Enterprise JavaBeans Components and CORBA Clients: A Developer Guide

Abstract

This paper discusses how to enable a client written in any language supported by CORBA to access Enterprise JavaBeans™ components (“EJB™ components”). This paper is directed at programmers with advanced knowledge of both the Java™ 2 Platform, Enterprise Edition (J2EE™) and CORBA (Common Object Request Broker Architecture).

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

January 2002

Enterprise JavaBeans™ Components and CORBA Clients: A Developer Guide

This paper discusses how to enable a client written in any language supported by CORBA to access Enterprise JavaBeans™ components ("EJB™ components"). This paper is directed at programmers with advanced knowledge of both the Java™ 2 Platform, Enterprise Edition (J2EE™) and CORBA (Common Object Request Broker Architecture).

J2EE technology simplifies enterprise applications by basing them on standardized, modular and re-usable components based on the Enterprise JavaBeans™ (EJB™) architecture, providing a complete set of services to those components, and handling many details of application behavior automatically. By automating many of the time-consuming and difficult tasks of application development, J2EE technology allows enterprise developers to focus on adding value, that is, enhancing business logic, rather than building infrastructure.

The **EJB™ server-side component model** simplifies development of middleware components that are transactional, scalable, and portable. Enterprise JavaBeans servers reduce the complexity of developing middleware by providing automatic support for middleware services such as transactions, security, database connectivity, and more.

CORBA is an Object Management Group (OMG) standard that is an open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard Internet Inter-ORB Protocol (IIOP), a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network. To learn more about CORBA, visit http://www.omg.org/gettingstarted/gettingstartedindex.htm.

CORBA technology complements the Java platform by providing a distributed objects framework, services to support that framework, and interoperability with other languages. CORBA technology is an integral part of the Java 2 platform, being used in Enterprise JavaBeans components, Java Remote Method Invocation APIs running over Internet Inter-ORB Protocol ("Java RMI-IIOP"), and Java IDL APIs ("Java IDL").

OMG Interface Definition Language (IDL) is used to describe the interfaces being implemented by the remote objects. IDL is used to define the name of the interface and the names of each of the attributes and methods. Once you create the IDL file, you can use an IDL compiler to generate the client stub and the server skeleton in any language for which the OMG has defined a specification for such language mapping. To learn more about OMG IDL, visit http://www.omg.org/gettingstarted/omg_idl.htm.

Java IDL makes it possible for distributed Java applications to transparently invoke operations on remote network services using the industry standard OMG IDL and IIOP defined by the Object Management Group (http://www.omg.org). Java RMI over IIOP APIs enable the programming of CORBA servers and applications via the `javax.rmi` API.

Developers who program EJB components follow the Java RMI programming model for their distributed object model, where the required transport common across all application servers is Java RMI-IIOP. In heterogeneous server environments, the standard mapping of the EJB architecture to CORBA enables the following interoperability:

- A client using an ORB from one vendor can access enterprise beans residing on a server enabled with Enterprise JavaBeans technology ("EJB server") provided by another vendor.
- Enterprise beans in one EJB server can access enterprise beans in another EJB server.
- A CORBA client written in a language other than the Java programming language can access any EJB component as long as there is a mapping from OMG IDL to that programming language.

The rest of this document provides an example of a CORBA client application accessing an enterprise bean object. In this document, a CORBA client means a client application written in any language supported by CORBA, including the Java programming language, C++, C, Smalltalk, COBOL, Ada, Lisp, or Python. While the Java code in this example is specific to enterprise beans, the process for developing a CORBA client that accesses a server created using the Java RMI-IIOP APIs is the same.

Links to similar example applications from other vendors who implement J2EE technology can be found in "Links to similar examples" on page 20.

Developing a CORBA Client that Accesses an Enterprise Bean

This is an example of how to develop a CORBA client application that accesses an EJB component. In this example, the client is written in the C++ programming language, but the client could be written in any language supported by CORBA.

The general process for developing a CORBA client so that it can access an enterprise bean is demonstrated in the following sections:

1. “Write the Enterprise Bean”, on page 5.
2. “Generate the CORBA IDL”, on page 9.
3. “Create a CORBA client”, on page 10.
4. “Deploy the Enterprise Bean”, on page 14.
5. “Run the client executable”, on page 15.

This document also includes:

- “Creating a Java RMI-IIOP client application”, on page 16.
- “Where to go from here”, on page 19.
- “Tips for complex interfaces”, on page 19.
- “Links to similar examples”, on page 20.

In order to make the example simple, we have taken a few shortcuts. For information on building more advanced solutions, see “Tips for complex interfaces” on page 19.

▼ Write the Enterprise Bean

The following examples show the code for an enterprise bean that will accept simple `String` log messages sent to the application server from Java RMI-IIOP and CORBA clients. The enterprise bean prints them on the server along with the current server time.

1. **Create the following files:** `Logger.java`**,** `LoggerHome.java`**,** `LoggerEJB.java`**, and** `LogMessage.java` **in the** `/Java/src/ejbinterop` **directory.**

Logger.java

The file `Logger.java` is the enterprise bean's remote interface, and as such, it extends `EJBObject`. A remote interface provides the remote client view of an EJB object and defines the business methods callable by a remote client.

CODE EXAMPLE 1 Logger.java

```
package ejbinterop;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

/**
 * Accepts simple String log messages and prints
 * them on the server.
 */
public interface Logger extends EJBObject
{
    /**
     * Logs the given message on the server with
     * the current server time.
     */
    void logString(String message) throws RemoteException;
}
```

LoggerHome.java

The file `LoggerHome.java` extends `EJBHome`. The `EJBHome` interface must be extended by all EJB component's remote home interfaces. A home interface defines the methods that allow a remote client to create, find, and remove EJB objects, as well as home business methods that are not specific to an EJB instance.

CODE EXAMPLE 2 LoggerHome.java

```
package ejbinterop;

import java.rmi.RemoteException;
import javax.ejb.EJBHome;
import javax.ejb.CreateException;

public interface LoggerHome extends EJBHome
{
    Logger create() throws RemoteException, CreateException;
}
```

LoggerEJB.java

The file `LoggerEJB.java` contains the code for a session bean. A session bean is an enterprise bean that is created by a client and that usually exists only for the duration of a single client-server session. A session bean performs operations such as calculations or accessing a database for the client. In this example, the enterprise bean accepts simple `String` log messages from the client and prints them on the server.

CODE EXAMPLE 3 LoggerEJB.java

```
package ejbinterop;

import javax.ejb.*;
import java.util.*;
import java.rmi.*;
import java.io.*;

/**
 * Accepts simple String log messages and prints
 * them on the server.
 */
public class LoggerEJB implements SessionBean {

    public LoggerEJB() {}
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sc) {}

    /**
     * Logs the given message on the server with
     * the current server time.
     */
    public void logString(String message) {
        LogMessage msg = new LogMessage(message);

        System.out.println(msg);
    }
}
```

LogMessage.java

The file `LogMessage.java` takes the current date and time, creates a formatted `String` showing the message, and prints the message to the server.

CODE EXAMPLE 4 LogMessage.java

```
package ejbinterop;

import java.io.Serializable;
import java.util.Date;
import java.text.*;

/**
 * Simple message class that handles pretty
 * printing of log messages.
 */
public class LogMessage implements Serializable
{
    private String message;
    private long datetime;

    /**
     * Constructor taking the message. This will
     * take the current date and time.
     */
    public LogMessage(String msg) {
        message = msg;
        datetime = (new Date()).getTime();
    }

    /**
     * Creates a formatted String showing the message.
     */
    public String toString() {
        StringBuffer sbuf = new StringBuffer();
        DateFormat dformat
            = DateFormat.getDateTimeInstance(DateFormat.MEDIUM,
                  DateFormat.LONG);
        FieldPosition fpos = new
            FieldPosition(DateFormat.DATE_FIELD);
        dformat.format(new Date(datetime), sbuf, fpos);
        sbuf.append(": ");
        sbuf.append(message);
        return sbuf.toString();
    }
}
```

2. **To compile the files written in this section:**

```
javac -classpath $J2EE_HOME/lib/j2ee.jar:<ejbinterop_directory>
  *.java
```

These commands create `class` files for all of the `.java` files in the current directory. This command and others in this paper assume that the *J2EE_HOME* environment variable has been set correctly. Using *$J2EE_HOME* is a convention of the Unix® operating environment. Substitute *%J2EE_HOME%* when working in the Microsoft Windows operating environment.

▼ Generate the CORBA IDL

This sections discusses generating the Interface Definition Language (IDL) files from the Java class files generated in the previous section. In this example, we will use the `rmic` compiler to map the Java code to IDL. IDL provides a purely declarative, programming language-independent way of specifying an object's API.

3. **Run the `rmic` compiler against the Java class files generated in the previous step as follows:**

```
rmic -idl -noValueMethods -classpath
  $J2EE_HOME/lib/j2ee.jar:<path_to_ejbinterop_dir>
  -d <path_to_where_idl_files_should_be_generated>
  ejbinterop.Logger ejbinterop.LoggerHome
```

In the preceding example, we are including the .jar file containing definitions for the `javax.ejb` package as well as the directory to our `ejbinterop` files. If you're using the Java™ 2 Platform, Enterprise Edition (J2EE™), version 1.3 Reference Implementation (RI), the `.jar` files are located in *$J2EE_HOME*`/lib/j2ee.jar`.

In the command line for `rmic` above, we recommend a shortcut — using the `noValueMethods` option. This option tells `rmic` to skip any methods with parameter or return types that would be mapped to CORBA value types. The advantage is that it will prevent us from generating a lot of unnecessary IDL that we might have to implement in the C++ client. The disadvantage is that we can only use primitive data types, arrays, and `Strings` as parameters or return values, and not our own Java class types. Read more about this in "Tips for complex interfaces", on page 19.

Running the `rmic` compiler on the Java class files generates the following files to the directory indicated with the *-d* option in the `rmic` statement above:

- `java/lang/Ex.idl`
- `java/lang/Exception.idl`
- `java/lang/Object.idl`
- `java/lang/Throwable.idl`
- `java/lang/ThrowableEx.idl`
- `javax/ejb/CreateEx.idl`
- `javax/ejb/CreateException.idl`
- `javax/ejb/EJBHome.idl`
- `javax/ejb/EJBMetaData.idl`

- javax/ejb/EJBObject.idl
- javax/ejb/Handle.idl
- javax/ejb/HomeHandle.idl
- javax/ejb/RemoveEx.idl
- javax/ejb/RemoveException.idl
- ejbinterop/Logger.idl
- ejbinterop/LoggerHome.idl

Note – A number of these generated files contain API that can only be used within a Java programming environment. For example, the `EJBMetaData` implementation is currently specific to each application server, and thus it will be difficult to develop equivalents that will continue to work over time on platforms other than the Java platform. One option is to remove these from the IDL, but if you do, you'll have to remove them from the IDL every time you change the Java interface and regenerate the IDL files from the `rmic` compiler.

Note – Since CORBA exceptions don't support inheritance, the Java language to IDL mapping creates an `Ex` class that contains a CORBA value type representing the actual Java exception. In this basic example, we're not going to worry much about exception support. More information about exceptions can be found at `http://java.sun.com/j2se/1.4/docs/guide/idl/jidlExceptions.html`.

---

4. **Compile the IDL files with your C++ vendor's "IDL to C++" compiler to generate the C++ code corresponding to the IDL. The steps for this procedure vary by vendor, so consult your product documentation for the specific steps for your vendor.**

▼ Create a CORBA client

The client application can be written in any language supported by CORBA. The following example provides the code for a simple C++ client that, given an Object Request Broker (ORB) and a `corbaname` URL for a `LoggerHome` object, logs a simple `String` message on the server. You'll have to adjust the `include` statements and modify the code for registering the value factories based on your C++ ORB vendor's libraries. This example was written for ORBacus for C++ 4.0.5 and some of the C++ code in this example is specific to that product.

A `corbaname` URL is a human-readable URL format that enables you to access CORBA objects. It is used to resolve a stringified name from a specific naming context. This is a new feature in the J2EE v 1.3 platform as part of the CORBA Interoperable Naming Service (INS). INS is an extension to CORBA Object Services

(COS) Naming Service, which was delivered in previous releases of the J2EE platform. To read more about INS, visit http://java.sun.com/j2se/1.4/docs/guide/idl/jidlNaming.html#INS.

In this example, the client code does the following:

1. Creates an Object Request Broker (ORB). The ORB connects objects requesting services to the objects providing them.
2. Registers value factories.
3. Looks up the `LoggerHome` object in the naming context pointed to by the `corbaname` URL.
4. Performs a safe downcast from the object returned to a `LoggerHome` object.
5. Creates a `LoggerEJB` object reference.
6. Logs our message.
7. Tells the application server we won't use this EJB reference again.

5. **Create the client using C++ code similar to the following. The exact code may vary with your C++ implementation. This code was written for ORBacus for C++ 4.0.5 and some of the C++ code in this example may be specific to that product.**

CODE EXAMPLE 5 Client.cpp

```
#include <fstream.h>

// C++ ORB Vendor specific include files
// These are from C++ ORBacus 4.0.5

#include <OB/CORBA.h>
#include <OB/OBORB.h>

// Include files generated from our IDL
#include <java/lang/Exception.h>
#include <java/lang/Throwable.h>
#include <javax/ejb/CreateException.h>
#include <javax/ejb/RemoveException.h>
#include <ejbinterop/Logger.h>
#include <ejbinterop/LoggerHome.h>

/**
 * Given an ORB and a corbaname URL for a LoggerHome
 * object, logs a simple string message on the server.
 */
void
run(CORBA::ORB_ptr orb, const char* logger_home_url)
{
  cout << "Looking for: " << logger_home_url << endl;
```

```
  // Look up the LoggerHome object in the naming context
  // pointed to by the corbaname URL
  CORBA::Object_var home_obj
    = orb->string_to_object(logger_home_url);

  // Perform a safe downcast
  ejbinterop::LoggerHome_var home
    = ejbinterop::LoggerHome::_narrow(home_obj.in());

  assert(!CORBA::is_nil(home));

  // Create a Logger EJB reference
  ejbinterop::Logger_var logger = home->create();

  CORBA::WStringValue_var msg =
    new CORBA::WStringValue((const CORBA::WChar*)L"Message
      from a C++ client");

  cout << "Logging..." << endl;

  // Log our message
  logger->logString(msg);

  // Tell the application server we won't use this
  // EJB reference any more
  logger->remove();

  cout << "Done" << endl;
}

/**
 * Simple main method that checks arguments, creates an
 * ORB, and handles exceptions.
 */
int
main(int argc, char* argv[])
{
  int exit_code = 0;
  CORBA::ORB_var orb;

  try {

  // Check the arguments
  if (argc != 2) {
    cerr << "Usage: Client <corbaname URL of LoggerHome>" << endl;
    return 1;
  }
```

```
    // Create an ORB
    orb = CORBA::ORB_init(argc, argv);

    // Register value factories

    // NOTE: This is overkill for the example since we'll never
    // get these exceptions.  Also, the _OB_id method is a
    // proprietary feature of ORBacus C++ generated code.
    CORBA::ValueFactory factory = new java::lang::Throwable_init;
    orb -> register_value_factory(java::lang::Throwable::_OB_id(),
      factory);
    factory -> _remove_ref();

    factory = new java::lang::Exception_init;
    orb -> register_value_factory(java::lang::Exception::_OB_id(),
      factory);
    factory -> _remove_ref();

    factory = new javax::ejb::CreateException_init;
    orb ->
      register_value_factory(javax::ejb::CreateException::_OB_id(),
        factory);
    factory -> _remove_ref();

    factory = new javax::ejb::RemoveException_init;
    orb ->
      register_value_factory(javax::ejb::RemoveException::_OB_id(),
        factory);
    factory -> _remove_ref();

    // Perform the work
    run(orb, argv[1]);

  } catch(const CORBA::Exception& ex) {
    // Handle any CORBA related exceptions
    cerr << ex._to_string() << endl;
    exit_code = 1;
  }

    // Release any ORB resources
    if (!CORBA::is_nil(orb)) {
      try {
        orb -> destroy();
      } catch(const CORBA::Exception& ex) {
        cerr << ex._to_string() << endl;
        exit_code = 1;
      }
    }
```

```
  return exit_code;
}
```

6. **Use your C++ compiler to compile all of the C++ files, including the Client.cpp file, to create a Client executable. Such tools vary widely across platforms, so consult your product documentation for instructions.**

▼ Deploy the Enterprise Bean

7. **The next step is to deploy the enterprise bean using your favorite application server. The following steps describe how to deploy the `LoggerEJB` component using the J2EE 1.3 Reference Implementation (RI).**

   1. Start the RI application from a terminal window or command prompt by typing:

      ```
      $J2EE_HOME/bin/j2ee -verbose
      ```

   2. When the J2EE 1.3 RI indicates "J2EE server startup complete", run the deployment tool from another terminal window or command prompt by typing:

      ```
      $J2EE_HOME/bin/deploytool
      ```

   3. From the deployment tool, select `File -> New -> Application.`
   4. In the Application File Name field, enter `Logger.ear` to indicate in which file to create the application.
   5. In the Application Display Name field, enter `Logger`
   6. Select OK to save the settings and close this dialog window.
   7. From the deployment tool, select `File -> New -> Enterprise Bean.`
   8. Select Next if you get the Introduction screen. If not, continue.
   9. In the New EnterpriseBean Wizard, select Edit in the Contents box.
   10. Expand the Available Files list, and add the following four `.class` files from our `ejbinterop` package: `Logger.class`, `LoggerHome.class`, `LoggerEJB.class`, `LogMessage.class`. Select OK, then Next.
   11. Select `Stateless Session Bean Type.`
   12. Select `ejbinterop.LoggerEJB` for the `Enterprise Bean Class`.
   13. Select `ejbinterop.LoggerHome` for the `Remote Home Interface`.
   14. Select `ejbinterop.Logger` for the `Remote Interface`.
   15. Select the Next button until you get to the `Security Settings` page.

16. Select the `Deployment Settings` button.

17. Select `Support Client Choice`.

18. Select OK to save the settings and close this dialog window.

19. Select Finish.

20. From the deployment tool, select, `Tools -> Deploy.`

21. If running the **Java RMI-IIOP client only**, select Return Client JAR.

22. Select Next.

23. Enter `ejbinterop/logger` in the `JNDI Name` for our LoggerEJB field.

24. Select Finish.

25. Select File -> Exit to exit the deploytool.

Now, the Logger application with our `LoggerEJB` components are deployed and ready to receive messages.

▼ Run the client executable

8. **Run the client executable. One way you can run the client executable is to enter the following URL in a terminal window from the directory containing the executable client file:**

```
Client corbaname:iiop:1.2@localhost:1050#ejbinterop/logger
```

In this URL,

- `Client` is the name of the application to run.
- `corbaname` specifies that we will resolve a stringified name from a specific naming context.
- `iiop:1.2` tells the ORB to use the IIOP protocol and GIOP 1.2.
- The host machine on which to find the reference is `localhost`, the local machine. To expand this example to run on two machines, enter the IP address or host name of the machine on which the server is running instead of `localhost`.
- `1050` is the port on which the naming service is listening for requests. By default in the J2EE v.1.3 RI, the default port the naming service listens on is port 1050. The portion of the reference up to this point at the hash mark (`Client corbaname:iiop:1.2@localhost:1050`) is the URL that returns the root naming context.
- `ejbinterop/logger` is the name to resolve in the naming context.

If you are using the J2EE 1.3 Reference Implementation, you should see a message similar to the following printed on the application server:

```
Sep 21, 2001 3:33:07 PM PDT: Message from a C++ client ejbinterop/
logger is the name to be resolved from the Naming Service.
```

▼ Stop the J2EE Server

9. **Stop the J2EE server. To stop the server, enter this command in a terminal window or command prompt.**

   ```
   $J2EE_HOME/bin/j2ee -stop
   ```

   Procedures for stopping running processes vary among operating systems, so if you are running a different server, consult your system documentation for details.

Creating a Java RMI-IIOP client application

Using the same example, we can easily develop a Java RMI-IIOP client that connects to an enterprise bean. The differences from the example using a C++ client are:

- In your client CLASSPATH, you must include the location of the client `.jar` file created by the J2EE application server running the desired enterprise bean. That `.jar` file contains the necessary client stubs.
- When deploying the application using the J2EE 1.3 RI, check the box `Return Client Jar` in the Deploytool on the first page of the Deploy screen.

The following code is the Java RMI-IIOP version of a client for our `LoggerEJB` component. Follow the same steps as those presented for the C++ client example. When running the client, use the same URL as in the C++ example.

CODE EXAMPLE 6 LogClient.java

```
package ejbinterop;
import java.rmi.RemoteException;
import javax.rmi.*;
import java.io.*;
import javax.naming.*;
import javax.ejb.*;

/**
 * Simple Java RMI-IIOP client that uses an EJB component.
 */
public class LogClient
```

```
{
    /**
     * Given a corbaname URL for a LoggerHome,
     * log a simple String message on the server.
     */
    public static void run(String loggerHomeURL)
        throws CreateException, RemoveException,
               RemoteException, NamingException
    {
        System.out.println("Looking for: " + loggerHomeURL);

        // Create an InitialContext. This will use the
        // CosNaming provider we will specify at runtime.
        InitialContext ic = new InitialContext();

        // Lookup the LoggerHome in the naming context
        // pointed to by the corbaname URL
        Object homeObj = ic.lookup(loggerHomeURL);

        // Perform a safe downcast
        LoggerHome home
            = (LoggerHome)PortableRemoteObject.narrow(homeObj,
                 LoggerHome.class);

        // Create a Logger EJB reference
        Logger logger = home.create();

        System.out.println("Logging...");

        // Log our message
        logger.logString("Message from a Java RMI-IIOP client");

        // Tell the application server we won't use this
        // EJB reference anymore
        logger.remove();

        System.out.println("Done");
    }

    /**
     * Simple main method to check arguments and handle
     * exceptions.
     */
    public static void main(String args[])
    {
        try {

            if (args.length != 1) {
             System.out.println("Args: corbaname URL of LoggerHome");
```

```
                System.exit(1);
            }

            LogClient.run(args[0]);

        } catch (Throwable t) {
            t.printStackTrace();
            System.exit(1);
        }
    }
}
```

Running the application with the Java RMI-IIOP client

When running the example application with a Java RMI-IIOP client instead of a C++ client, follow these steps:

1. **Compile the `.java` files in the `ejbinterop/` directory with the following command:**

   ```
   javac -classpath $J2EE_HOME/lib/j2ee.jar:<ejbinterop_directory>
     *.java
   ```

2. **Deploy the Enterprise JavaBean component as described in "Deploy the Enterprise Bean" on page 14. Remember to select Return Client JAR on the Tools -> Deploy page when running a Java RMI-IIOP client application. The section on deploying the RI also starts the J2EE RI.**

3. **Run the client application using a command similar to the following:**

   ```
   java -classpath $J2EE_HOME/lib/j2ee.jar:
   <path to LoggerClient.jar>/LoggerClient.jar:
   <directory_above_ejbinterop>:<ejbinterop_directory>
   ejbinterop.LogClient
   corbaname:iiop:1.2@localhost:1050#ejbinterop/logger
   ```

   In the window in which the J2EE RI is running, you will see this:

   ```
   Jan 31, 2002 2:27:47 PM PST: Message from a Java RMI-IIOP client
   ```

   In the window from where the client is run, you will see this:

   ```
   Looking for: corbaname:iiop:1.2@localhost:1050#ejbinterop/logger
   Logging...
   Done
   ```

4. **Stop the J2EE server.**

Beyond the basic application

This section contains the following information:

- "Where to go from here" on page 19
- "Tips for complex interfaces" on page 19
- "Links to similar examples" on page 20

Where to go from here

To enhance the application you could:

- Develop the example to use valuetypes.

  To do this, remove the `-noValueMethods` switch when you run `rmic`. Rerun your IDL to C++ language mapping compiler to verify that it supports the valuetypes that have been generated.
- Add another method to `Logger` that actually takes a `LogMessage`.

Tips for complex interfaces

The interfaces are key to the communication between clients and servers speaking different languages. To increase the probability of success in this area, consider the following suggestions:

- Avoid using complex Java classes, such as collections in `java.util`, for method parameters or return types.

  After these types are mapped to IDL, you will be forced to implement them in your client programming language. In addition, since Java Object Serialization and RMI-IIOP APIs allow the wire format and internal representation of classes to evolve over time, your CORBA client applications may be incompatible across Java™ 2 Platform, Standard Edition (J2SE™) implementations or versions.
- Start with IDL.

  You may want complex data structures in your return types or method parameters. In this case, try starting with IDL. Define data structures and even exceptions in IDL, and then use them in your EJB interfaces. This will prevent artifacts of the reverse mapping from creeping into your CORBA interfaces.

  For instance, try defining the `LogMessage` class in IDL initially, and then using the resulting class of a Java language to IDL compilation as a method parameter in the Logger EJB component.

- Avoid overloading in EJB interfaces.

  CORBA IDL does not support method overloading, and the Java language to IDL mapping specification handles this by creating IDL method definitions that combine the method name with all its IDL parameter types. This leads to very unfriendly method names for developers using languages other than the Java programming language.

- Consider using bridges.

  If the available options are still too limited or impact the code you wish to write, consider using a server–side bridge. You can read more about constructing such bridges from the sites listed in the links section.

Links to similar examples

Several vendors implementing J2EE technology have excellent examples and tips for integrating CORBA and Enterprise JavaBeans technology:

- IONA - *Calling Enterprise Beans from CORBA Clients* at http://www.iona.com/docs/iportal_application_server/3.0/DevelopGuide/html/intro-RMI.html#311099
- BEA - *EJB-to-CORBA/Java Simpapp Sample Application* at http://edocs.bea.com/wle/wle50/interop/ejbcorba.htm
- Borland - *Sevens steps to build a VisiBroker C++ CORBA Client for an EJB Server* at http://www.borland.com/devsupport/appserver/faq/ejbcpp/ejb_cpp.html