protection Attribute on the Files for the Callback Sample Application

During the installation of the WLE software, the sample application files are marked read-only. Before you can edit or build the files in the Callback sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

**Windows NT**

```
prompt>attrib -r drive:\workdirectory\*.*
```

**UNIX**

```
prompt>/bin/ksh
```

```
ksh prompt>chmod u+w /workdirectory/*.*
```

On the UNIX operating system platform, you also need to change the permission of `runme.ksh` to give execute permission to the file, as follows:

```
ksh prompt>chmod +x runme.ksh
```

## Verifying the Settings of the Environment Variables

Before building and running the Callback sample application, you need to ensure that certain environment variables are set on your system. In most cases, these environment variables are set as part of the installation procedure. However, you need to check the environment variables to ensure they reflect correct information.

Table 3-5 lists the environment variables required to run the Callback sample application.

**Table 3-5 Required Environment Variables for the Callback Sample Application**

| Environment Variable | Description |
|---|---|
| `TUXDIR` | The directory path where you installed the WLE software. For example:<br>**Windows NT**<br>`TUXDIR=c:\WLEDir`<br>**UNIX**<br>`TUXDIR=/usr/local/WLEDir` |
| `JAVA_HOME` | The directory path where you installed the JDK software. For example:<br>**Windows NT**<br>`JAVA_HOME=c:\JDK1.2`<br>**UNIX**<br>`JAVA_HOME=/usr/local/JDK1.2` |

To verify that the information for the environment variables defined during installation is correct, perform the following steps:

**Windows NT**

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.

   The Control Panel appears.
3. Click the System icon.

   The System Properties window appears.

4. Click the Environment tab.

   The Environment page appears.

5. Check the settings for `TUXDIR` and `JAVA_HOME`.

**UNIX**

```
ksh prompt>printenv TUXDIR
ksh prompt>printenv JAVA_HOME
```

To change the settings, perform the following steps:

**Windows NT**

1. On the Environment page in the System Properties window, click the environment variable you want to change, or enter the name of the environment variable in the Variable field.
2. Enter the correct information for the environment variable in the Value field.
3. Click OK to save the changes.

**UNIX**

```
ksh prompt>export TUXDIR=directorypath
ksh prompt>export JAVA_HOME=directorypath
```

Table 3-6 lists additional environment variables that may be set prior to running the Callback sample application.

**Table 3-6 Optional Environment Variables for the Callback Sample Application**

| Environment Variable | Description |
|---|---|
| `HOST` | The host name portion of the TCP/IP network address used by the ISL process to accept connections from the ORB. The default value is the name of the local machine. |
| `PORT` | The TCP port number at which the ISL process listens for incoming requests; it must be a number between 0 and 65535. The default is 2468. |

| **Environment Variable** | **Description** |
|---|---|
| `IPCKEY` | The address of shared memory; it must be a number greater than 32769 unique to this application on this system. The default value is 55532. |
| `CALLBACK_PORT` | The TCP port number at which the client application process listens for incoming callback requests; it must be a number between 0 and 65535. The default value is 2458. |

## Executing the runme Command

The `runme` command automates the following steps:

1. Setting the system environment variables
2. Loading the `UBBCONFIG` file
3. Compiling the code for the client application
4. Compiling the code for the server application
5. Starting the server application using the `tmboot` command
6. Starting the client application
7. Stopping the server application using the `tmshutdown` command

**Note:** You can also run the Callback sample application manually. The steps for manually running the Callback sample application are described in the `Readme.txt` file.

To build and run the Callback sample application, enter the `runme` command, as follows:

**Windows NT**

```
prompt>cd workdirectory

prompt>runme
```

**UNIX**

```
ksh prompt>cd workdirectory
```

```
ksh prompt>./runme.ksh
```

The Callback sample application runs and prints the following messages:

```
Testing simpapp
    cleaned up
    prepared
    built
    loaded ubb
    booted
    ran
    shutdown
    saved results
  PASSED
```

**Note:** After executing the `runme` command, you may get a message indicating that the `Host`, `Port`, and `IPCKEY` parameters in the `UBBCONFIG` file conflict with an existing `UBBCONFIG` file. If this occurs, you need to set these parameters to different values to get the Callback sample application running on your machine.

The `runme` command starts the following application processes:

- `TMSYSEVT`

  The BEA TUXEDO system event broker.

- `TMFFNAME`

  The following three `TMFFNAME` server processes are started:

  - The `TMFFNAME` server process started with the `-N` and `-M` options is the Master NameManager service. The NameManager service maintains a mapping of the application-supplied names to object references.
  - The `TMFFNAME` server process started with only the `-N` option is the Slave NameManager service.
  - The `TMFFNAME` server process started with the `-F` option contains the FactoryFinder object.

- `JavaServer`

The server application server process that implements the `SimpleFactory` and `Simple` interfaces. The JavaServer process has one option, `simple.jar`, which is the Java ARchive (JAR) file that was created for the application.

- `ISL`

  The IIOP Listener server process.

Table 3-7 lists the files in the work directory generated by the `runme` command.

**Table 3-7 Files Generated by the `runme` Command**

| File | Description |
|---|---|
| `SimpleFactory.java,` `SimpleFactoryHolder.java,` `SimpleFactoryHelper.java,` `_SimpleFactoryStub.java,` `_SimpleFactoryImplBase.java,` `Simple.java` `SimpleHolder.java,` `SimpleHelper.java,` `_SimpleStub.java,` `_SimpleImplBase.java,` `Callback.java,` `CallbackHolder.java` `CallbackHelper.java` `_CallbackStub.java` `_CallbackImplBase.java` | Client stubs, skeletons, and Java Helper and Holder classes for the `SimpleFactory, Simple, and Callback` interfaces. For a description of the files, see Table 3-3. |
| `Simple.ser` | The Server Descriptor File. |
| `Simple.jar` | The server JAR file. |
| `SimpleJCS.jar` | The JAR file for the joint client/server application. |
| `.adm/.keybd` | A file that contains the security encryption key database. |
| `results` | A generated directory. |

Table 3-8 lists files in the `results` directory generated by the `runme` command.

**Table 3-8 Files in the `results` Directory Generated by the runme Command**

| File | Description |
|---|---|
| `input` | Contains the input that the `runme` command provides to the Java client application. |
| `output` | Contains the output produced when the `runme` command executes the Java client application. |
| `expected_output` | Contains the output that is expected when the Java client application is executed by the `runme` command. The data in the `output` file is compared to the data in the `expected_output` file to determine whether or not the test passed or failed. |
| `log` | Contains the output generated by the `runme` command. If the `runme` command fails, check this file for errors. |
| `setenv.cmd` | Contains the commands to set the environment variables needed to build and run the Callback sample application on the Windows NT operating system platform. |
| `setenv.ksh` | Contains the commands to set the environment variables needed to build and run the Callback sample application on the UNIX operating system platform. |
| `stderr` | Generated by the `tmboot` command, which is executed by the `runme` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.err.println` method sends the output to the `stderr` file instead of to the `ULOG` file. |

**Table 3-8 Files in the `results` Directory Generated by the runme Command**

| File | Description |
|---|---|
| `stdout` | Generated by the `tmboot` command, which is executed by the `runme` command. If the `-noredirect` JavaServer option is specified in the `UBBCONFIG` file, the `System.out.println` method sends the output to the `stdout` file instead of to the `ULOG` file. |
| `tmsysevt.dat` | Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the `tmboot` command in the `runme` command. |
| `tuxconfig` | A binary version of the `UBBCONFIG` file. |
| `ubb` | The `UBBCONFIG` file for the Callback sample application. |
| `ULOG.<date>` | A log file that contains messages generated by the `tmboot` command. |

## Using the Callback Sample Application

This section describes how to use the Callback sample application after the `runme` command is executed.

Run the joint client/server application in the Callback sample application, as follows:

**Windows NT**

```
prompt>tmboot -y
prompt>java -classpath %CLIENTCLASSPATH% -DTOBJADDR=%TOBJADDR
-Dorg.omg.CORBA.ORBPort=%CALLBACK_PORT% SimpleJCS
String?
Hello World
HELLO WORLD
hello world
```

**UNIX**

```
ksh prompt>tmboot
ksh prompt>java -classpath $CLIENTCLASSPATH -DTOBJADDR=$TOBJADDR
-Dorg.omg.CORBA.ORBPort=$CALLBACK_PORT SimpleJCS
String?
Hello World
HELLO WORLD
hello world
```

Before using another sample application, enter the following commands to stop the Callback sample application and to remove unnecessary files from the work directory:

**Windows NT**

```
prompt>tmshutdown -y
```

```
prompt>nmake -f makefile.nt clean
```

**UNIX**

```
ksh prompt>tmshutdown -y
```

```
ksh prompt>make -f makefile.mk clean
```

# Index