Contents

1. Introduction to CORBA Request-Level Interceptors
   Interceptor Architecture .............................................................. 1-2
   Capabilities and Limitations ......................................................... 1-3
   Execution Flow ........................................................................... 1-5
   Client-side Execution ................................................................. 1-5
   Client-side Exception Handling .................................................... 1-7
   Target-side Execution ................................................................. 1-9
   Target-side Exception Handling .................................................... 1-10
   The exception_occurred Method .................................................... 1-12
   About Short-circuit Behavior ....................................................... 1-12
   Using Multiple Request-Level Interceptors ................................... 1-12
   Multiple Client-side Interceptors .................................................. 1-15
   Multiple Target-side Interceptors .................................................. 1-15
   Interceptors and Meta-Operations ................................................ 1-16

2. Developing CORBA Request-Level Interceptors
   Step 1: Identify the Interfaces of Your CORBA Application ............... 2-2
   Step 2: Write the Interceptor Implementation Code .......................... 2-2
   Starting the Implementation File .................................................. 2-3
   Initializing the Interceptor at Run Time ........................................ 2-3
   Obtaining the Interface Name from a Request ................................ 2-4
   Identifying Operations in the Request ........................................... 2-5
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Deploying CORBA Request-Level Interceptors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Registering an Interceptor</td>
<td>3-1</td>
</tr>
<tr>
<td></td>
<td>Unregistering an Interceptor</td>
<td>3-2</td>
</tr>
<tr>
<td></td>
<td>Changing the Order in Which Interceptors Are Called</td>
<td>3-3</td>
</tr>
<tr>
<td>4</td>
<td>PersonQuery Sample Application</td>
<td></td>
</tr>
<tr>
<td></td>
<td>How the PersonQuery Sample Application Works</td>
<td>4-2</td>
</tr>
<tr>
<td></td>
<td>PersonQuery Database</td>
<td>4-2</td>
</tr>
<tr>
<td></td>
<td>Client Application Command-line Interface</td>
<td>4-3</td>
</tr>
<tr>
<td></td>
<td>The OMG IDL for the PersonQuery Sample Application</td>
<td>4-5</td>
</tr>
<tr>
<td></td>
<td>Building and Running the PersonQuery Sample Application</td>
<td>4-8</td>
</tr>
<tr>
<td></td>
<td>Copying the Files for the PersonQuery Sample Application</td>
<td>4-9</td>
</tr>
<tr>
<td></td>
<td>Changing the Protection on PersonQuery Application Files</td>
<td>4-12</td>
</tr>
<tr>
<td></td>
<td>Setting the Environment Variables</td>
<td>4-13</td>
</tr>
<tr>
<td></td>
<td>Building the CORBA Client and Server Applications</td>
<td>4-13</td>
</tr>
<tr>
<td></td>
<td>Start the PersonQuery Client and Server Applications</td>
<td>4-13</td>
</tr>
<tr>
<td></td>
<td>Running the PersonQuery Sample Application</td>
<td>4-13</td>
</tr>
<tr>
<td></td>
<td>Stopping the PersonQuery Sample Application</td>
<td>4-14</td>
</tr>
<tr>
<td>5</td>
<td>InterceptorSimp Sample Interceptors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>How the PersonQuery Sample Interceptors Work</td>
<td>5-1</td>
</tr>
<tr>
<td></td>
<td>Registering and Running the PersonQuery Interceptors</td>
<td>5-2</td>
</tr>
</tbody>
</table>

Implementing the Interceptor’s Response Operation ......................... 2-5
Reading Parameters Out of a Data Input Stream ............................ 2-6
Exceptions ................................................................................. 2-7
Step 3: Create the Interceptor Header File .................................. 2-7
Step 4: Build the Interceptor ................................................... 2-9
Step 5: Test the Interceptor ...................................................... 2-9

Using CORBA Request-Level Interceptors
Examining the Output of the Interceptors ....................................................... 5-3
Unregistering the Interceptors ........................................................................... 5-3
Unregistering the Interceptors ........................................................................... 5-4

6. InterceptorSec Sample Interceptors
   How the PersonQuery Sample Interceptors Work ........................................ 6-1
   How the InterceptorSec Target-side Interceptor Works ............................ 6-2
   Using the SecurityCurrent Object .............................................................. 6-3
   Obtaining the SecurityCurrent Object ....................................................... 6-3
   Creating the List of User Attributes ............................................................ 6-4
   Registering and Running the PersonQuery Interceptors ............................ 6-6
   Examining the Interceptor Output ............................................................... 6-7
   Unregistering the Interceptors ....................................................................... 6-7

7. InterceptorData Sample Interceptors
   InterceptorDataClient Interceptor ................................................................. 7-1
   InterceptorDataTarget Interceptor ............................................................... 7-2
   Implementing the InterceptorData Interceptors ........................................ 7-3
   Registering and Running the InterceptorData Interceptors ...................... 7-3
   Examining the Interceptor Output ............................................................... 7-4
   Unregistering the Interceptors ....................................................................... 7-6

8. Request-Level Interceptor API
   Interceptor Hierarchy .................................................................................. 8-2
   Note on Unused Interfaces ........................................................................... 8-2
   Interceptors::Interceptor Interface ............................................................ 8-3
   RequestLevelInterceptor::
      RequestInterceptor Interface ................................................................ 8-8
RequestLevelInterceptor::
    ClientRequestInterceptor Interface ........................................ 8-17

RequestLevelInterceptor::
    TargetRequestInterceptor Interface ....................................... 8-24

CORBA::DataInputStream Interface ............................................ 8-32

A. Starter Request-Level Interceptor Files

    Starter Implementation Code .............................................. A-1
    Starter Header File Code ................................................ A-10
Introduction to CORBA Request-Level Interceptors

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

A request-level interceptor is a user-written CORBA object that provides a means to insert functionality, such as security or monitoring components, into the invocation path between the client and server components of a CORBA application. When you have an interceptor installed and registered with an Object Request Broker (ORB) on a particular machine, the interceptor is involved with all the CORBA applications on that machine. You can use interceptors to insert any additional functionality into the invocation path of an object invocation, at either the client, or the server, or both ends of the invocation.

Request-level interceptors are not usually part of a typical CORBA environment. Implementing them is considered an advanced programming task.

The CORBA environment in the BEA Tuxedo system supports two categories of interceptors:

- Client-side interceptors, which are called by the ORB at the client side of an invocation and are run in the process of an entity making a request. Client-side interceptors inherit from the ClientRequestInterceptor class.
Target-side interceptors, which are called by the ORB at the target side of an invocation and are run with the target application process. The target of an invocation may be a CORBA server application or a CORBA joint client/server application. Target-side interceptors inherit from the TargetRequestInterceptor class.

The CORBA environment BEA Tuxedo system is very flexible about where you can install and use interceptors, with respect to the relative location of the client and target objects. It is transparent to a client application whether the target of its request is in the same or a different process.

Although client- and target-side interceptors inherit from separate interfaces, it is often convenient to implement the interceptors in a single source file.

Interceptor Architecture

The following figure shows the relationship between request-level interceptors and the BEA Tuxedo system.

Note the following about the BEA Tuxedo implementation of CORBA interceptors:

- Interceptors are registered administratively and are called by the ORB at the appropriate time during the execution of the application.

- When a client-side interceptor is installed and registered with an ORB, that interceptor is called with every request coming from any CORBA client application on that machine.
During the course of a single, successful request-response cycle of an invocation, a client-side interceptor is called twice by the ORB:

a. When the request is first issued from the client application and arrives at the ORB (the `client_invoke` operation)

b. When the target response arrives back at the client application process (the `client_response` operation)

- When a target-side interceptor is installed and registered with an ORB, that interceptor is called with every request that arrives for any target object on that machine.

During the course of a single request-response cycle of an invocation, a target-side interceptor is called twice by the ORB:

a. When the client request first arrives at the ORB (the `target_invoke` operation)

b. When the target object response arrives at the ORB (the `target_response` operation)

- You can install and register multiple client- or target-side interceptors with an ORB.
- Interceptors are independent of each other, and they do not require knowledge about the potential presence of other interceptors.
- Interceptors can _short-circuit_ an invocation by returning a response directly to the client without involving the target object at all.
- Interceptors impact overall application performance because they add an additional step in the execution of every request.

The ORB maintains a list of registered interceptors. Registering an interceptor is something you do as an administrative task. During application run time, the ORB uses this list to determine when to call the interceptors and in what order, because multiple interceptors can be installed and created. When you have multiple interceptors registered, the ORB executes each interceptor consecutively. Establishing the order in which multiple interceptors are called is also an administrative task.

**Capabilities and Limitations**

Request-level interceptors are especially useful for implementing several different types of service applications, such as:

- Instrumentation points for collecting statistics
- Probe points that include monitoring or tracing facilities
Security checks to determine whether a particular type of invocation should be permitted, or whether a specific bit of information can be returned to a client application. For more information about interceptors and security, see Chapter 6, “InterceptorSec Sample Interceptors.”

The following are current limitations on CORBA interceptors:

- Interceptors are called only by an ORB. Neither CORBA client nor server applications can call an interceptor directly.
- Interceptors implemented in a specific programming language can intercept invocations only from entities that are also implemented in that same language.
- Interceptors cannot write to the DataInputStream object.
- Interceptors cannot pass or manipulate the service context object.
- Interceptors cannot pass or manipulate the transaction current object.
- Interceptors cannot invoke methods on the Tobj_Bootstrap object.
- The REPLY_NO_EXCEPTION return status value is not supported, although it appears in the method signatures operations on interceptor classes.
- An interceptor can make invocations on other objects; however, those invocations are subject to interception as well. When an interceptor invokes an object, make sure the interceptor doesn’t intercept its own invocation in an infinite loop—which will happen if the object being invoked is in the same server process as the interceptor. In such a situation, the system can hang.
- The method signatures for operations on classes derived from the RequestLevelInterceptor interface include parameters for the following interfaces:
  - RequestLevelInterceptor::DataOutputStream
  - RequestLevelInterceptor::ServiceContextList

These interfaces are not used in the BEA Tuxedo product. These interfaces are defined in the BEA Tuxedo software so that you do not need to recompile your CORBA application if an implementation of these interfaces is ever provided in a future release of the BEA Tuxedo product. The ORB will always pass a nil object for the actual argument. You should not attempt to use these arguments; doing so will likely end the process with a serious error.
Execution Flow

The following sections explain what happens during the execution of a CORBA application that uses interceptors. In general, request-level interceptors are instantiated and initialized only when the ORB is initialized. At no other time can request-level interceptors be instantiated.

The return status of an interceptor controls the execution flow of the ORB run-time and any other request-level interceptors that may be installed.

Depending on the return status of an interceptor after it has been called, one of the following events may occur:

- The invocation resumes its normal path to the target object, back to the client application, or to another interceptor.
- The interceptor on either the client or the server side services the client request and sends an exception back to the client. (In this case, the request may never be sent to the target object, or the target object may provide a response that the interceptor replaces with an exception. This would happen transparently to the client application.)

Multiple request-level interceptors can be involved in a single invocation, and no interceptor needs to know about any other interceptor.

The events that take place during a request-response cycle of an invocation are presented in two categories:

- Client-side execution
- Target-side execution

Client-side Execution

Each interceptor is called twice during the request-response cycle of an invocation: once when a request is going from the client towards the target, and again when a response returns back to the client. The client interceptor class, `ClientRequestInterceptor`, has two corresponding operations, among others, for these two calls:

- `client_invoke()` — called when the request made on an object reference arrives at the client-side ORB.
- `client_response()` — called when the response is returned back towards the entity making the request.
The flow of execution of a CORBA application that uses a client-side interceptor is shown in Figure 1-1. This figure shows a basic and successful request-response invocation cycle (that is, no exceptions are raised).

Figure 1-1  Client-side Interceptor

In Figure 1-1, note the following events that are called out:

1. The request leaves the client and arrives at the ORB.
2. The ORB calls the client_invoke operation on the client-side interceptor. (The section “Using Multiple Request-Level Interceptors,” explains what happens when you have multiple client-side interceptors installed.)
3. The client-side interceptor processes the request and returns a status code to the ORB.
4. If no exception is returned as a result of the `client_invoke` operation, the request resumes its path toward the target object.

5. The target object processes the request and issues a response.

6. The response arrives back at the ORB, and the ORB calls the `client_response` operation on the interceptor.

7. The interceptor processes the response and returns a status code to the ORB.

8. The response is sent to the client application.

**Client-side Exception Handling**

The `client_invoke` and `client_response` operations each return a status value that indicates whether the client interceptor processing should continue. The interceptors may return exception status values, which cause exception handling to take place. Table 1-1 shows what happens depending on what status value is returned from these operations, and shows how the interceptors, together with the ORB, handle exceptions.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Return Status Value</th>
<th>What Happens</th>
</tr>
</thead>
<tbody>
<tr>
<td>client_invoke()</td>
<td>INVOKE_NO_EXCEPTION</td>
<td>The ORB continues normal processing of the request toward the target, calling other interceptors, if any.</td>
</tr>
<tr>
<td></td>
<td>REPLY_NO_EXCEPTION</td>
<td>The interceptor has serviced the request and no further process toward the target is needed. The request will be considered serviced as if the target processed it. Thus, the ORB short circuits the invocation and starts calling interceptors back towards the client. The client_response operation is not called on the same interceptor, but this operation on any previously invoked interceptor is called.</td>
</tr>
<tr>
<td></td>
<td>(In version 8.0 of the BEA Tuxedo product, the ORB cannot process this return value, so do not implement this as a return value in your interceptors.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REPLY_EXCEPTION</td>
<td>The interceptor returns an exception to the ORB. The ORB then calls each previous client-side interceptors’ exception_occurred operation. The exception_occurred method gives these previous interceptors an opportunity to clean up state before the ORB returns an exception back to the client application. Thus, the ORB short circuits the invocation, and the invocation is now complete. For more information about the exception_occurred method, see the section “The exception_occurred Method” on page 1-12.</td>
</tr>
<tr>
<td>client_response()</td>
<td>RESPONSE_NO_EXCEPTION</td>
<td>The ORB continues normal processing of the request toward the client, calling other interceptors, if any.</td>
</tr>
<tr>
<td></td>
<td>RESPONSE_EXCEPTION</td>
<td>The interceptor passes an exception back to the ORB, overriding any previous result of the request. The ORB invokes the exception_occurred method on each previous interceptor back towards the client, then returns an exception to the client application.</td>
</tr>
</tbody>
</table>
**Target-side Execution**

As on the client side, a target-side interceptor is called twice during a request-response cycle. Target-side interceptors inherit from the `TargetRequestInterceptor` class, which includes the following operations:

- `target_invoke()`—called when the request arrives at the ORB that is part of the target object process.
- `target_response()`—called when the response is sent back to the client.

The flow of execution of a CORBA application that uses a target-side interceptor is shown in **Figure 1-2**. This figure shows a basic and successful request-response invocation cycle (that is, no exceptions are raised).

**Figure 1-2  Target-side Interceptor**
In Figure 1-2, note the following events that are called out:

1. The client request arrives at the ORB.
2. The ORB calls the `target_invoke` operation on the target-side interceptor. (The section “Using Multiple Request-Level Interceptors,” explains what happens when you have multiple target-side interceptors installed.)
3. The target-side interceptor processes the request and returns a status code to the ORB.
4. If no exception is raised during the execution of the `target_invoke` operation, the request resumes its path toward the target object.
5. The target object processes the request and issues a response.
6. The target-side ORB calls the `target_response` operation on the interceptor.
7. The interceptor processes the response and returns a status code to the ORB.
8. The response is sent to the client application.

**Target-side Exception Handling**

Table 1-2 shows what happens to an invocation on the target side depending on what status values are returned by the `target_invoke` and `target_response` operations, explaining what happens when exceptions are thrown.
### Execution Flow

#### Using CORBA Request-Level Interceptors

<table>
<thead>
<tr>
<th>Operation</th>
<th>Return Status Value</th>
<th>What Happens</th>
</tr>
</thead>
<tbody>
<tr>
<td>target_invoke()</td>
<td>INVOKE_NO_EXCEPTION</td>
<td>The ORB continues normal processing of the request toward the target (the object implementation), calling other interceptors, if any.</td>
</tr>
<tr>
<td></td>
<td>REPLY_NO_EXCEPTION</td>
<td>The interceptor has serviced the request and no further process toward the target is needed. The request will be considered serviced as if the target processed it. Thus, the ORB short circuits the invocation and starts calling interceptors back towards the client. The target_response operation is not called on the same interceptor, but this operation on any previously invoked interceptor is called.</td>
</tr>
<tr>
<td></td>
<td>(In version 8.0 of the BEA Tuxedo product, the ORB cannot process this return value, so do not implement this as a return value in your interceptors.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>REPLY_EXCEPTION</td>
<td>The interceptor returns an exception to the ORB. The ORB then calls each previous target-side interceptors’ exception_occurred operation. The exception_occurred method gives these previous interceptors an opportunity to clean up state before the ORB returns an exception back to the client ORB. Thus, the target ORB short circuits the invocation, and the invocation is now complete. For more information about the exception_occurred method, see the section “The exception_occurred Method” on page 1-12.</td>
</tr>
<tr>
<td>target_response()</td>
<td>RESPONSE_NO_EXCEPTION</td>
<td>The ORB continues normal processing of the request toward the client, calling other interceptors, if any.</td>
</tr>
<tr>
<td></td>
<td>RESPONSE_EXCEPTION</td>
<td>The interceptor passes a new exception back ORB, overriding any previous result of the request. Instead of calling the target_response operation for interceptors on the way back to the client, the ORB calls the exception_occurred operation on those interceptors instead.</td>
</tr>
</tbody>
</table>
The `exception_occurred` Method

Every interceptor has the `exception_occurred` method, which the ORB may call under the following circumstances:

- The ORB has found an internal problem; for example, an operating system resource error or a communication problem.

- A different interceptor has set an exception (rather than an exception being generated by the ORB or the method). For example, the ORB is calling Interceptors A and B, respectively. Interceptor A has set an exception, so the ORB then calls the `exception_occurred` method on Interceptor B instead of the `client_response` or `target_response` methods. Your interceptor can take advantage of this behavior to examine both the context in which the response containing the exception is being processed and the actual value of the exception without reading the exception from the `DataInputStream` structure.

- The client application is using a deferred synchronous DII invocation on a `Request` object and then releases the `Request` object. In this case no response is delivered to the client.

When one of the preceding situations has occurred, calling the `exception_occurred` method is an alternative to calling the `client_response` or `target_response` methods; however, the effect is essentially the same in that the client invocation is complete.

For more information about keeping track of requests, see the section “Implementing the Interceptor’s Response Operation” on page 2-5.

About Short-circuit Behavior

As mentioned earlier, an interceptor can short-circuit a client request by servicing the request itself or by returning an exception. In either case, the client request is never actually serviced by the target object.

This short-circuit behavior works only in the `client_invoke` or `target_invoke` methods. It doesn’t apply to the `client_response` or `target_response` methods.

Using Multiple Request-Level Interceptors

Multiple request-level interceptors are installed in a queue such that the ORB can execute one after the other in a sequential order. The ORB gives each request-level interceptor the request in succession until there are no more request-level interceptors left in the queue to execute. If all interceptors indicate success, the request is processed. The ORB delivers the resulting response
to the transport in the client case, or to the object implementation in the target case. The ORB executes the interceptors servicing a response in the reverse order than that of servicing a request.

When an interceptor does not indicate success, a short circuit response results. This short circuit can be performed by the `client_invoke` or `target_invoke` operations. The status returned from the interceptor tells the ORB that the interceptor itself has decided to respond to the request with an exception, rather than to allow the target object to handle the request. (An interceptor’s `client_response` or `target_response` operation cannot perform any short-circuit behavior, but it can replace the target response.)

Each interceptor is normally unaware of the other interceptors, unless they explicitly share information. This independent programming model is preserved by the execution semantics with regards to short circuits: When an interceptor indicates that a response should be short-circuited and not reach its intended destination (which is the transport on the client side, and the object implementation on the target side), the response circulates back through the interceptors through which it has successfully passed. For example, if Interceptor A returns the status value `INVOKE_NO_EXCEPTION` after processing a `client_invoke` operation, expecting the request to be delivered, and the next Interceptor, B, denies the request with an exception, that exception gets put into the response and is delivered to Interceptor A’s `exception_occurred` operation. The analogous execution model on the target side is in effect also.

Figure 1-3 shows the sequence of execution when multiple client-side interceptors are installed on an ORB. (A similar series of operations occur with multiple target-side interceptors.)
In Figure 1-3, note the following events that are called out:

1. The client request arrives in the ORB, and the ORB calls Interceptors A through D in sequence.
2. The request goes to the target object.
3. The target object processes the request and returns a response.
4. The response arrives back at the ORB with the client-side interceptors. The ORB then calls each of the registered interceptors in a sequence that’s the reverse of the order in which they were called when the request went out.
5. The response arrives back at the client application.
Multiple Client-side Interceptors

When the ORB receives a request, the ORB calls each client-side interceptor’s client_invoke operation in turn. If the return value INVOKE_NO_EXCEPTION is returned from each client_invoke operation (the normal case), the resulting request is marshaled into a message by the ORB and sent to the target object.

Under the following circumstances, instead of calling the client_response operation on remaining interceptors back towards the client, the ORB calls the exception_occurred on those interceptors, and then returns an exception back to the client application:

- The return value from any client_invoke operation is REPLY_EXCEPTION.
  In this instance, the ORB ceases to propagate the request to remaining interceptors or to the transport. The ORB thus short-circuits the request.

- The return value from any client_response operation is RESPONSE_EXCEPTION.
  In this instance, the interceptor passes an exception back to the ORB, overriding any previous result of the request.

Multiple Target-side Interceptors

As with the client-side interceptor processing, the ORB calls each target-side interceptor’s target_invoke operation in succession. If the return value INVOKE_NO_EXCEPTION is returned from each target_invoke operation, the request is passed onto the target object.

Under the following circumstances, instead of calling the target_response operation on remaining interceptors back towards the client, the ORB calls the exception_occurred on those interceptors, and then returns an exception back towards the client application:

- The return value from any target_invoke operation is REPLY_EXCEPTION.
  In this instance, the ORB ceases to propagate the request to any remaining interceptors and the target object. At this point the ORB returns a response to the client ORB, and the target ORB short-circuits the request.

- The return value from any target_response operation is RESPONSE_EXCEPTION.
  In this instance, the interceptor passes an exception back to the ORB, overriding any previous result of the request.
Interceptors and Meta-Operations

Meta-operations are operations that support the CORBA Object interface, such as is_a, get_interface, and non_existent. Some meta-operations can be performed by the ORB without issuing an invocation, but other operations sometimes need to invoke the object; namely, the is_a, get_interface, and non_existent methods. These operations can thus trigger interceptors.

The CORBA-specified language binding of these operations converts the operation names from the names defined in IDL to the following:

- _is_a
- _interface
- _non_existent (or _not_existent)

If you are implementing a security-based interceptor, be aware of this behavior because the ORB may invoke these operations as part of a client request. You typically should avoid the situation where an interceptor permits only a specific set of client requests to be sent to a target object, but fails to take these meta-operations into account.
Developing CORBA Request-Level Interceptors

Developing a CORBA request-level interceptor typically involves the following steps:

- **Step 1: Identify the Interfaces of Your CORBA Application**
  Also identify the machines on which you plan to deploy the interceptors.

- **Step 2: Write the Interceptor Implementation Code**

- **Step 3: Create the Interceptor Header File**

- **Step 4: Build the Interceptor**

- **Step 5: Test the Interceptor**

The preceding steps are usually iterative. For example, the first time you build and test your interceptor, you might have only the most basic code in the interceptor that merely verifies that the interceptor is running. With subsequent builds and tests, you gradually implement the full functionality of the interceptor.

The sections that follow explain each of these steps in detail, using the sample interceptors packaged with the BEA Tuxedo software for examples.

**Notes:** The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code
samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Step 1: Identify the Interfaces of Your CORBA Application

Deploying an interceptor on a given machine constitutes a significant overhead because that interceptor will be invoked every time any application on that machine issues (in the case of a client-side interceptor) or receives (target-side interceptor) a request. Therefore, any interceptor you create must be well-matched to those applications.

For example, a security interceptor typically needs to know about what kinds of requests are of concern, and what kinds of data are being handled in the request.

Any interceptor that deals with specific requests needs to be able to extract the interface repository ID from the request. With that interface knowledge, the interceptor then has a way of knowing what kind of data is in the request, and can then handle that data in a request-specific fashion.

In addition, if a request is sent that is not of interest, the interceptor needs to be able to pass the request through quickly and efficiently.

The PersonQuery example described in Chapter 4, “PersonQuery Sample Application,” uses an interceptor that determines whether the user of the PersonQuery client application can receive addresses. If the identity of the user matches specific criteria, the interceptor allows the full address number to be returned to the client. If no match exists, the interceptor returns only the string of \( x \) characters to the log file in place of the address.

Step 2: Write the Interceptor Implementation Code

To implement an interceptor:

- For your first pass on implementing an interceptor, keep it simple. For example, you might decide for each function member to implement a statement that prints a message to a log file. This would simply verify that the interceptor is properly built, registered, and running. Once you know your interceptor is working properly, you can iteratively add code until you have all the functionality you need.
Step 2: Write the Interceptor Implementation Code

- If you are planning to deploy client- and target-side interceptors to implement a specific piece of functionality, you can implement both interceptors in a single source file. Then when you build and deploy the interceptors, you can configure them separately on the client- and target-side machines if you desire. The sample interceptors provided with the BEA Tuxedo software are done this way.

The topics that follow discuss implementation considerations that may be typical of many interceptors. Examples from the InterceptorData interceptors, which are described in Chapter 7, “InterceptorData Sample Interceptors,” are provided.

Starting the Implementation File

You can use the code fragments included in Appendix A as a place to start implementing your interceptor. You may use the code included in Appendix A, or you may copy the following starter files available at the WebLogic Enterprise Developer Center on the BEA Web site:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intercep.h</td>
<td>Interceptor header starter file. The contents of this file, and instructions for using it, are in the section “Step 3: Create the Interceptor Header File” on page -7.</td>
</tr>
<tr>
<td>intercep.cpp</td>
<td>Interceptor implementation starter file.</td>
</tr>
</tbody>
</table>

For information about getting these starter files from the WebLogic Enterprise Developer Center, see the Release Notes.

You can start your interceptor implementation using the sample interceptor code provided in Appendix A, where YourInterceptor represents the name of the interceptor you are implementing. The ORB will always pass nil references for the ServiceContextList and CORBA::DataOutputStream parameters. You should not use or reference those parameters. You should not test those parameters for nil because this restriction may change in a future version.

Initializing the Interceptor at Run Time

All interceptors are instantiated when the ORB is initialized. At no other time are request-level interceptors instantiated. As part of initializing, the interceptor’s initialization routine must instantiate an instance of an implementation for a client interceptor, or a target interceptor, or both, depending upon what the interceptor intends to support. As mentioned earlier, a single
shareable image can support both the client-side and target-side interceptors. The instances of any interceptor instantiated are then returned from the initialization routine and registered with the ORB run time.

The following code fragment is from the InterceptorData interceptor, and shows the declaration of the initialization operation invoked by the client-side ORB when that ORB is initialized:

```c
void InterceptorDataClientInit(  
    CORBA::ORB_ptr TheORB,  
    RequestLevelInterceptor::ClientRequestInterceptor ** ClientPtr,  
    RequestLevelInterceptor::TargetRequestInterceptor ** TargetPtr,  
    CORBA::Boolean * RetStatus)
```

The following code fragment shows the statements to instantiate the InterceptorData client interceptor class. Note that this fragment uses a class named tracker, which is used for keeping track of each incoming client request so that it can be matched with the response returned by the target object. The tracker class is described in the section “Identifying Operations in the Request” on page -5.

```c
ClientInterceptorData * client = new ClientInterceptorData(TheORB, tracker);
if  (!client)  
{  
tmpfile << "InterceptorDataClientInit: Client alloc failed"  
<< endl << endl;
    *RetStatus = CORBA_FALSE;
    delete tracker;
    return;
}
```

The following code fragment shows the statements to return the interceptor class to the ORB:

```c
*ClientPtr = client;  
*TargetPtr = 0;  
*RetStatus = CORBA_TRUE;
return;
```

**Obtaining the Interface Name from a Request**

If you have an interceptor that works with specific interfaces or requests, the interceptor needs a way to extract the interface ID associated with a request so that the interceptor can identify it and thus know how to handle the data in the request. For example, the InterceptorData interceptor manipulates the request parameters sent in requests from the PersonQuery application. To manipulate the request parameters, the interceptor needs to know which request is being sent.
The following code fragment from the InterceptorData sample shows the interface ID extracted from the RequestContext structure:

```c++
if (strcmp(request_context.interface_id.in(),
           PersonQuery::_get_interface_name()) != 0)
    return ret_status;
```

### Identifying Operations in the Request

Using the extracted interface ID, the InterceptorData sample uses a simple `switch` statement to identify the operation in the client request. That way, the interceptor will know what to do with the request parameters contained in the request.

The following code fragment shows the switch statement that checks for either the `Exit` operation or the operation to query the database for a person by name. Note the use of the `parser` object, which extracts operations from the request retrieved from the `tracker` object.

```c++
m_outfile << "   Operation:      " << request_context.operation << endl;
PQ parser;
PQ::op_key key = parser.MapOperation(request_context.operation.in());
switch (key) {
    default:
        m_outfile << "    ERROR: operation is not member of " << request_context.interface_id.in() << endl;
        return Interceptors::REPLY_EXCEPTION;
    case PQ::Exit:
        m_outfile << endl;
        return ret_status;
    case PQ::ByPerson:
        {
            PersonQuery::Person per;
            parser.GetByPerson(request_arg_stream, &per);
            m_outfile << "   Parameters:" << endl;
            m_outfile << per << endl;
        } break;
```

### Implementing the Interceptor’s Response Operation

Extracting an interface ID out of a client request is fairly straightforward. However, it’s not quite as simple to do that with a target response. If an interceptor needs to know what interface and
operation is associated with the response it receives from the ORB, it needs to have special logic for tracking requests. It is the interceptor’s responsibility to track requests coming from the client.

The InterceptorData samples implement a language object, called Tracker, that keeps a record of the target-bound requests, and then matches the target responses to them when those responses arrive back at the interceptor.

The client_response and target_response operations on the InterceptorData samples extract interface and operation information from the Tracker object when responses are returned from the target.

The following InterceptorData code fragment extracts the request associated with a response:

```cpp
RequestInfo * req_info = m_tracker->CompleteRequest(reply_context);
if (!req_info)
{
    m_outfile << "   unable to find request for this reply (must not be one we care about)" << endl << endl;
    return Interceptors::RESPONSE_NO_EXCEPTION;
}

// This is the interface we are expecting. Now identify the operation being invoked, so we can parse the request parameters.

m_outfile << "   ReplyStatus:    ";
OutputReplyStatus(m_outfile, reply_context.reply_status);

m_outfile << "   Interface:      " << req_info->intf() << endl;

m_outfile << "   Operation:      " << req_info->op() << endl;

PQ parser;
PQ::op_key key = parser.MapOperation(req_info->op());
```

Now that the interceptor has obtained the request associated with the response, the interceptor can handle the data in the response appropriately.

**Reading Parameters Out of a Data Input Stream**

The following code fragment shows an example of how the InterceptorData sample places the request parameters from a data stream into a structure. The parameter $S$ in the following code fragment represents a pointer to a `DataInputStream` structure that can be used by the interceptor implementation to retrieve the value of the reply parameters of the PersonQuery operation. The code encapsulated by the braces in this code fragment extracts the parameters of the response.
Step 3: Create the Interceptor Header File

from the DataInputStream structure. For more information about the DataInputStream structure, see Chapter 8, “Request-Level Interceptor API.”

void PQ::get_addr(CORBA::DataInputStream_ptr S, 
    PersonQuery::Address *addr)
{
    addr->number = S->read_short();
    addr->street = S->read_string();
    addr->town = S->read_string();
    addr->state = S->read_string();
    addr->country = S->read_string();
}

Exceptions

Exceptions from interceptors returned via the excep_val parameter can only be a derived type from the CORBA::SystemException base class. (Any other exception type that the interceptor implementations return to the ORB is converted by the ORB to a CORBA::UNKNOWN exception, which is passed via the excep_val parameter.) You need to map exceptions to a CORBA::SystemException class or one of its derivatives.

Step 3: Create the Interceptor Header File

After you have created any implementation code in the interceptor implementation file, you need to provide any data or operations as needed to the interceptor header file.

The following code example shows basic information that is required in the header file for an interceptor implementation file that implements both client- and target-side interceptors.

This example also shows:

- The include file needed for security
- Target data members for security

In this code example, YourInterceptor represents the name of the interceptor you are creating.

#include <CORBA.h>
#include <RequestLevelInterceptor.h>
#include <security_c.h>        //for security

class YourInterceptorClient : public virtual 
    RequestLevelInterceptor::ClientRequestInterceptor
private:
    YourInterceptorClient() {}
    CORBA::ORB_ptr m_orb;
public:
    YourInterceptorClient(CORBA::ORB_ptr TheOrb);
    ~YourInterceptorClient() {}
    Interceptors::ShutdownReturnStatus shutdown(
        Interceptors::ShutdownReason reason,
        CORBA::Exception_ptr & excep_val);
    CORBA::String id();
    void exception_occurred(
        const RequestLevelInterceptor::ReplyContext & reply_context,
        CORBA::Exception_ptr excep_val);
    Interceptors::InvokeReturnStatus client_invoke(
        const RequestLevelInterceptor::RequestContext & request_context,
        RequestLevelInterceptor::ServiceContextList_ptr service_context,
        CORBA::DataInputStream_ptr request_arg_stream,
        CORBA::DataOutputStream_ptr reply_arg_stream,
        CORBA::Exception_ptr & excep_val);
    Interceptors::ResponseReturnStatus client_response(
        const RequestLevelInterceptor::ReplyContext & reply_context,
        RequestLevelInterceptor::ServiceContextList_ptr service_context,
        CORBA::DataInputStream_ptr arg_stream,
        CORBA::Exception_ptr & excep_val);
};

class YourInterceptorTarget : public virtual RequestLevelInterceptor::TargetRequestInterceptor
{
private:
    YourInterceptorTarget() {}
    CORBA::ORB_ptr m_orb;
    SecurityLevel1::Current_ptr m_security_current; //for security
    Security::AttributeTypeList * m_attributes_to_get; //for security
public:
    YourInterceptorTarget(CORBA::ORB_ptr TheOrb);
    ~YourInterceptorTarget();
    Interceptors::ShutdownReturnStatus shutdown(
        Interceptors::ShutdownReason reason,
        CORBA::Exception_ptr & excep_val);
    CORBA::String id();
    void exception_occurred(
        const RequestLevelInterceptor::ReplyContext & reply_context,
        CORBA::Exception_ptr excep_val);
    Interceptors::InvokeReturnStatus target_invoke(
        const RequestLevelInterceptor::RequestContext & request_context,
        RequestLevelInterceptor::ServiceContextList_ptr service_context,
Step 4: Build the Interceptor

CORBA::DataInputStream_ptr request_arg_stream,
CORBA::DataOutputStream_ptr reply_arg_stream,
CORBA::Exception_ptr & excep_val);
Interceptors::ResponseReturnStatus target_response (  
const RequestLevelInterceptor::ReplyContext & reply_context,
RequestLevelInterceptor::ServiceContextList_ptr service_context,
CORBA::DataInputStream_ptr arg_stream,
CORBA::Exception_ptr & excep_val);

};

Step 4: Build the Interceptor

Interceptors are built into shareable libraries. Therefore, the steps for building an interceptor are platform-specific. For details about the specific commands and options used to build interceptors on any particular platform, execute the makefile that builds the interceptor sample applications provided with the BEA Tuxedo software, and view the results of the build in the log file that results from the build.

The command to build the sample interceptors is as follows:

Windows 2003
> nmake -f makefile.nt

UNIX
> make -f makefile.mk

For more information about building and running the sample interceptors provided with the BEA Tuxedo software, see Chapter 4, “PersonQuery Sample Application.”

Step 5: Test the Interceptor

Testing an interceptor requires you to perform the following tasks:

- Register the interceptor
- Boot the CORBA server application using the tmboot command
- Run the CORBA client application
- Check the interceptor’s log file to verify the interceptor’s behavior

For information about registering interceptors, see Chapter 3, “Deploying CORBA Request-Level Interceptors.”
Deploying CORBA Request-Level Interceptors

There are three administrative tasks associated with managing the registration of CORBA request-level interceptors:

- Registering an Interceptor
- Unregistering an Interceptor
- Changing the Order in Which Interceptors Are Called

This section explains these three tasks.

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Registering an Interceptor

You use the `epifreg` command to register your interceptors with an ORB. When you register an interceptor, the interceptor is added to the end of the list of interceptors already registered with the ORB. This is important when you have multiple interceptors registered with an ORB.

The syntax of the `epifreg` command for registering interceptors is the following:
epifreg -t bea/wle -i AppRequestInterceptor \
    -p <InterceptorName> -f <FileName> -e <EntryPoint> \
    -u "DisplayName=<Administrative Name>" -v 1.0

In the preceding command line:

- **InterceptorName** represents the name of the interceptor registered with the ORB, and the
  name you choose needs to be unique among those previously registered. You use this name
  for specifying the order of multiple interceptors and for unregistering an interceptor. The
  **FileName**, **EntryPoint**, and **DisplayName** arguments that follow are associated with this
  name.

- **FileName** represents the location of the file containing the implementation of the
  interceptor. This name is operating system and language dependent. This file is a sharable
  image file.

- **EntryPoint** represents a string value that is the name of the entry point for the interceptor.
  This name is programming language specific. This value is the name of the initialization
  function in the shareable image that creates an instance of the interceptor.

- **DisplayName** specifies a string value used for administrative functions and other reporting
  purposes. This name is strictly an administrative name.

**Note:** When you register an interceptor on a machine on which BEA Tuxedo CORBA server
processes are already running, those processes will not be subject to interception. Only
those processes that are started after an interceptor is registered are subject to
interception. If you want to make sure that all CORBA server processes are subject to
interception, make sure that you register you interceptors before you boot any CORBA
server processes.

**Unregistering an Interceptor**

Use the `epifunreg` command to unregister an interceptor from an ORB. This command has the
following syntax:

epifunreg -t bea/wle -p <InterceptorName>

The argument `<InterceptorName>` is the same case-insensitive name specified in the `epifreg`
command. Unregistering an interceptor takes it out of the interceptor order.
Changing the Order in Which Interceptors Are Called

You can see the order in which interceptors are registered, and thus called, by using the following command:

```
epifregedit -t bea/wle -g -k SYSTEM/interfaces/AppRequestInterceptor
```

The `epifregedit` displays the order in which interceptors are executed when the ORB receives a request.

You can change the order in which the interceptors are executed using the following command:

```
epifregedit -t bea/wle -s -k SYSTEM/interfaces/AppRequestInterceptor \
-a Selector=Order -a Order=<InterceptorName1>,<InterceptorName2>,...
```

Each `<InterceptorName>` is the case-insensitive name of the interceptor that must have been previously registered. This command replaces the order currently in the registry. The `epifregedit` command must specify every interceptor that you want to have loaded and executed by the ORB. If an interceptor is still registered and if you do not specify its name using `epifregedit` command, the interceptor is not loaded.
PersonQuery Sample Application

To understand and use the interceptor examples packaged with the BEA Tuxedo software, you need to build and run the PersonQuery sample application. The PersonQuery sample application itself does not contain any interceptors; however, this application is used as the basis for the sample interceptor applications that are described in the three chapters that follow.

This topic includes the following sections:

- How the PersonQuery Sample Application Works
- The OMG IDL for the PersonQuery Sample Application
- Building and Running the PersonQuery Sample Application

**Notes:** The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.
How the PersonQuery Sample Application Works

The PersonQuery sample application implements a simple database query interface. Using the PersonQuery application, a user can get information about people in the database that match specific search criteria, such as:

- Physical characteristics, such as age, weight, hair color, eye color, or skin color
- Name, address, or other details

The PersonQuery application contains the following components:

- A client application, which issues requests that contain a variety of data types as parameters. The client application accepts command line input from the user in a specific form, packages the input according to the sample interface, and sends the appropriate request.

  When the client application receives the result of the query from the server, it will report the number of items that were found. The user can then enter the command that displays the result of the latest query, or specify a new query.

- A server application, which contains a simple, built-in database. The server application accesses the database to service the client request.

PersonQuery Database

The PersonQuery database in the server application contains the following information about each person in the database:

- Name
- Address
- U.S. Social Security number
- Sex
- Age
- Marital status
- Hobby
- Date of birth
- Height
How the PersonQuery Sample Application Works

- Weight
- Hair color
- Skin color
- Eye color
- Other physical characteristics

Client Application Command-line Interface

The PersonQuery sample application implements a simple command-line interface in the client component with which the user can enter database query commands and the command to exit from the application.

The database query commands have the following syntax:

Option? command [keyword] [command [keyword]]...

In this command syntax:

- **Option?** is the PersonQuery command prompt.
- **command** is one of the PersonQuery commands from Table 4-1.
- **keyword** is one of the keywords from Table 4-1. Note the following rules on specifying keywords:
  - Compound keywords, as typically supplied for the name and address commands, must be separated by spaces and enclosed in double-quote characters (""), as in the following command:
    
    Option? name "Thomas Mann"
  - When specifying an address, always separate street name, city name, state or province, country name, and other parts of the address with commas, as in the following command:
    
    Option? address "116 Einbahnstrasse, Frankfurt am Main, BRD"
- You may specify multiple commands in a single line, as in the following example:
    
    Option? hair brown eyes blue
<table>
<thead>
<tr>
<th>Command</th>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>&quot;firstname lastname&quot;</td>
<td>Queries by name. Strings with spaces must be quoted.</td>
</tr>
<tr>
<td>address</td>
<td>&quot;number street, city...&quot;</td>
<td>Queries by address. Strings with spaces must be quoted. Address parts are street number, street, town, state, and country. Entries for street, town, state, and country must be separated by commas.</td>
</tr>
<tr>
<td>ss</td>
<td>xxx-xx-xxxx</td>
<td>Queries by U.S. Social Security number. The keyword must in the form xxx-xx-xxxx.</td>
</tr>
<tr>
<td>sex</td>
<td>sex</td>
<td>Queries by sex. Choices are male, female, and cant_tell.</td>
</tr>
<tr>
<td>age</td>
<td>age</td>
<td>Queries by age.</td>
</tr>
<tr>
<td>marriage</td>
<td>status</td>
<td>Queries by marital status. Choices are married, single, divorced, and not_known.</td>
</tr>
<tr>
<td>hobby</td>
<td>hobby</td>
<td>Queries by hobby. Choices are who_cares, rocks, swim, tv, stamps, photo, and weaving.</td>
</tr>
<tr>
<td>dob</td>
<td>mm/dd/yyyy</td>
<td>Queries by date. The keyword must be in the form mm/dd/yyyy.</td>
</tr>
<tr>
<td>height</td>
<td>inches</td>
<td>Queries by height, in inches.</td>
</tr>
<tr>
<td>weight</td>
<td>pounds</td>
<td>Queries by weight, in pounds.</td>
</tr>
<tr>
<td>hair</td>
<td>color</td>
<td>Queries by hair color. Choices are white, black, red, brown, green, yellow, blue, gray, and unknown.</td>
</tr>
<tr>
<td>skin</td>
<td>color</td>
<td>Queries by skin color. Choices are white, black, brown, yellow, green, and red.</td>
</tr>
</tbody>
</table>
Listing 4-1 provides the OMG IDL code for the implemented in the PersonQuery sample application.

**Listing 4-1  OMG IDL Code for the PersonQuery Application Interfaces**

```
#pragma prefix "beasys.com"

interface PersonQuery
{
    enum MONTHS {Empty, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
                 Sep, Oct, Nov, Dec};
    struct date_
    {
        MONTHS Month;
        short Day;
        short Year;
    };
    typedef date_ Date;
    struct addr_
    {
        short number;
        string street;
    };
}
```

The OMG IDL for the PersonQuery Sample Application

Table 4-1  PersonQuery Application Commands and Keywords

<table>
<thead>
<tr>
<th>Command</th>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eyes</td>
<td>color</td>
<td>Queries by eye color. Choices are blue, brown, gray, green, violet, black, and hazel.</td>
</tr>
<tr>
<td>other</td>
<td>feature</td>
<td>Queries by other physical features. Choices are tattoo, limb (that is, a limb is missing), scar, and none.</td>
</tr>
<tr>
<td>result</td>
<td></td>
<td>Displays the result of last query on output.</td>
</tr>
<tr>
<td>exit</td>
<td></td>
<td>Displays bill for services rendered and closes application.</td>
</tr>
</tbody>
</table>
string town;
string state;
string country;
}
typedef addr_ Address;
enum MARRIAGE {not_known, single, married, divorced};
enum HOBBIES {who_cares, rocks, swim, tv, stamps, photo, weaving};
enum SEX {cant_tell, male, female};
enum COLOR {white, black, brown, yellow, red, green, blue, gray, violet, hazel, unknown, dontcare};
enum MARKINGS {dont_care, tattoo, scar, missing_limb, none};
struct person_ {
    string name;
    Address addr;
    string ss;
    SEX sex;
    short age;
    MARRIAGE mar;
    HOBBIES rec;
    Date dob;
    short ht;
    long wt;
    COLOR hair;
    COLOR eye;
    COLOR skin;
    MARKINGS other;
}
typedef person_ Person;
typedef sequence <Person> Possibles;
union reason_ switch (short)
{
    case 0: string name;
    case 1: Address addr;
    case 2: string ss;
    case 3: SEX sex;
    case 4: short age;
}
case 5:    MARRIAGE mar;
case 6:    HOBBIES rec;
case 7:    Date dob;
case 8:    short ht;
case 9:    long wt;
case 10:   COLOR hair;
case 11:   COLOR eyes;
case 12:   COLOR skin;
case 13:   MARKINGS other;
}

typedef reason_ Reason;

exception DataOutOfRange
{
    Reason why;
};

boolean    findPerson (in Person who, out Possibles hits)
raises (DataOutOfRange);

boolean    findPersonByName (in string name, out Possibles hits)
raises (DataOutOfRange);

boolean    findPersonByAddress (in Address addr, out Possibles hits)
raises (DataOutOfRange);

boolean    findPersonBySS (in string ss, out Possibles hits)
raises (DataOutOfRange);

boolean    findPersonByAge (in short age, out Possibles hits)
raises (DataOutOfRange);

boolean    findPersonByMarriage (in MARRIAGE mar, out Possibles hits)
raises (DataOutOfRange);

boolean    findPersonByHobbies (in HOBBIES rec, out Possibles hits)
raises (DataOutOfRange);

boolean    findPersonBydob (
Building and Running the PersonQuery Sample Application

To build and run the PersonQuery sample application:

1. Copy the files for the PersonQuery sample application into a work directory.
2. Change the protection of the files for the PersonQuery sample application.
3. Set the environment variables.
4. Build the CORBA client and server sample applications.
5. Start the PersonQuery client and server applications.
6. Using the client application, enter a number of commands to search the database on the server.
7. Stop the PersonQuery sample application.

**Copying the Files for the PersonQuery Sample Application**

The request-level interceptor sample application files are located in the following directory:

$TUXDIR\samples\corba\interceptors_cxx

To create a copy of these files so that you can build them and run them, do the following:

1. Create a working directory into which to copy the sample files.

2. Copy the interceptors_cxx samples to the working directory created in the previous step:

   **Windows 2003**
   
   ```
   > xcopy /s/i %TUXDIR%\samples\corba\interceptors_cxx <workdirectory>\cxx
   ```

   **UNIX**
   
   ```
   > cp -R $TUXDIR/samples/corba/interceptors_cxx <workdirectory>/cxx
   ```

3. Change to the working directory containing the sample files:

   **Windows 2003**
   
   ```
   > cd <workdirectory>\cxx
   ```

   **UNIX**
   
   ```
   > cd <workdirectory>/cxx
   ```

You will use the files listed and described in Table 4-2 in the PersonQuery sample application.
Table 4-2  Files Included in the Interceptors Sample Applications

<table>
<thead>
<tr>
<th>Directory</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>app_cxx</code> (subdirectory under <code>interceptors_cxx</code>)</td>
<td><code>Readme.txt</code></td>
<td>The file that provides the latest information about building and running the set of interceptor sample applications.</td>
</tr>
<tr>
<td></td>
<td><code>makefile.mk</code></td>
<td>The makefile for building the entire set of interceptor sample applications (the PersonQuery application and all the sample interceptors) on UNIX systems.</td>
</tr>
<tr>
<td></td>
<td><code>makefile.nt</code></td>
<td>The makefile for building the entire set of interceptors sample applications (the PersonQuery application and all the sample interceptors) on Windows 2003 systems.</td>
</tr>
<tr>
<td></td>
<td><code>makefile.inc</code></td>
<td>The generic makefile that uses the macros defined in the appropriate <code>platform.inc</code> file.</td>
</tr>
<tr>
<td></td>
<td><code>personquery_i.h</code> and <code>personquery_i.cpp</code></td>
<td>The implementation of the PersonQuery interfaces.</td>
</tr>
<tr>
<td></td>
<td><code>personqueryc.cpp</code></td>
<td>The PersonQuery client application source file.</td>
</tr>
<tr>
<td></td>
<td><code>personquerys.cpp</code></td>
<td>The PersonQuery database server source file.</td>
</tr>
<tr>
<td></td>
<td><code>setenv.ksh</code></td>
<td>The shell file that sets all the required environment variables for building the entire set of interceptor sample applications on UNIX systems.</td>
</tr>
<tr>
<td></td>
<td><code>setenv.cmd</code></td>
<td>The command file that sets all the required environment variables for building the entire set of interceptor sample applications on Windows 2003 systems.</td>
</tr>
</tbody>
</table>
### Table 4-2  Files Included in the Interceptors Sample Applications

<table>
<thead>
<tr>
<th>Directory</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_cxx</td>
<td>InterceptorData.cpp</td>
<td>The InterceptorData C++ source file.</td>
</tr>
<tr>
<td></td>
<td>InterceptorData.h</td>
<td>The InterceptorData class definition file.</td>
</tr>
<tr>
<td></td>
<td>makefile.inc</td>
<td>The generic makefile that uses the macros defined in the appropriate platform.inc file to build the InterceptorData interceptors.</td>
</tr>
<tr>
<td></td>
<td>makefile.mk</td>
<td>The makefile that builds the InterceptorData interceptors on UNIX systems.</td>
</tr>
<tr>
<td></td>
<td>makefile.nt</td>
<td>The makefile that builds the InterceptorData interceptors on Windows 2003 systems.</td>
</tr>
<tr>
<td>simple_cxx</td>
<td>InterceptorSimp.cpp</td>
<td>The InterceptorSimp C++ source file.</td>
</tr>
<tr>
<td>(subdirectory under</td>
<td>InterceptorSimp.h</td>
<td>The InterceptorSimp class definition file.</td>
</tr>
<tr>
<td>interceptors_cxx)</td>
<td>makefile.inc</td>
<td>The generic makefile that uses the macros defined in the appropriate platform.inc file to build the InterceptorSimp interceptors.</td>
</tr>
<tr>
<td></td>
<td>makefile.mk</td>
<td>The makefile that builds the InterceptorSimp interceptors on UNIX systems.</td>
</tr>
<tr>
<td></td>
<td>makefile.nt</td>
<td>The makefile that builds the InterceptorSimp interceptors on Windows 2003 systems.</td>
</tr>
<tr>
<td>security_cxx</td>
<td>InterceptorSec.cpp</td>
<td>The InterceptorSec C++ source file.</td>
</tr>
<tr>
<td>(subdirectory under</td>
<td>InterceptorSec.h</td>
<td>The InterceptorSec class definition file.</td>
</tr>
<tr>
<td>interceptors_cxx)</td>
<td>makefile.inc</td>
<td>The generic makefile that uses the macros defined in the appropriate platform.inc file to build the InterceptorSec interceptors.</td>
</tr>
<tr>
<td></td>
<td>makefile.mk</td>
<td>The makefile that builds the InterceptorSec interceptors on UNIX systems.</td>
</tr>
<tr>
<td></td>
<td>makefile.nt</td>
<td>The makefile that builds the InterceptorSec interceptors on Windows 2003 systems.</td>
</tr>
</tbody>
</table>
Changing the Protection on PersonQuery Application Files

During the installation of the BEA Tuxedo software, the sample application files are marked read-only. Before you can edit or build the files in the PersonQuery sample application, you need to change the protection attribute of the files you copied into your work directory, as follows. First make sure you are in the working directory into which you copied the sample application files.

**Windows 2003**

```
prompt>attrib -r /s .*.*
```

**UNIX**

```
prompt>/bin/ksh
ksh prompt>chmod -R u+w .*.*
```
Setting the Environment Variables

Before building and running the PersonQuery sample application, you need to set the environment in which the application runs. To set the environment variables and other property settings needed to build and run the PersonQuery sample application, enter the following command:

Windows 2003
> setenv.cmd

UNIX:
> $ . ./setenv.ksh

Building the CORBA Client and Server Applications

The following command builds the PersonQuery application, creates a machine-specific UBBCONFIG file, and loads the UBBCONFIG file:

Windows 2003
> nmake -f makefile.nt

UNIX
$ make -f makefile.mk

Note: For convenience, the makefile executed in this step builds the entire set of interceptor samples. This includes the InterceptorSimp, InterceptorSec, and InterceptorData interceptors as well. Details on implementing and building those interceptors, as well as running them with the PersonQuery sample application, are provided in the chapters that follow.

Start the PersonQuery Client and Server Applications

Start the PersonQuery sample application by entering the following command:

prompt> tmboot -y

Running the PersonQuery Sample Application

A typical usage scenario of the PersonQuery sample application involves the following steps:

1. Enter a query command for one feature, and check for number of returned items, for example:
Option? hair brown eyes blue

2. Enter additional query data about the feature queried in the preceding step.
3. Continue queries until all the query data is narrowed down to a desirable level.
4. Enter the result command to see the final query result.
5. Start a new query cycle.
6. Enter the exit command to quit from the application.

Stopping the PersonQuery Sample Application

To stop the PersonQuery sample application, enter the following command:

prompt>tmshutdown -y
This topic includes the following sections:

- How the PersonQuery Sample Interceptors Work
- Registering and Running the PersonQuery Interceptors
- Examining the Output of the Interceptors
- Unregistering the Interceptors

Before trying out the steps described in this chapter, make sure you have completed all the steps described in Chapter 4, “PersonQuery Sample Application.”

### How the PersonQuery Sample Interceptors Work

The InterceptorSimp sample interceptor shows how the operation in a request passed to an interceptor can be accessed via a RequestContext object. When the InterceptorSimp sample intercepts a request, the interceptor does the following:

- Writes the operation name out to a data file, but does not interpret or modify the parameters in the request
- Returns appropriate status from the interceptor methods

Assuming a successful call to the interceptor, the client invocation is passed onto the target object and serviced in the usual way. Thus the InterceptorSimp sample interceptor shows the following:
An implementation of a basic monitoring service, which simply tracks each operation on the target object that has been invoked.

How an interceptor can identify the operation contained in the request by accessing the parameters passed by the ORB to the interceptor methods.

The InterceptorSimp sample interceptor also shows two different interceptors being defined and registered, but implemented in a single source file. In this example, the client and target interceptors are registered separately, with the client interceptor initialized first.

### Registering and Running the PersonQuery Interceptors

When you run the makefile that builds the PersonQuery sample application in Chapter 4, “PersonQuery Sample Application,” the entire set of sample interceptors are built as well, including the InterceptorSimp interceptor. This section describes how to register the InterceptorSimp interceptor so that it works with PersonQuery application at run time.

To register and run the InterceptorSimp client and server interceptors:

1. Change directory to the InterceptorSimp sample directory, where `workdirectory` represents the name of the directory into which you copied the interceptor sample applications in Chapter 4, “PersonQuery Sample Application.”

   **Windows 2003**
   ```
   > cd <workdirectory>\cxx\simple_cxx
   ```

   **UNIX**
   ```
   $ cd <workdirectory>/cxx/simple_cxx
   ```

2. Register the interceptor:

   **Windows 2003**
   ```
   > nmake -f makefile.nt config
   ```

   **UNIX**
   ```
   $ make -f makefile.mk config
   ```

3. Boot the CORBA server and run the client:

   **Windows 2003**
   ```
   > cd <workdirectory>\cxx\app_cxx
   > tmboot -y
   > PersonQueryClient
   ```
Examining the Output of the Interceptors

UNIX

> cd <workdirectory>/cxx/app_cxx
> tmboot -y
> PersonQueryClient

4. Perform any number of invocations using the PersonQuery client application, using the command syntax described in Chapter 4, “PersonQuery Sample Application.”

5. Stop the PersonQuery application:
> tmshutdown -y

Examining the Output of the Interceptors

The output from the simple client interceptor is in files named with the following syntax:
InterceptorSimpClientxxxx.out

In the preceding syntax line, xxxx represents the process ID of the executable within which the interceptor ran. For example, there are three InterceptorSimpClientxxxx.out files; one each for the following:

- The FactoryFinder, TMFFNAME
- The PersonQueryServer
- The PersonQueryClient

The content of each file varies according to how the ORB interacted with the executable. For example, target interceptors run on servers and client interceptors run on clients, so the InterceptorSimpClient log files typically have very little output from the target interceptor, but it has more output from the client interceptor.

Unregistering the Interceptors

After you have run the PersonQuery sample application with the InterceptorSimp sample interceptors, you can unregister those interceptors using the following steps:

1. Shut down all running CORBA applications by entering the following command:
   > tmshutdown -y

2. Unregister the interceptors.
Unregistering the Interceptors

To unregister the InterceptorSimp client and server interceptors:

1. Change directory to the InterceptorSimp sample directory, where `workdirectory` represents the name of the directory into which you copied the interceptor sample applications in Chapter, “PersonQuery Sample Application:”

   **Windows 2003**
   
   `> cd <workdirectory>cxx\simple_cxx`

   **UNIX**
   
   `$ cd <workdirectory>/cxx/simple_cxx`

2. Unregister the interceptors:

   **Windows 2003**
   
   `> nmake -f makefile.nt unconfig`

   **UNIX**
   
   `$ make -f makefile.mk unconfig`
This topic includes the following sections:

- How the PersonQuery Sample Interceptors Work
- Registering and Running the PersonQuery Interceptors
- Examining the Interceptor Output
- Unregistering the Interceptors

Before trying out the steps described in this chapter, make sure you have completed all the steps described in Chapter 4, “PersonQuery Sample Application.”

How the PersonQuery Sample Interceptors Work

The InterceptorSec sample interceptors show a simple client/server interceptor pair that implement a basic security model. The InterceptorSec client-side interceptor simply logs each client request that is handled by the ORB. The InterceptorSec target-side interceptor implements a simple security mechanism that checks to see whether the user of the client application is authorized to perform the operation in the request.

The InterceptorSec sample interceptors show the client and target interceptor pair initialized through a single initialization function and implemented in a single library. Since a single initialization function is called, the interceptor registration command registers one initialization function and one implementation library.
How the InterceptorSec Target-side Interceptor Works

When the target-side ORB receives a request, the ORB calls the InterceptorSec target-side interceptor and passes the RequestContext and DataInputStream objects from the client request.

The target-side interceptor then does the following to authorize the user of the client application for the operation contained in the request:

1. Checks to see if the request is an invocation on the PersonQuery interface. If it is not, the interceptor returns a INVOKE_NO_EXCEPTION.

2. If the operation contained in the request is an invocation on the PersonQuery interface, the interceptor:
   a. Obtains a reference to the SecurityCurrent object, which the interceptor then narrows.
   b. Invokes the SecurityContext object, requesting the attribute list for the user of the client application.
   c. Walks through the attribute list to obtain two attributes:
      d. Matches the user against the PrimaryGroupId and the AccessId. If the user successfully matches the criteria for these two attributes, the interceptor returns INVOKE_NO_EXCEPTION.
      e. If no match is found, the interceptor returns REPLY_EXCEPTION, which prevents the request from being sent to the target object. Instead, the ORB returns an exception to the client application.

The sections that immediately follow discuss interceptor security topics and show code fragments of interest from the InterceptorSec target-side interceptor.

<table>
<thead>
<tr>
<th>PrimaryGroupId</th>
<th>Identifies the client name of the user of the client application. In this interceptor, the client name must contain either the character r or a NULL string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessId</td>
<td>Identifies the user of the client application. In this interceptor, the username must have the characters R, P, or N (either upper- or lowercase).</td>
</tr>
</tbody>
</table>
Using the SecurityCurrent Object

Interceptors obtain the SecurityCurrent object from the ORB, not from the Bootstrap object. The SecurityCurrent object available from the ORB has the API that interceptors need for obtaining information about the client.

To obtain the SecurityCurrent object, your interceptors can invoke the `resolve_initial_references("SecurityCurrent")` operation on the ORB. The interceptor can then narrow the SecurityCurrent reference to a `SecurityCurrentLevel1` current.

Obtaining the SecurityCurrent Object

The SecurityCurrent object is available only through the ORB, and this object’s primary functionality is to provide CORBA server applications access to attributes related to the client invocation.

The ORB’s `resolve_initial_references("SecurityCurrent")` method provides the interceptor a reference to a SecurityCurrent object from which the interceptor is provided with Level 1 Security functionality. The interceptor can obtain the attributes of the client invocation via the `get_attributes` method on the SecurityCurrent object, which returns an attribute list to the interceptor. The attribute list contains the attributes that pertain to the user of the client application that performed the invocation being intercepted. The behavior of any and all methods from the CORBA security service is still the same, with the exceptions noted above.

The following C++ code fragment shows obtaining the SecurityCurrent object.

```cpp
try {
    sec_current = m_orb->resolve_initial_references("SecurityCurrent");
} catch (...) {
    *m_outfile <<
        "ERROR: ORB::resolve_initial_references threw exception"
        << endl << endl << flush;
    excep_val = new CORBA::UNKNOWN();
    return Interceptors::REPLY_EXCEPTION;
} if (CORBA::is_nil(sec_current.in())) {
    *m_outfile << "ERROR: No SecurityCurrent present"
        << endl << endl << flush;
    excep_val = new CORBA::NO_PERMISSION();
```
return Interceptors::REPLY_EXCEPTION;
}

m_security_current = SecurityLevel1::Current::_narrow(sec_current.in());
if (!m_security_current)
{
    *m_outfile << "ERROR: Couldn't narrow security current to SecurityLevel1::Current"
    << endl << endl << flush;
    excep_val = new CORBA::NO_PERMISSION();
    return Interceptors::REPLY_EXCEPTION;
}

Creating the List of User Attributes

The code fragments in this section show how the InterceptorSec target-side interceptor creates a list of user attributes and then walks through this list to determine if the user matches the authorization criteria.

In the InterceptorSec sample, creating the list of attributes, then walking through them are done in separate steps. Note that if you specify a client attribute list length of zero (0) to be returned, the SecurityCurrent object returns all the attributes for the client.

    // Get the attributes that correspond to the information that we need to
    // do an authorization check:
    // PrimaryGroupId (clientname of the logged in client)
    // AccessId (username of the logged in client)
    Security::AttributeList_var client_attr = 0;
    try
    {
        client_attr =
        m_security_current->get_attributes("m_attributes_to_get");
    }
    catch (CORBA::NO_PERMISSION) {};

    The following fragment shows creating the list:

    Security::AttributeTypeList_var attr = new Security::AttributeTypeList(2);
    if (!attr.ptr())
    {
        cout <<
            "ERROR: can't allocation security list: Out of memory"
            << endl << endl << flush;
        return;
    }
    attr.inout().length(2);
    attr[(CORBA::ULong)0].attribute_family.family_definer = 0;
    attr[(CORBA::ULong)0].attribute_family.family = 1;
    attr[(CORBA::ULong)0].attribute_type = Security::PrimaryGroupId;
    attr[(CORBA::ULong)1].attribute_family.family_definer = 0;
attr[(CORBA::ULong)1].attribute_family.family = 1;
attr[(CORBA::ULong)1].attribute_type = Security::AccessId;
m_attributes_to_get = attr._retn();
return;

The following fragment shows walking through the attribute list to check whether the user matches the authorization criteria:

if  (client_attr[i].attribute_type.attribute_type == Security::PrimaryGroupId) 
{
    //
    // This attribute is the client name.
    // Compare to some client name value.
    // For this example, we're going to accept anything with
    // an 'r' in it, or a NULL string. You will want to compare
    // the client name to some set of values you have authorized.
    //
    if  ((strlen(value_buffer) == 0) ||
     (strchr(value_buffer, 'r') != 0))
    {
        *m_outfile << "    INFO: Valid client name found: "
                     << value_buffer << endl;
        clientname_ok = 1;
    }
    else
    {
        *m_outfile << "    ERROR: Invalid client name found: "
                     << value_buffer << endl;
    }
}
else if (client_attr[i].attribute_type.attribute_type == Security::AccessId) 
{
    // This attribute is the user name. We're arbitrarily
    // choosing to authorize anyone who has an 'r', 'n', or 'p'
    // in their user id. You will likely want to choose
    // some other criteria for authorization.
    //
    if  ((strchr(value_buffer, 'r') != 0) ||
     (strchr(value_buffer, 'R') != 0) ||
     (strchr(value_buffer, 'p') != 0) ||
     (strchr(value_buffer, 'P') != 0) ||
     (strchr(value_buffer, 'n') != 0) ||
     (strchr(value_buffer, 'N') != 0))
    {
        *m_outfile << "    INFO: Valid username found: "
                     << value_buffer << endl;
        username_ok = 1;
    }
}
Registering and Running the PersonQuery Interceptors

When you run the makefile that builds the PersonQuery sample application in Chapter 4, “PersonQuery Sample Application,” the entire set of sample interceptors are built as well, including the InterceptorSec interceptor. This section describes how to register the InterceptorSec interceptor so that it works with PersonQuery application at run time.

To register and run the InterceptorSec client and server interceptors:

1. Change directory to the InterceptorSec sample directory, where workdirectory represents the name of the directory into which you copied the interceptor sample applications in Chapter, “PersonQuery Sample Application:”
   
   Windows 2003
   > cd <workdirectory>\cxx\security_cxx
   
   UNIX
   $ cd <workdirectory>/cxx/security_cxx

2. Register the interceptor:
   
   Windows 2003
   > nmake -f makefile.nt config
   
   UNIX
   $ make -f makefile.mk config

3. Boot the CORBA server and run the CORBA client:
   
   Windows 2003
   > cd <workdirectory>\cxx\app_cxx
   > tmboot -y
   > PersonQueryClient
   
   UNIX
   > cd <workdirectory>/cxx/app_cxx
   > tmboot -y
   > PersonQueryClient

4. Perform any number of invocations using the PersonQuery client application, using the command syntax described in Chapter 4, “PersonQuery Sample Application.”
5. Stop the PersonQuery application:
   \[ \texttt{tmshutdown -y} \]

**Examining the Interceptor Output**

The InterceptorSec client and target interceptors log their output to the files named, respectively, `InterceptorSecClientxxx.out` and `InterceptorSecTargetxxx.out`. These files contain debugging output from the interceptors that is automatically loaded and executed by the ORB for the PersonQuery application.

**Unregistering the Interceptors**

After you have run the PersonQuery sample application with the InterceptorSec sample interceptors, you can unregister those interceptors using the following steps:

1. Shut down all running CORBA applications by entering the following command:
   \[ \texttt{tmshutdown -y} \]

2. Change directory to the InterceptorSec sample directory, where `workdirectory` represents the name of the directory into which you copied the interceptor sample applications in Chapter, “PersonQuery Sample Application:”
   
   **Windows 2003**
   \[ \texttt{cd <workdirectory>\cxx\security_cxx} \]
   
   **UNIX**
   \[ \texttt{$ cd <workdirectory>/cxx/security_cxx} \]

3. Unregister the interceptors:
   
   **Windows 2003**
   \[ \texttt{rmake -f makefile.nt unconfig} \]
   
   **UNIX**
   \[ \texttt{$ make -f makefile.mk unconfig} \]
InterceptorData Sample Interceptors

This chapter describes the following two sample interceptors that are designed to be used with the PersonQuery sample application:

- **InterceptorDataClient Interceptor**, which is installed on the machine hosting the PersonQuery client component.
- **InterceptorDataTarget Interceptor**, which is installed on the machine hosting the PersonQuery server component.

This chapter explains how each interceptor works, then shows how to build and run them with the PersonQuery sample application.

**InterceptorDataClient Interceptor**

The InterceptorDataClient interceptor intercepts and logs each client application request and reply parameters. This interceptor also allows certain operations on the PersonQuery server application to be invoked by users of the client application who meet specific criteria. The InterceptorDataClient interceptor implements the `InterceptorDataClient` interface, which inherits from the `ClientRequestInterceptor` class.

The InterceptorDataClient class implements the methods as follows:

- **id()**
  
  This method returns the string `InterceptorDataClient`.

- **shutdown()**

  This method removes the request from the tracker object.
• exception_occurred()
  When invoked by the ORB, this method removes the request from the tracker object.

• client_invoke()
  This method determines if the interface and operation are “of interest.” If the client request is “of interest,” this method parses the request parameters and outputs the parameters to the log file. If the client request is not “of interest,” this method simply returns.

• client_response()
  This method determines if the interface and operation in the request are “of interest.” If the interface and operation are “of interest,” this method walks through the CORBA DataInputStream parameter to obtain the reply parameters and write them to the log file. If the interface and operation in the request are not “of interest,” this method simply returns.

In addition, the data interceptor provides the InterceptorDataClientInit method to initialize the client interceptor class.

InterceptorDataTarget Interceptor

The InterceptorDataTarget interceptor intercepts and logs request and reply data parameters. This interceptor also removes sensitive data from specific reply parameters by masking the data with x characters. The InterceptorDataTarget interceptor implements the InterceptorDataTarget interface, which inherits from the TargetRequestInterceptor class.

The InterceptorDataTarget class implements the methods as follows:

• id()
  This method returns the string InterceptorDataTarget.

• shutdown()
  This method simply returns.

• exception_occurred()
  This method removes the request from the tracker object.

• target_invoke()
  This method determines if the interface and operation are “of interest.” If so, this method parses the request parameters and outputs that data to the log file. If the interface and operation in the request are not “of interest,” this method simply returns. If the operation in the request is exit, this method returns the status value INVOKE_NO_EXCEPTION.
Implementing the InterceptorData Interceptors

• target_response()

This method determines if the interface and operation are “of interest.” If so, this method
walks through the DataInputStream parameter to obtain the response parameters and
output to the log file. Sensitive data items are substituted in the log. For example, a
person’s social security number will not be output to the log. If the interface and operation
in the request are not “of interest,” this method simply returns.

In addition, the data interceptor provides the InterceptorDataTargetInit method to initialize
the target interceptor class.

Implementing the InterceptorData Interceptors

Information about the code used to implement the InterceptorData interceptors is provided in
Chapter 2, “Developing CORBA Request-Level Interceptors.” Refer to that chapter for
information about how to do the following:

1. “Starting the Implementation File” on page 2-3.
2. “Initializing the Interceptor at Run Time” on page 2-3.
3. “Obtaining the Interface Name from a Request” on page 2-4.
6. “Reading Parameters Out of a Data Input Stream” on page 2-6.

Registering and Running the InterceptorData Interceptors

When you run the makefile that builds the PersonQuery sample application in Chapter 4,
“PersonQuery Sample Application,” the entire set of sample interceptors are built as well,
including the InterceptorData interceptors. This section describes how to register the
InterceptorData interceptor so that it works with PersonQuery application at run time.

To register and run the InterceptorData client and server interceptors:

1. Change directory to the InterceptorData sample directory, where workdirectory represents
the name of the directory into which you copied the interceptor sample applications in
Chapter 4, “PersonQuery Sample Application:”
Windows 2003
> cd <workdirectory>cxx\data_cxx

UNIX
$ cd <workdirectory>/cxx/data_cxx

2. Register the interceptor:

Windows 2003
> nmake -f makefile.nt config

UNIX
$ make -f makefile.mk config

3. Boot the CORBA server and run the CORBA client:

Windows 2003
> cd <workdirectory>cxx\app_cxx
> tmboot -y
> PersonQueryClient

UNIX
> cd <workdirectory>/cxx/app_cxx
> tmboot -y
> PersonQueryClient

4. Perform any number of invocations using the PersonQuery client application, using the
command syntax described in Chapter 4, “PersonQuery Sample Application.”

5. Stop the PersonQuery application:
> tmshutdown -y

Examining the Interceptor Output

The InterceptorData client and target interceptors log each invocation. For each PersonQuery
application session, the client interceptor creates a log file named
InterceptorDataClientxxx.out, and the target interceptor creates a log file named
InterceptorDataTargetxxx.out. This section shows sample log file data for each
interceptor.

Sample Client Interceptor Log Output
Examining the Interceptor Output

InterceptorDataClientInit called
ClientInterceptorData::id called

ClientInterceptorData::client_invoke called
ClientInterceptorData::client_response called
Request Id: 1
  unable to find request for this reply (must not be one we care about)

ClientInterceptorData::client_invoke called
Request Id: 2
Interface: IDL:beasys.com/PersonQuery:1.0
Operation: findPerson
Parameters:
  name: ALISTER LANCASHIRE
  address: 3 PENNY LANE
          LONDON GB UK
  ss: 999-99-9999
  sex: can't tell
  age(yrs.): 85
  marital status: single
  hobby: stamp collecting
  date-of-birth: 11/25/1913
  height(in.): 32
  weight(lbs.): 57
  hair color: unknown
  eye color: blue
  skin color: white
  other markings: missing limb

Sample Target Interceptor Log Output

InterceptorDataTargetInit called
TargetInterceptorData::id called

TargetInterceptorData::target_response called
Request Id: 2
ReplyStatus: GIOP::NO_EXCEPTION
Interface: IDL:beasys.com/PersonQuery:1.0
Operation: findPerson
Method Result: TRUE
Parameters:
  Maximum: 8
  Length: 8

Item 0
  name: ALISTER LANCASHIRE
  address: 3 PENNY LANE
          LONDON GB UK
  ss: NO PRIVILEDGE
sex: NO PRIVILEGE
age (years): NO PRIVILEGE
marital status: NO PRIVILEGE
hobby: stamp collecting
date-of-birth: NO PRIVILEGE
height (in.): 32
weight (lbs.): 57
hair color: unknown
eye color: blue
skin color: NO PRIVILEGE
other markings: missing limb

Unregistering the Interceptors

After you have run the PersonQuery sample application with the InterceptorData sample interceptors, you can unregister those interceptors using the following steps:

1. Shut down all running CORBA applications by entering the following command:
   > tmshutdown -y

2. Change directory to the InterceptorData sample directory, where workdirectory represents the name of the directory into which you copied the interceptor sample applications in Chapter, “PersonQuery Sample Application:”
   Windows 2003
   > cd <workdirectory>\cxx\data_cxx
   UNIX
   $ cd <workdirectory>/cxx/data_cxx

3. Unregister the interceptors:
   Windows 2003
   > nmake -f makefile.nt unconfig
   UNIX
   $ make -f makefile.mk unconfig
This chapter documents the following interfaces that you use to implement request-level interceptors:

- Interceptors::Interceptor
- RequestLevelInterceptor::RequestInterceptor
- RequestLevelInterceptor::ClientRequestInterceptor
- RequestLevelInterceptor::TargetRequestInterceptor
- CORBA::DataInputStream

Each of these interfaces is a **locality-constrained object**. Any attempt to pass a reference outside its locality (that is, its process), or any attempt to externalize an object supporting this interface using the CORBA ORB object_to_string operation, results in the CORBA MARSHAL system exception (CORBA::MARSHAL) being raised.
Interceptor Hierarchy

Request-level interceptors are divided into two interfaces, providing separate client- and
target-side functionality. The following figure illustrates the inheritance hierarchy of the
request-level interceptors supported in the BEA Tuxedo product.

Note on Unused Interfaces

The method signatures for operations on classes derived from the RequestLevelInterceptor
interface include parameters for the following interfaces:

- RequestLevelInterceptor::DataOutputStream
- RequestLevelInterceptor::ServiceContextList
These interfaces are not used in the BEA Tuxedo product. However, they are defined in the BEA Tuxedo product so that you do not need to recompile your CORBA application if an implementation of these interfaces is ever provided in a future release of the BEA Tuxedo product. The ORB always passes a nil for the actual argument. You should not attempt to use this argument; doing so will likely end the process with a serious error.

**Interceptors::Interceptor Interface**

The `Interceptors::Interceptor` interface is defined as the base interface of all types of interceptors, including request-level interceptors. This interface contains the set of operations and attributes that are supported by all types of interceptors. The `Interceptors::Interceptor` interface is defined as an abstract interface; thus an instance of the interface cannot be instantiated.

**Listing 8-1  OMG IDL for the Interceptors::Interceptor Interface**

```idl
// File: Interceptors.idl
#ifndef _INTERCEPTORS_IDL
#define _INTERCEPTORS_IDL

#pragma prefix "beasys.com"

module Interceptors {
    native ExceptionValue;

    local Interceptor {
        readonly attribute string id; // identifier of interceptor

        // called by ORB when interceptor is being shutdown
        ShutdownReturnStatus shutdown(
            in ShutdownReason reason,
            out ExceptionValue excep_val
        ); // locality constrained
    }
}
```

The implementation of the operations _duplicate, _narrow, and _nil are inherited from the implementation of the CORBA::LocalBase interface provided by the CORBA ORB in the BEA Tuxedo product.

Listing 8-2  C++ Declaration of the Interceptors::Interceptor Interface

```cpp
#ifndef _INTERCEPTORS_H
#define _INTERCEPTORS_H

#include <string.h>
#include <CORBA.h>

class OBBEXPDLL Interceptors
{
 public:
 class Interceptor;
 typedef Interceptor * Interceptor_ptr;

 enum InvokeReturnStatus
 {
  INVOKE_NO_EXCEPTION, // proceed normally
  REPLY_NO_EXCEPTION, // stop proceeding; start reply processing
  REPLY_EXCEPTION     // stop proceeding; reply with exception
  };

 enum ResponseReturnStatus
 {
  RESPONSE_NO_EXCEPTION, // proceed normally
  RESPONSE_EXCEPTION
  };

 enum ShutdownReturnStatus
 {
  SHUTDOWN_NO_EXCEPTION,
  SHUTDOWN_EXCEPTION
  };

 enum ShutdownReason
 {
  ORB_SHUTDOWN,
  CONNECTION_ABORTED,
  RESOURCES_EXCEEDED
  };

```
Interceptor::id

Synopsis
Obtains the vendor assigned identity of the interceptor as a string value.

C++ Mapping

```
virtual CORBA::String id() = 0;
```

Parameters
None.

Exceptions
None.
Description
The id accessor operation is used by the ORB to obtain the vendor assigned identity of the interceptor as a string value. This attribute is used primarily for debugging and tracing of operations on the interceptors called by the ORB.

Return Values
This operation returns a pointer to a null-terminated string containing the identity of the interceptor as assigned by the provider of the interceptor implementation.

Interceptor::shutdown
Synopsis
Notifies an implementation of an interceptor that the interceptor is being shut down.

C++ Binding
virtual ShutdownReturnStatus shutdown( ShutdownReason reason,
   CORBA::Exception_ptr & excep_val) = 0;

Parameters
reason
A ShutdownReason value that indicates the reason why the interceptor is being shut down. The following ShutdownReason values can be passed to the operation:

<table>
<thead>
<tr>
<th>Status Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORB_SHUTDOWN</td>
<td>Indicates that the ORB is being shut down.</td>
</tr>
<tr>
<td>RESOURCES_EXCEEDED</td>
<td>Indicates that resources of the process have been exhausted.</td>
</tr>
<tr>
<td>CONNECTION_ABORTED</td>
<td>This exception is not reported in BEA Tuxedo 8.0.</td>
</tr>
</tbody>
</table>

excep_val
A reference to an ExceptionValue in which the operation is to store any exception raised. This parameter is valid only if a value of SHUTDOWN_EXCEPTION is returned from the operation.

ExceptionValue is mapped to the class CORBA::Exception.
Exceptions

None.

Description

The shutdown operation is used by the ORB to notify an implementation of an interceptor that the interceptor is being shut down. The ORB destroys the instance of the interceptor once control is returned from the operation back to the ORB.

Return Values

SHUTDOWN_NO_EXCEPTION
Indicates that the operation has not raised an exception.

SHUTDOWN_EXCEPTION
Indicates that the operation has raised an exception. The value of the exception is stored in the excep_val parameter.
Request-Level Interceptor API

**RequestLevelInterceptor::RequestInterceptor Interface**

The `RequestLevelInterceptor::RequestInterceptor` interface is the base interface of all request-level interceptors. It inherits directly from the `Interceptors::Interceptor` interface.

The `RequestLevelInterceptor::RequestInterceptor` interface:

- Contains the set of operations and attributes that are supported by all request-level interceptors.
- Is defined as an abstract interface; therefore, an instance of the interface cannot be instantiated.

The `local` keyword in OMG IDL indicates that the `RequestInterceptor` interface is not a normal CORBA object, so it cannot be used as such.

**Listing 8-3 OMG IDL for the RequestLevelInterceptor::RequestInterceptor Interface**

```idl
#ifndef _REQUEST_LEVEL_INTERCEPTOR_IDL
#define _REQUEST_LEVEL_INTERCEPTOR_IDL

#include <orb.idl>
#include <Giop.idl>
#include <Interceptors.idl>

#pragma prefix "beasys.com"

module RequestLevelInterceptor
{
    local RequestInterceptor : Interceptors::Interceptor
    {
        void exception_occurred(
            in ReplyContext reply_context,
            in ExceptionValue excep_val
        );
    };
}
#endif /* _REQUEST_LEVEL_INTERCEPTOR_IDL */
```

Using CORBA Request-Level Interceptors
The implementation of the RequestInterceptor interface inherits from CORBA::LocalBase rather than from CORBA::Object. CORBA::LocalBase provides an implementation of the operations _duplicate, _narrow, and _nil, similar to those of CORBA::Object.

Listing 8-4   C++ Declaration for the RequestInterceptor Interface

```cpp
#ifndef _RequestLevelInterceptor_h
#define _RequestLevelInterceptor_h

#include <CORBA.h>
#include <IOP.h>
#include <GIOP.h>
#include <Interceptors.h>

class OBBEXPDLL RequestLevelInterceptor
{
  public:
    class   RequestInterceptor;
    typedef RequestInterceptor *   RequestInterceptor_ptr;

    struct  RequestContext
    {
      Interceptors::Version struct_version;
      CORBA::ULong request_id;
      CORBA::Octet response_flags;
      GIOP::TargetAddress target;
      CORBA::String_var interface_id;
      CORBA::String_var operation;
      RequestContext &operator=(const RequestContext &_obj);
    };

    typedef RequestContext *   RequestContext_ptr;
    typedef GIOP::ReplyStatusType_1_2 ReplyStatus;

    struct  ReplyContext
    {
      Interceptors::Version struct_version;
      CORBA::ULong request_id;
      ReplyStatus reply_status;
    };

    typedef ReplyContext *   ReplyContext_ptr;

    class OBBEXPDLL RequestInterceptor :
    {
      public virtual  Interceptors::Interceptor
    };
```

Using CORBA Request-Level Interceptors 8-9
Request-Level Interceptor API

```
public:
  static RequestInterceptor_ptr
    _duplicate(RequestInterceptor_ptr obj);
  static RequestInterceptor_ptr
    _narrow(RequestInterceptor_ptr obj);
  inline static RequestInterceptor_ptr _nil() { return 0; }

  virtual void
    exception_occurred( const ReplyContext & reply_context,
                          CORBA::Exception_ptr excep_val) = 0;

protected:
  RequestInterceptor(CORBA::LocalBase_ptr obj = 0) { }
  virtual ~RequestInterceptor(){ }

private:
  RequestInterceptor( const RequestInterceptor&) { }
  void operator=(const RequestInterceptor&) { }
};  // class RequestInterceptor
```

RequestContext Structure

Synopsis
Contains the information that represents the context in which a request is to be processed.

C++ Binding
```
struct RequestContext
{
  Interceptors::Version struct_version;
  CORBA::ULong request_id;
  CORBA::Octet response_flags;
  GIOP::TargetAddress target;
  CORBA::String_var interface_id;
  CORBA::String_var operation;
  RequestContext &operator=(const RequestContext &_obj);
};
```
Members

```c
struct_version
```

An indication of the version of the RequestContext that provides an indication of the format and members. The version information is separated into the following two pieces:

<table>
<thead>
<tr>
<th>Version Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>major_version</td>
<td>Indicates the major version value. The value of this member is incremented anytime a change is made to the contents or layout of a RequestContext that is not backward compatible with previous versions.</td>
</tr>
<tr>
<td>minor_version</td>
<td>Indicates the minor version value. The value of this member is incremented anytime a change is made to the contents or layout of a RequestContext that is backward compatible with previous versions.</td>
</tr>
</tbody>
</table>

```c
request_id
```

An unsigned long value that specifies the identifier assigned to a request by the initiating ORB.

```c
response_flags
```

The lowest order bit of response_flags is set to 1 if a reply message is expected for this request. If the operation is not defined as oneway, and the request is not invoked via the DII with the INV_NO_RESPONSE flag set, response_flags will be set to \x03. If the operation is defined as oneway, or the request is invoked via the DII with the INV_NO_RESPONSE flag set, response_flags may be set to \x00 or \x01. When this flag is set to \x01 for a oneway operation, receipt of a reply does not imply that the operation has necessarily completed.

```c
target
```

A discriminated union that identifies the object that is the target of the invocation. The discriminator indicates the format in which the target addressing is presented. The possible discriminator values are:
The **RequestContext** data object contains the information that represents the context in which a request is to be processed. The context information contained in the **RequestContext** provides information necessary to coordinate between the processing of a given request and its corresponding reply.

The context information in the **RequestContext** structure cannot be modified by the interceptor implementation. The ORB maintains ownership of the **RequestContext** and is responsible for freeing any resources associated with the **RequestContext** when it has completed using it.
ReplyContext Structure

Synopsis

Contains the information that represents the context in which a reply is to be processed.

C++ Binding

```cpp
struct ReplyContext {
    Interceptors::Version struct_version;
    CORBA::ULong request_id;
    ReplyStatus reply_status;
};
```

Members

- **struct_version**
  
  An indication of the version of the ReplyContext that provides an indication of the format and members. The version information is separated into the following two pieces:

<table>
<thead>
<tr>
<th>Version Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>major_version</td>
<td>Indicates the major version value. The value of this member is incremented anytime a change is made to the contents or layout of a ReplyContext that is not backward compatible with previous versions.</td>
</tr>
<tr>
<td>minor_version</td>
<td>Indicates the minor version value. The value of this member is incremented anytime a change is made to the contents or layout of a ReplyContext that is backward compatible with previous versions.</td>
</tr>
</tbody>
</table>

- **request_id**

  An unsigned long value that specifies the identifier assigned to a request by the initiating ORB.

- **reply_status**

  Indicates the completion status of the associated request, and also determines part of the reply body contents.
## Status Value Description

<table>
<thead>
<tr>
<th>Status Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_EXCEPTION</td>
<td>Indicates that the requested operation completed successfully and that the value of the arg_stream parameter contains the return values of the operation.</td>
</tr>
<tr>
<td>USER_EXCEPTION</td>
<td>Indicates that the requested operation failed because of an exception reported by the target object.</td>
</tr>
<tr>
<td>SYSTEM_EXCEPTION</td>
<td>Indicates that the request operation failed because of an exception reported either by the target object or by the infrastructure.</td>
</tr>
<tr>
<td>LOCATION_FORWARD</td>
<td>Indicates that the body contains an object reference (IOR). The client ORB is responsible for resending the original request to that (different) object. This resending is transparent to the client program making the request, but the resending is not transparent to the interceptor.</td>
</tr>
<tr>
<td>LOCATION_FORWARD_PERM</td>
<td>Indicates that the body contains an object reference. The usage is similar to LOCATION_FORWARD, but when used by a server, this value also provides an indication to the client that the client may replace the old IOR with the new IOR. Both the old IOR and the new IOR are valid, but the new IOR is preferred for future use. This resending is transparent to the client program making the request, but the resending is not transparent to the interceptor.</td>
</tr>
<tr>
<td>NEEDS_ADDRESSING_MODE</td>
<td>Indicates that the body contains a GIOP::AddressingDisposition. The client ORB is responsible for resending the original request using the requested addressing mode. This resending is transparent to the client program making the request, but the resending is not transparent to the interceptor.</td>
</tr>
</tbody>
</table>
Description

The ReplyContext data object contains the information that represents the context in which a reply is to be processed. The context information contained in ReplyContext provides information necessary to coordinate between the processing of a given request and its corresponding reply.

The context information in ReplyContext cannot be modified by the interceptor implementation. The ORB maintains ownership of ReplyContext and is responsible for freeing any resources associated with ReplyContext when it has completed using it.

RequestInterceptor::exception_occurred

Synopsis

Is called by the ORB to allow the interceptor to clean up any state that the interceptor might have been managing that is specific to a request.

C++ Binding

```cpp
virtual void
exception_occurred( const ReplyContext & reply_context,
                     CORBA::Exception_ptr excep_val) = 0;
```

Parameters

- `reply_context` A reference to a ReplyContext that contains information about the context in which the reply is being performed.
- `excep_val` A pointer to the exception reported by the ORB or by another interceptor.

Exceptions

None.

Description

The exception_occurred operation is called on a request-level interceptor implementation in one of three cases:

1. Another interceptor sets an exception (rather than an exception being generated by the ORB or the method).
2. The ORB detects an operating system or communication-related problem.

3. A client deletes a Request object that was used to initiate a deferred synchronous DII. The exception_occurred method is called instead of the client_response or target_response method of that interceptor. The ORB calls the exception_occurred method to allow the interceptor implementation to clean up any state that it might have been managing that is specific to a request.

Return Values

None.
RequestLevelInterceptor:: ClientRequestInterceptor Interface

This is the base interface of all request-level interceptors. It inherits directly from the RequestLevelInterceptor::RequestInterceptor interface. The interface contains the set of operations and attributes that are supported by all client-side request-level interceptors.

Listing 8-5  OMG IDL Definition

//File: RequestLevelInterceptor.idl
#ifndef _REQUEST_LEVEL_INTERCEPTOR_IDL
#define _REQUEST_LEVEL_INTERCEPTOR_IDL
#include <orb.idl>
#include <Giop.idl>
#include <Interceptors.idl>
#pragma prefix "beasys.com"
module RequestLevelInterceptor
{
  local ClientRequestInterceptor : RequestInterceptor
  {
    InvokeReturnStatus
      client_invoke(
        in    RequestContext           request_context,
        in    ServiceContextList       service_context,
        in    CORBA::DataInputStream   request_arg_stream,
        in    CORBA::DataOutputStream  reply_arg_stream,
        out   ExceptionValue           excep_val
      );
    
    ResponseReturnStatus
      client_response(
        in    ReplyContext           reply_context,
        in    ServiceContextList     service_context,
        in    CORBA::DataInputStream arg_stream,
        out   ExceptionValue         excep_val
      );
  }
};
#endif /* _REQUEST_LEVEL_INTERCEPTOR_IDL */
The implementation of the operations _duplicate, _narrow, and _nil are inherited indirectly from the implementation of the CORBA::LocalBase interface provided by the CORBA ORB in the BEA Tuxedo product.

Listing 8-6  C++ Declaration

```cpp
#ifndef _RequestLevelInterceptor_h
#define _RequestLevelInterceptor_h

#include <CORBA.h>
#include <IOP.h>
#include <GIOP.h>
#include <Interceptors.h>

class OBBEXPDLL RequestLevelInterceptor
{
    public:
        class ClientRequestInterceptor;
        typedef ClientRequestInterceptor * ClientRequestInterceptor_ptr;

    class OBBEXPDLL ClientRequestInterceptor : public virtual RequestInterceptor
    {
        public:
            static ClientRequestInterceptor_ptr _duplicate(ClientRequestInterceptor_ptr obj);
            static ClientRequestInterceptor_ptr _narrow(ClientRequestInterceptor_ptr obj);
            inline static ClientRequestInterceptor_ptr _nil() { return 0; }

        virtual Interceptors::InvokeReturnStatus client_invoke(
            const RequestContext & request_context,
            ServiceContextList_ptr service_context,
            CORBA::DataInputStream_ptr request_arg_stream,
            CORBA::DataOutputStream_ptr reply_arg_stream,
            CORBA::Exception_ptr & excep_val ) = 0;

        virtual Interceptors::ResponseReturnStatus client_response(
            const ReplyContext & reply_context,
            ServiceContextList_ptr service_context,
            CORBA::DataInputStream_ptr arg_stream,
            CORBA::Exception_ptr & excep_val ) = 0;
    }
}
```

ClientRequestInterceptor::client_invoke

protected:
ClientRequestInterceptor(CORBA::LocalBase_ptr obj = 0) { }
virtual ~ClientRequestInterceptor(){ }

private:
ClientRequestInterceptor( const ClientRequestInterceptor&)
{ }
void operator=(const ClientRequestInterceptor&) { }
};  // class ClientRequestInterceptor

#endif /* _RequestLevelInterceptor_h */

ClientRequestInterceptor::client_invoke

Synopsis
Is called by the client-side ORB anytime the client application sends an invocation to a target object.

C++ Binding

virtual Interceptors::InvokeReturnStatus
client_invoke(
const RequestContext & request_context,
ServiceContextList_ptr service_context,
CORBA::DataInputStream_ptr request_arg_stream,
CORBA::DataOutputStream_ptr reply_arg_stream,
CORBA::Exception_ptr & excep_val = 0;

Parameters

request_context
A reference to a RequestContext that contains information about the context in which the request is being performed.

service_context
A pointer to a ServiceContextList containing service context information to be sent as part of the request to the target object.

Note: In BEA Tuxedo 8.0, the value of this parameter is always a nil object.
Request-Level Interceptor API

request_arg_stream
A pointer to a DataInputStream that can be used by the interceptor implementation to retrieve the value of the parameter of the operation.

The DataInputStream contains all in and inout parameters, in the order in which they are specified in the operation’s IDL definition, from left to right. A nil DataInputStream indicates that no arguments exist.

reply_arg_stream
A pointer to a CORBA::DataOutputStream that can be used to populate the parameters to be returned to the initiator of the invocation as a reply. The use of this parameter is only valid if a status of REPLY_NO_EXCEPTION is returned.

Note: In BEA Tuxedo 8.0, the value of this parameter is always a nil object.

excep_val
A reference to a location in which the interceptor can return an exception in order to report an error. The use of this parameter is only valid if a status of REPLY_EXCEPTION is returned. Note that the ORB is responsible for the memory management for the excep_val parameter.

Exceptions
None.

Description
The client_invoke operation is called on an interceptor implementation that supports the RequestLevelInterceptor::ClientRequestInterceptor interceptor interface. The operation is called by the ORB anytime that an invocation is being sent to a target object, regardless of whether the target object is in a different address space or the same address space as the target object.

Return Values

INVOKE_NO_EXCEPTION
Indicates that the interceptor successfully performed any processing required and that the ORB should continue processing the invocation in order to deliver it to the target object.

REPLY_NO_EXCEPTION
Indicates that the interceptor successfully performed any processing required to totally satisfy the request. The ORB should consider the request completed and begins processing any information in the reply_arg_stream, if any, as the return parameter values for the request.

Note: In BEA Tuxedo 8.0, an interceptor cannot return this status value.
ClientRequestInterceptor::client_response

REPLY_EXCEPTION
Indicates that the interceptor encountered an error that should result in the discontinued processing of the request toward the target. The parameter `excep_val` is used to report the exception to the ORB. The ORB calls interceptors on the way back to the client application with the `exception_occurred` operation rather than with the `client_response` operation. Note that the ORB is responsible for the memory management for the `excep_val` parameter.

ClientRequestInterceptor::client_response

Synopsis
Is called on an interceptor implementation that supports the `RequestLevelInterceptor::ClientRequestInterceptor` interface.

C++ Binding

```cpp
virtual Interceptors::ResponseReturnStatus
client_response(
    const ReplyContext & reply_context,
    ServiceContextList_ptr service_context,
    CORBA::DataInputStream_ptr arg_stream,
    CORBA::Exception_ptr & excep_val ) = 0;
```

Parameters

- `reply_context`
A reference to a `ReplyContext` that contains information about the context in which the reply is being performed.

- `service_context`
A pointer to a `ServiceContextList` containing service context information received as a result of processing the request by the target object.
**Note:** In BEA Tuxedo 8.0, the value of this parameter is always a nil object.

- `arg_stream`
A pointer to a `DataInputStream` that can be used by the interceptor implementation to retrieve the value of the reply parameters of the operation.

The following table identifies what the `client_response` method returns in the `DataInputStream` object based on the status contained in the `ReplyContext` object:
Request-Level Interceptor API

<table>
<thead>
<tr>
<th>Status Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATION_FORWARD,</td>
<td>A nil DataInputStream is supplied.</td>
</tr>
<tr>
<td>LOCATION_FORWARD_PERM,</td>
<td></td>
</tr>
<tr>
<td>or NEEDS_ADDRESSING_MODE</td>
<td></td>
</tr>
<tr>
<td>NO_EXCEPTION</td>
<td>The DataInputStream contains first any operation return value, then any inout and out parameters in the order in which they appear in the operation’s IDL definition, from left to right. A nil DataInputStream indicates that no arguments exist.</td>
</tr>
<tr>
<td>USER_EXCEPTION or SYSTEM_EXCEPTION</td>
<td>The DataInputStream contains the exception that was raised by the operation.</td>
</tr>
</tbody>
</table>

**Note:** Exceptions contain a string followed by any exception members. The string contains the repository ID for the exception. The exception members are passed in the same manner as a struct. A system exception contains two unsigned long members, a minor code, and a completion status.

**excep_val**

A reference to a location in which the interceptor can return an exception in order to report an error. The use of this parameter is only valid if a status of REPLY_EXCEPTION is returned. Note that the ORB is responsible for the memory management for the excep_val parameter.

**Exceptions**

None.

**Description**

The client_response operation is called on an interceptor implementation that supports the RequestLevelInterceptor::ClientRequestInterceptor interface. The operation is called by the ORB anytime that a reply to an invocation is being received by the initiator of the request, regardless of whether the initiator is in a different address space or the same address space as the target object.
Return Values

RESPONSE_NO_EXCEPTION
Indicates that the interceptor successfully performed any processing required and that the ORB should continue processing the reply to the request to deliver it to the initiator of the request.

RESPONSE_EXCEPTION
Indicates that the interceptor encountered an error. The parameter `excep_val` is used to report the exception to the ORB. Any interceptors not yet called on the way back to the client have their `exception_occurred` operation called by the ORB to notify them that processing the request has failed.
RequestLevelInterceptor::
TargetRequestInterceptor Interface

This is the base interface of all request-level interceptors. It inherits directly from the RequestLevelInterceptor::RequestInterceptor interface. The interface contains the set of operations and attributes that are supported by all target-side request-level interceptors.

Listing 8-7  OMG IDL Definition

//File: RequestLevelInterceptor.idl

#ifndef _REQUEST_LEVEL_INTERCEPTOR_IDL
#define _REQUEST_LEVEL_INTERCEPTOR_IDL

#include <orb.idl>
#include <Giop.idl>
#include <Interceptors.idl>

#pragma prefix "beasys.com"

module RequestLevelInterceptor
{
  local TargetRequestInterceptor : RequestInterceptor
  {
    InvokeReturnStatus
    target_invoke(
      in RequestContext request_context,
      in ServiceContextList service_context,
      in CORBA::DataInputStream request_arg_stream,
      in CORBA::DataOutputStream reply_arg_stream,
      out ExceptionValue excep_val
    );

    ResponseReturnStatus
    target_response(
      in ReplyContext reply_context,
      in ServiceContextList service_context,
    );
  }
};
The implementation of the operations _duplicate, _narrow, and _nil are inherited indirectly from the implementation of the CORBA::LocalBase interface provided by the CORBA ORB in the BEA Tuxedo product.

Listing 8-8  C++ Declaration

```c++
#include <CORBA.h>
#include <IOP.h>
#include <GIOP.h>
#include <Interceptors.h>

class OBBEXPDLL RequestLevelInterceptor
{
public:
    class TargetRequestInterceptor;
    typedef TargetRequestInterceptor * TargetRequestInterceptor_ptr;

class OBBEXPDLL TargetRequestInterceptor : public virtual RequestInterceptor
{
    public:
        static TargetRequestInterceptor_ptr _duplicate(TargetRequestInterceptor_ptr obj);
        static TargetRequestInterceptor_ptr _narrow(TargetRequestInterceptor_ptr obj);
        inline static TargetRequestInterceptor_ptr _nil() { return 0; }

        virtual Interceptors::InvokeReturnStatus target_invoke(
            const RequestContext & request_context,
            ServiceContextList_ptr service_context,
            int arg, CORBA::String_var arg_str,
            CORBA::DataInputStream arg_stream,
            ExceptionValue excep_val
        );

        #endif /* _REQUEST_LEVEL_INTERCEPTOR_IDL */
```
Request-Level Interceptor API

```cpp
CORBA::DataInputStream_ptr request_arg_stream,
CORBA::DataOutputStream_ptr reply_arg_stream,
CORBA::Exception_ptr & excep_val ) = 0;

virtual Interceptors::ResponseReturnStatus
target_response(
    const ReplyContext & reply_context,
    ServiceContextList_ptr service_context,
    CORBA::DataInputStream_ptr arg_stream,
    CORBA::Exception_ptr & excep_val ) = 0;

protected:
    TargetRequestInterceptor(CORBA::LocalBase_ptr obj = 0) { }
    virtual ~TargetRequestInterceptor(){ }

private:
    TargetRequestInterceptor( const TargetRequestInterceptor&)
    { }        // TargetRequestInterceptor
    virtual operator=(const TargetRequestInterceptor&) { }  // TargetRequestInterceptor
};  // class TargetRequestInterceptor
#endif /* _RequestLevelInterceptor_h */
```

**TargetRequestInterceptor::target_invoke**

**Synopsis**

Is called by the target-side ORB anytime an invocation is being received by a target object.

**C++ Binding**

```cpp
virtual Interceptors::InvokeReturnStatus
target_invoke(
    const RequestContext & request_context,
    ServiceContextList_ptr service_context,
    CORBA::DataInputStream_ptr request_arg_stream,
    CORBA::DataOutputStream_ptr reply_arg_stream,
    CORBA::Exception_ptr & excep_val ) = 0;
```

**Parameters**

- `request_context`
  - A reference to a `RequestContext` that contains information about the context in which the request is being performed.
TargetRequestInterceptor::target_invoke

service_context
A pointer to a ServiceContextList containing service context information received as part of the request to the target object. In BEA Tuxedo 8.0, the value of this parameter is always a nil object.

request_arg_stream
A pointer to a DataInputStream that can be used by the interceptor implementation to retrieve the value of the parameter of the operation. The DataInputStream contains all in and inout parameters, in the order in which they are specified in the operation’s IDL definition, from left to right. A nil DataInputStream indicates that no arguments exist.

reply_arg_stream
A pointer to a DataOutputStream that can be used to populate the parameters to be returned to the initiator of the invocation as a reply. The use of this parameter is only valid if a status of REPLY_NO_EXCEPTION is returned. In BEA Tuxedo 8.0, the value of this parameter is always a nil object.

excep_val
A reference to a location in which the interceptor can return an exception in order to report an error. The use of this parameter is only valid if a status of REPLY_EXCEPTION is returned. Note that the ORB is responsible for the memory management for the excep_val parameter.

Exceptions
None.

Description
The target_invoke operation is called on an interceptor implementation that supports the RequestLevelInterceptor::TargetRequestInterceptor interface. The operation is called by the ORB anytime that an invocation is being received by a target object, regardless of whether the target object is in a different address space or the same address space as the target object.

Return Values

INVOKE_NO_EXCEPTION
Indicates that the interceptor successfully performed any processing required and that the ORB should continue processing the invocation in order to deliver it to the target object.

REPLY_NO_EXCEPTION
Indicates that the interceptor successfully performed any processing required to totally satisfy the request. The ORB should consider the request completed and begins processing
any information in the \texttt{reply\_arg\_stream}, if any, as the return parameter values for the request.

\textbf{Note:} In BEA Tuxedo 8.0, an interceptor cannot return this status value.

\textbf{REPLY\_EXCEPTION}
Indicates that the interceptor encountered an error that should result in the discontinued processing of the request in order to deliver it to the target object. The parameter \texttt{excep\_val} is used to report the exception to the ORB. The ORB calls interceptors on the way back to the client with the \texttt{exception\_occurred} operation, rather than with the \texttt{target\_response} operation. Note that the ORB is responsible for the memory management for the \texttt{excep\_val} parameter.

\textbf{TargetRequestInterceptor::target\_response}

\textbf{Synopsis}
Is called by the target-side ORB anytime that a reply to an invocation is being sent to the initiator of the request.

\textbf{C++ Binding}

\begin{verbatim}
virtual Interceptors::ResponseReturnStatus
target_response(
   const ReplyContext & reply_context,
   ServiceContextList_ptr service_context,
   CORBA::DataInputStream_ptr arg_stream,
   CORBA::Exception_ptr & excep_val ) = 0;
\end{verbatim}

\textbf{Parameters}

\begin{itemize}
\item \texttt{reply\_context}
A reference to a \texttt{ReplyContext} that contains information about the context in which the reply is being performed.

\item \texttt{service\_context}
A pointer to a \texttt{ServiceContextList} containing service context information to be sent as a result of processing the request by the target object.
\textbf{Note:} In BEA Tuxedo 8.0, the value of this parameter is always a nil object.

\item \texttt{arg\_stream}
A pointer to a \texttt{DataInputStream} that can be used by the interceptor implementation to retrieve the value of the reply parameters of the operation.
\end{itemize}
The following table identifies what the target_response method returns in the DataInputStream object based on the status contained in the ReplyContext object:

<table>
<thead>
<tr>
<th>Status Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATION_FORWARD,</td>
<td>A nil DataInputStream is supplied</td>
</tr>
<tr>
<td>LOCATION_FORWARD_PERM,</td>
<td></td>
</tr>
<tr>
<td>or NEEDS_ADDRESSING_MODE</td>
<td></td>
</tr>
<tr>
<td>NO_EXCEPTION</td>
<td>The DataInputStream contains first any operation return value, then any</td>
</tr>
<tr>
<td></td>
<td>inout and out parameters in the order in which they appear in the operation's</td>
</tr>
<tr>
<td></td>
<td>IDL definition, from left to right. A nil DataInputStream indicates that</td>
</tr>
<tr>
<td></td>
<td>no arguments exist.</td>
</tr>
<tr>
<td>USER_EXCEPTION or SYSTEM_EXCEPTION</td>
<td>The DataInputStream contains the exception that was raised by the operation.</td>
</tr>
</tbody>
</table>

**Note:** Exceptions contain a string followed by any exception members. The string contains the repository ID for the exception. The exception members are passed in the same manner as a struct. A system exception contains two unsigned long members, a minor code, and a completion status.

```markdown
A reference to a location in which the interceptor can return an exception in order to report an error. The use of this parameter is valid only if a status of REPLY_EXCEPTION is returned. Note that the ORB is responsible for the memory management for the excep_val parameter.
```

**Exceptions**

None.

**Description**

The target_response operation is called on an interceptor implementation that supports the RequestLevelInterceptor::TargetRequestInterceptor interface. The operation is called by the target-side ORB anytime that a reply to an invocation is being sent to the initiator of the request, regardless of whether the initiator is in a different address space or the same address space as the target object.
Request-Level Interceptor API

Return Values

RESPONSE_NO_EXCEPTION
Indicates that the interceptor successfully performed any processing required and that the ORB should continue processing the reply to the request to deliver it to the initiator of the request.

RESPONSE_EXCEPTION
Indicates that the interceptor encountered an error. The parameter excep_val is used to report the exception to the ORB. Any interceptors not yet called on the way back to the client have their exception_occurred operation called by the ORB in order to notify them that processing the request has failed. Note that the ORB is responsible for the memory management for the excep_val parameter.

AppRequestInterceptorInit

Synopsis
Instantiates and initializes client-side and target-side interceptors.

C++ Binding

```c
typedef void (*AppRequestInterceptorInit)(
    CORBA::ORB_ptr TheORB,
    RequestLevelInterceptor::ClientRequestInterceptor ** ClientPtr,
    RequestLevelInterceptor::TargetRequestInterceptor ** TargetPtr,
    CORBA::Boolean * RetStatus);
```

Parameters

TheORB
A pointer to the ORB object with which the implementation of the interceptors are associated.

ClientPtr
A pointer in which to return a pointer to the instance of the RequestLevelInterceptor::ClientRequestInterceptor that was instantiated for use by the ORB.

TargetPtr
A pointer in which to return a pointer to the instance of the RequestLevelInterceptor::TargetRequestInterceptor that was instantiated for use by the ORB.
RetStatus
A pointer to a location into which the interceptor implementation indicates whether the instantiation and initialization of the interceptor was successful. A value of `CORBA::TRUE` is used to indicate that instantiation and initialization of the interceptors was successful. A value of `CORBA::FALSE` is used to indicate that the instantiation and initialization of the interceptors was unsuccessful for some reason.

Exceptions
None.

Description
The `AppRequestInterceptorInit` function is a user-provided function that is used by the ORB to instantiate and initialize client-side and target-side interceptors.

Return Values
None.
CORBA::DataInputStream Interface

The abstract valuetype keywords in IDL applied to DataInputStream indicates that it is not the same as an interface.

Listing 8-9  OMG IDL Definition

module CORBA {  
//... all the rest

// Definitions used by DataInputStream
typedef sequence<any>            AnySeq;
typedef sequence<boolean>        BooleanSeq;
typedef sequence<char>           CharSeq;
typedef sequence<octet>          OctetSeq;
typedef sequence<short>          ShortSeq;
typedef sequence<unsigned short> UShortSeq;
typedef sequence<long>           LongSeq;
typedef sequence<unsigned long> ULongSeq;
typedef sequence<float>          FloatSeq;
typedef sequence<double>         DoubleSeq;

// DataInputStream - for reading data from the stream
abstract valuetype DataInputStream  
{
    any                         read_any();  // Raises NO_IMPLEMENT
    boolean                    read_boolean();
    char                       read_char();
    octet                      read_octet();
    short                      read_short();
    unsigned short            read_ushort();
    long                       read_long();
    unsigned long             read_ulong();
    float                      read_float();
    double                     read_double();
    string                     read_string();
    Object                     read_Object();
    TypeCode                   read_TypeCode();
    void                       read_any_array( inout AnySeq seq,  
                                            in unsigned long offset,  
                                            in unsigned long length);  
                               // Raises NO_IMPLEMENT
    void                       read_boolean_array(inout BooleanSeq seq,  
                                            in unsigned long offset,  
                                            in unsigned long length);
    void                       read_char_array( inout CharSeq seq,  
                                            in unsigned long offset,  
                                            in unsigned long length);
}
The implementation of CORBA::DataInputStream inherits from CORBA::ValueBase rather than from CORBA::Object. You cannot use, for example, _duplicate, _narrow, and _nil operations since they apply only to CORBA::Object. At this time, there is nothing of interest for users in the CORBA::ValueBase interface.

Listing 8-10 C++ Declaration

class CORBA
{
    public:

    class AnySeq /* Normal sequence definition */;
    typedef AnySeq * AnySeq_ptr;

class BooleanSeq /* Normal sequence definition */;
    typedef BooleanSeq * BooleanSeq_ptr;
    static const CORBA::TypeCode_ptr _tc_BooleanSeq;
};
class CharSeq {/* Normal sequence definition */};
typedef CharSeq * CharSeq_ptr;
static const CORBA::TypeCode_ptr _tc_CharSeq;

class OctetSeq {/* Normal sequence definition */};
typedef OctetSeq * OctetSeq_ptr;
static const CORBA::TypeCode_ptr _tc_OctetSeq;

class ShortSeq {/* Normal sequence definition */};
typedef ShortSeq * ShortSeq_ptr;
static const CORBA::TypeCode_ptr _tc_ShortSeq;

class UshortSeq {/* Normal sequence definition */};
typedef UShortSeq * UShortSeq_ptr;
static const CORBA::TypeCode_ptr _tc_UShortSeq;

class LongSeq {/* Normal sequence definition */};
typedef LongSeq * LongSeq_ptr;
static const CORBA::TypeCode_ptr _tc_LongSeq;

class UlongSeq {/* Normal sequence definition */};
typedef ULongSeq * ULongSeq_ptr;
static const CORBA::TypeCode_ptr _tc_ULongSeq;

class FloatSeq {/* Normal sequence definition */};
typedef FloatSeq * FloatSeq_ptr;
static const CORBA::TypeCode_ptr _tc_FloatSeq;

class DoubleSeq {/* Normal sequence definition */};
typedef DoubleSeq * DoubleSeq_ptr;
static const CORBA::TypeCode_ptr _tc_DoubleSeq;

class OBEXPDLL DataInputStream : public virtual ValueBase
{
public:
    static DataInputStream_ptr _downcast(ValueBase_ptr obj);

    virtual Any * read_any (); // Raises NO_IMPLEMENT
    virtual Boolean read_boolean ();
    virtual Char read_char ();
    virtual Octet read_octet ();
    virtual Short read_short ();
    virtual UShort read_ushort ();
    virtual Long read_long ();
    virtual ULong read_ulong ();
    virtual Float read_float ();
};
virtual Double        read_double ();
virtual Char *        read_string ();
virtual Object_ptr    read_Object ();
virtual TypeCode_ptr  read_TypeCode ();

virtual void read_any_array ( AnySeq & seq,
                              ULong offset, ULong length);
// Raises NO_IMPLEMENT
virtual void read_boolean_array(BooleanSeq & seq,
                                ULong offset, ULong length);
virtual void read_char_array ( CharSeq & seq,
                              ULong offset, ULong length);
virtual void read_octet_array ( OctetSeq & seq,
                              ULong offset, ULong length);
virtual void read_short_array ( ShortSeq & seq,
                              ULong offset, ULong length);
virtual void read_ushort_array (UShortSeq & seq,
                               ULong offset, ULong length);
virtual void read_long_array ( LongSeq & seq,
                              ULong offset, ULong length);
virtual void read_ulong_array ( ULongSeq & seq,
                              ULong offset, ULong length);
virtual void read_float_array ( FloatSeq & seq,
                              ULong offset, ULong length);
virtual void read_double_array (DoubleSeq & seq,
                                ULong offset, ULong length);

protected:
    DataInputStream() {
    }
    virtual ~DataInputStream() {
    }

private:
    void operator=(const DataInputStream&) {
    }
};

typedef    DataInputStream *    DataInputStream_ptr;

---

**Synopsis**

Returns a value from the stream.
Request-Level Interceptor API

C++ Binding

```cpp
<primitive> read_<primitive>();
```

Parameters
None.

Exceptions
None.

Description
The operations to read a primitive element (<primitive>) from a DataInputStream all have the same format. Each operation returns a value from the stream.

**Note:** String_var, TypeCode_var, or Object_var can be used for memory management. Otherwise, strings must be released using the string_free() operation on the CORBA object, and TypeCode or Object pointers must be released using the release() operation on the CORBA object.

The primitives are the following:

- AnySeq (Not implemented)
- BooleanSeq
- CharSeq
- OctetSeq
- ShortSeq
- UshortSeq
- LongSeq
- UlongSeq
- FloatSeq
- DoubleSeq

Return Values
None.

**DataStream::read_array_<primitive>**

Synopsis
Returns an array of primitive values from the stream into a CORBA sequence.
DataInputStream::read_array_<primitive>

C++ Binding

```cpp
void read_array_<primitive>(
    <primitive>Seq & seq,
    ULong offset,
    ULong length);
```

Parameters

- `<primitive>Seq`: A sequence of the appropriate type that will receive the array elements read.
  
  If the sequence was not long enough to contain the additional elements, the length will be set to the sum offset+length. (The length will not be adjusted down.)

- `Offset`: The offset into the array to read the elements. That is, the array will have new elements starting at array index offset up to array index offset+length-1.

- `Length`: The number of elements of the array to be returned into the `seq` parameter.

Exceptions

None.

Description

The operations to read an array of primitive elements (<primitive>) from a DataInputStream all have the same format. Each operation returns an array of primitive values from the stream into a CORBA sequence of that same primitive type.

The primitives are the following:

- AnySeq (Not implemented)
- BooleanSeq
- CharSeq
- OctetSeq
- ShortSeq
- UShortSeq
- LongSeq
- ULongSeq
- FloatSeq
- DoubleSeq
Return Values

None.
This appendix contains the following code that you can use as a place to start implementing your interceptors:

- Starter Implementation Code
- Starter Header File Code

If you use this code, replace the string YourInterceptor with the name of the interceptor you are implementing.

**Starter Implementation Code**

```c
#include <windows.h>
#include <ctype.h>

#include "YourInterceptor.h"

// Cleanup class -- suggested
class Cleanup
{
  public:
    Cleanup();
    ~Cleanup();

    // <<<Fill in your code here>>>
```

Using CORBA Request-Level Interceptors
static Cleanup CleanupOnImageExit;

#define SECURITY_BUFFSIZE 100

#if defined(WIN32)
// suggestion for standard DLL processing

BOOL WINAPI DllMain( HANDLE hDLL,
    DWORD dwReason,
    LPVOID lpReserved )
{
    switch( dwReason )
    {
    case DLL_PROCESS_ATTACH:
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    }

    // Return that the operation was successful
    return( TRUE );
}
#endif /* WIN32 */

/***************************************************************

FUNCTION NAME:         YourInterceptorInit

FUNCTIONAL DESCRIPTION:

    Initialization routine called by the ORB during initialization.
    This routine will create and return instances of the
    RequestLevelInterceptor classes that it supports.

NOTE: An interceptor library can support more than one set of
      interceptors by supplying multiple initialization entry points
      (each initialization entry must be separately registered with the
      ORB) Also, it is legal for only one kind of interceptor to be
      supplied (i.e. only a client or only a target.)

***************************************************************/
#endif  /* WIN32 */
extern "C" __declspec(dllexport) void __cdecl
#else
extern "C" void
#endif
YourInterceptor
Init(
CORBA::ORB_ptr                                       TheORB,
RequestLevelInterceptor::ClientRequestInterceptor ** ClientPtr,
RequestLevelInterceptor::TargetRequestInterceptor ** TargetPtr,
CORBA::Boolean *                                     RetStatus)
{
   // <<<Fill in your code here>>>}
}/*************************************************************/
FUNCTION NAME:       YourInterceptorClient constructor
FUNCTIONAL DESCRIPTION:
********************************************************************************
YourInterceptorClient::YourInterceptorClient(CORBA::ORB_ptr TheOrb)
{
   // This next line is useful, but not absolutely necessary.
   m_orb = TheOrb;
   // <<<Fill in your code here>>>}
}/********************************************************************************
FUNCTION NAME:       YourInterceptorClient::shutdown
FUNCTIONAL DESCRIPTION:
The shutdown operation is used by the ORB to notify an implementation of an interceptor that the interceptor is being shutdown. The ORB will destroy the instance of the interceptor once control is returned from the operation back to the ORB.
********************************************************************************
Interceptors::ShutdownReturnStatus YourInterceptorClient::shutdown(
Interceptors::ShutdownReason reason,
CORBA::Exception_ptr & excep_val)
{
   // The following lines are a suggestion only. Replace them if you wish.
Interceptors::ShutdownReturnStatus ret_status =
Interceptors::SHUTDOWN_NO_EXCEPTION;
switch (reason)
{
    case Interceptors::ORB_SHUTDOWN:
        // <<<Fill in your code here>>>
        break;
    case Interceptors::CONNECTION_ABORTED:
        // <<<Fill in your code here>>>
        break;
    case Interceptors::RESOURCES_EXCEEDED:
        // <<<Fill in your code here>>>
        break;
}
return ret_status;

/*****************************/
FUNCTION NAME: YourInterceptorClient::id

FUNCTIONAL DESCRIPTION:
The id accessor operation is used by the ORB to obtain
the vendor assigned identity of the interceptor as a string
value. This attribute is used primarily for debugging and
tracing of operations on the interceptors called by the ORB.

*******************************************************************************/
CORBA::String YourInterceptorClient::id()
{
    // <<<Fill in your code here>>>

    // The next line is a possible implementation that is useful
    return CORBA::string_dup("YourInterceptorClient");
}

/*****************************/
FUNCTION NAME: YourInterceptorClient::exception_occurred

FUNCTIONAL DESCRIPTION:
The exception_occurred operation is called on a request-level
interceptor implementation when an exception occurs.
It is called instead of the <xxx>_response
method of that interceptor. The ORB calls this operation to
allow the interceptor implementation to clean-up any state that it might have been managing that is specific to a request.

```c++
void YourInterceptorClient::exception_occurred (const RequestLevelInterceptor::ReplyContext & reply_context, CORBA::Exception_ptr & excep_val)
{
    // <<<Fill in your code here>>>
}
```

```c++
FUNCTION NAME: YourInterceptorClient::client_invoke

FUNCTIONAL DESCRIPTION:

This operation is called by the ORB anytime that an invocation is being sent to a target object, regardless of whether the target object is in a different address space or the same address space as the target object.

```c++
Interceptors::InvokeReturnStatus YourInterceptorClient::client_invoke (const RequestLevelInterceptor::RequestContext & request_context, RequestLevelInterceptor::ServiceContextList_ptr service_context, CORBA::DataInputStream_ptr request_arg_stream, CORBA::DataOutputStream_ptr reply_arg_stream, CORBA::Exception_ptr & excep_val)
{
    // The next line is a suggestion that works in conjunction with the last line below
    Interceptors::InvokeReturnStatus ret_status = Interceptors::INVOKE_NO_EXCEPTION;

    // <<<Fill in your code here>>>

    return ret_status;
}
```

```c++
FUNCTION NAME: YourInterceptorClient::client_response

FUNCTIONAL DESCRIPTION:
```
The operation is called by the ORB anytime that a reply to an invocation is being received by the initiator of the request, regardless of whether the initiator is in a different address space or the same address space as the target object.

Interceptors::ResponseReturnStatus YourInterceptorClient::client_response (const RequestLevelInterceptor::ReplyContext & reply_context, RequestLevelInterceptor::ServiceContextList_ptr service_context, CORBA::DataInputStream_ptr arg_stream, CORBA::Exception_ptr & excep_val)
{
    // The next line is a suggestion that works in conjunction with the last line below
    // See the examples for other suggestions of general use
    Interceptors::ResponseReturnStatus ret_status = Interceptors::RESPONSE_NO_EXCEPTION;

    // <<<Fill in your code here>>>

    return ret_status;
}

FUNCTION NAME:  YourInterceptorTarget constructor

FUNCTIONAL DESCRIPTION:

This function constructs the target interceptor instance. This example provides data members that could be used to implement a security interceptor.

YourInterceptorTarget::YourInterceptorTarget (CORBA::ORB_ptr TheOrb) :
    m_orb(TheOrb), // suggestion
    m_security_current(0), // suggestion for security interceptors
    m_attributes_to_get(0) // suggestion for security interceptors
{
    // <<<Fill in your code here>>>

Using CORBA Request-Level Interceptors
Starter Implementation Code

FUNCTION NAME: YourInterceptorTarget::shutdown

FUNCTIONAL DESCRIPTION:

The shutdown operation is used by the ORB to notify an implementation of an interceptor that the interceptor is being shutdown. The ORB will destroy the instance of the interceptor once control is returned from the operation back to the ORB.

FUNCTIONAL DESCRIPTION:

FUNCTION NAME: YourInterceptorTarget::id

FUNCTIONAL DESCRIPTION:
The id accessor operation is used by the ORB to obtain the vendor assigned identity of the interceptor as a string value. This attribute is used primarily for debugging and tracing of operations on the interceptors called by the ORB.

```c++
CORBA::String YourInterceptorTarget::id()
{
    // <<<Fill in your code here>>>

    // The next line is a possible implementation that is useful
    return CORBA::string_dup("YourInterceptorTarget");
}
```

**FUNCTION NAME:** YourInterceptorTarget::exception_occurred

**FUNCTIONAL DESCRIPTION:**

The exception_occurred operation is called on a request-level interceptor implementation when an exception occurs. It is called instead of the <xxx>_response method of that interceptor. The ORB calls this operation to allow the interceptor implementation to clean-up any state that it might have been managing that is specific to a request.

```c++
void YourInterceptorTarget::exception_occurred (const RequestLevelInterceptor::ReplyContext & reply_context, CORBA::Exception_ptr excep_val)
{
    // <<<Fill in your code here>>>
}
```

**FUNCTION NAME:** YourInterceptorTarget::target_invoke

**FUNCTIONAL DESCRIPTION:**

The operation is called by the ORB anytime that an invocation is being received by a target object, regardless of whether the target object is in a different address space or the same address space as the target object.
Starter Implementation Code

**************************************************************
Interceptors::InvokeReturnStatus YourInterceptorTarget::target_invoke {
    const RequestLevelInterceptor::RequestContext & request_context,
    RequestLevelInterceptor::ServiceContextList_ptr service_context,
    CORBA::DataInputStream_ptr request_arg_stream,
    CORBA::DataOutputStream_ptr reply_arg_stream,
    CORBA::Exception_ptr & excep_val)
    {
        // The next line is a suggestion that works in conjunction with the last line
        // below
        Interceptors::InvokeReturnStatus ret_status =
        Interceptors::INVOKE_NO_EXCEPTION;

        // <<<Fill in your code here>>>

        return ret_status;
    }
}

FUNCTION NAME: YourInterceptorTarget::target_response

FUNCTIONAL DESCRIPTION:

The operation is called by the ORB anytime that a reply to an invocation is being sent to the initiator of the request, regardless of whether the initiator is in a different address space or the same address space as the target object.

**************************************************************
Interceptors::ResponseReturnStatus YourInterceptorTarget::target_response {
    const RequestLevelInterceptor::ReplyContext & reply_context,
    RequestLevelInterceptor::ServiceContextList_ptr service_context,
    CORBA::DataInputStream_ptr arg_stream,
    CORBA::Exception_ptr & excep_val)
    {
        // The next line is a suggestion that works in conjunction with the last line
        // below
        Interceptors::ResponseReturnStatus ret_status =
        Interceptors::RESPONSE_NO_EXCEPTION;
return ret_status;
}

FUNCTION NAME: YourInterceptorTarget destructor

FUNCTIONAL DESCRIPTION:

Starter Header File Code

#include <CORBA.h>
#include <RequestLevelInterceptor.h>
#include <security_c.h>  //used with security

class YourInterceptorClient : public virtual RequestLevelInterceptor::ClientRequestInterceptor {
private:
  YourInterceptorClient() {}  
  CORBA::ORB_ptr m_orb;
public:
  YourInterceptorClient(CORBA::ORB_ptr TheOrb);
  ~YourInterceptorClient() {}  
  Interceptors::ShutdownReturnStatus shutdown(
    Interceptors::ShutdownReason reason,
    CORBA::Exception_ptr & excep_val);
  CORBA::String id();
  void exception_occurred (  
    const RequestLevelInterceptor::ReplyContext & reply_context,
    CORBA::Exception_ptr excep_val);
  Interceptors::InvokeReturnStatus client_invoke (  
    const RequestLevelInterceptor::RequestContext & request_context,
    RequestLevelInterceptor::ServiceContextList_ptr service_context,
CORBA::DataInputStream_ptr request_arg_stream,
CORBA::DataOutputStream_ptr reply_arg_stream,
CORBA::Exception_ptr & excep_val);
Interceptors::ResponseReturnStatus client_response (
    const RequestLevelInterceptor::ReplyContext & reply_context,
    RequestLevelInterceptor::ServiceContextList_ptr service_context,
    CORBA::DataInputStream_ptr arg_stream,
    CORBA::Exception_ptr & excep_val);
};

class YourInterceptorTarget : public virtual
    RequestLevelInterceptor::TargetRequestInterceptor
{
private:

    YourInterceptorTarget() {}
CORBA::ORB_ptr  m_orb;
SecurityLevel1::Current_ptr m_security_current;    //used with security
Security::AttributeTypeList * m_attributes_to_get; //used with security
public:

    YourInterceptorTarget(CORBAsORB_ptr TheOrb);
~YourInterceptorTarget();
Interceptors::ShutdownReturnStatus shutdown(
    Interceptors::ShutdownReason reason,
    CORBA::Exception_ptr & excep_val);
CORBA::String id();
void exception_occurred (
    const RequestLevelInterceptor::ReplyContext & reply_context,
    CORBA::Exception_ptr excep_val);
Interceptors::InvokeReturnStatus target_invoke (
    const RequestLevelInterceptor::RequestContext & request_context,
    RequestLevelInterceptor::ServiceContextList_ptr service_context,
    CORBA::DataInputStream_ptr request_arg_stream,
    CORBA::DataOutputStream_ptr reply_arg_stream,
    CORBA::Exception_ptr & excep_val);
Interceptors::ResponseReturnStatus target_response (
    const RequestLevelInterceptor::ReplyContext & reply_context,
    RequestLevelInterceptor::ServiceContextList_ptr service_context,
    CORBA::DataInputStream_ptr arg_stream,
    CORBA::Exception_ptr & excep_val);
};