



Programmer's Guide



VERSION 4.5

VisiBroker® for Java™

Inprise Corporation, 100 Enterprise Way
Scotts Valley, CA 95066-3249

Inprise may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1996, 2000 Inprise Corporation. All rights reserved. All Inprise and Borland brands and product names are trademarks or registered trademarks of Inprise Corporation. Java and all Javabased marks are trademarks or registered trademarks in the United States and other countries. Other brands and product names are trademarks or registered trademarks of their respective owners.

Printed in the U.S.A. P D F V J D 0 0 4 5 W W 2 1 0 0 2

Contents

Chapter 1

Introduction 1-1

What's new in this manual	1-1
What's new in Visibroker 4.1	1-2
What's new in Visibroker 4.5	1-3
What's in this guide?	1-4
Manual conventions	1-5
Typographic conventions	1-5
Platform conventions	1-5
Where to find additional information	1-6
Contacting Inprise developer support.	1-6

Part I

Basic Concepts

Chapter 2

Understanding the CORBA model 2-1

What is CORBA?	2-1
What is VisiBroker?	2-2
VisiBroker for Java features.	2-3
VisiBroker Smart Agent architecture	2-3
Enhanced object discovery with the Location Service	2-3
Implementation and object activation support	2-3
Robust thread and connection management	2-3
IDL compilers.	2-4
Dynamic invocation with DII and DSI.	2-4
Interface and implementation repositories	2-4
Server-side portability	2-5
Customizing the ORB with interceptors and object wrappers	2-5
Backing stores in the Naming Service	2-5
Web naming.	2-5
Defining interfaces without IDL	2-6
Gatekeeper (optional feature).	2-6
VisiBroker CORBA compliance.	2-6
VisiBroker development environment.	2-6
Programmer's tools	2-6
CORBA services tools	2-7
Administration tools	2-7
Java Development Environment	2-7
Java Runtime Environment	2-7
What's Required for VisiBroker?	2-8
Java-enabled Web Browser	2-8
Interoperability with VisiBroker for C++	2-8
Interoperability with other ORB products	2-9

IDL to Java mapping	2-9
-------------------------------	-----

Chapter 3

Setting up your environment 3-1

Setting the Path environment variable	3-1
Updating the PATH on a Windows platform.	3-1
Updating the PATH on a Windows NT platform	3-1
Setting the Path on a UNIX platform	3-2
CLASSPATH.	3-2
Setting the VBROKER_ADM environment variable	3-2
Setting VBROKER_ADM on a Windows platform	3-2
Setting VBROKER_ADM on a UNIX platform.	3-2
Setting the OSAGENT_PORT environment variable	3-3
Logging output	3-3

Chapter 4

Developing an example application with VisiBroker 4-1

Development process.	4-1
Step 1: Defining object interfaces	4-3
Writing the account interface in IDL	4-3
Step 2: Generating client stubs and server servants.	4-3
Files produced by the idl compiler	4-4
Step 3: Implementing the client	4-5
Client.java	4-5
Binding to the AccountManager object	4-5
Obtaining an Account object	4-6
Obtaining the balance.	4-6
AccountManagerHelper.java.	4-6
Other methods	4-6
Step 4: Implementing the server.	4-6
Server.java	4-7
Step 5: Building the example	4-8
Compiling the example.	4-8
Step 6: Starting the server and running the example	4-8
Starting the Smart Agent	4-8
Starting the server.	4-9
Running the client.	4-9
Deploying applications with VisiBroker	4-9
VisiBroker for Java applications	4-10

Deploying applications.	4-10
Using vbj.	4-11
Executing client applications	4-11
Executing server applications	4-12

Chapter 5

Handling exceptions **5-1**

Exceptions in the CORBA model.	5-1
System exceptions	5-1
Obtaining completion status	5-2
Catching system exceptions.	5-3
Downcasting exceptions to a system exception.	5-3
Catching specific types of system exceptions	5-4
User exceptions	5-5
Defining user exceptions	5-5
Modifying the object to raise the exception	5-6
Catching user exceptions	5-6
Adding fields to user exceptions	5-7

Part II

Server concepts

Chapter 6

Server basics **6-1**

Overview	6-1
Initializing the ORB	6-1
Creating the POA	6-2
Obtaining a reference to the root POA	6-2
Creating the child POA	6-3
Implementing servant methods	6-3
Activating the POA	6-5
Activating objects	6-5
Waiting for client requests	6-5
Complete example.	6-6

Chapter 7

Using POAs **7-1**

What is a Portable Object Adapter?	7-1
POA terminology.	7-2
Steps for creating and using POAs	7-3
POA policies	7-3
Thread policy	7-3
Lifespan policy	7-4
Object ID Uniqueness policy	7-4
ID Assignment policy	7-4
Servant Retention policy.	7-4

Request Processing policy	7-5
Implicit Activation policy.	7-5
Bind Support policy	7-5
Creating POAs	7-6
POA naming convention	7-6
Obtaining the rootPOA	7-6
Setting the POA properties.	7-7
Creating and activating the POA	7-7
Activating objects	7-8
Activating objects explicitly	7-8
Activating objects on demand	7-9
Activating objects implicitly	7-9
Activating with the default servant	7-9
Deactivating objects.	7-11
Using servants and servant managers	7-12
ServantActivators	7-13
ServantLocators	7-15
Managing POAs with the POA manager.	7-17
Getting the current state	7-18
Holding state.	7-18
Active state.	7-19
Discarding state	7-19
Inactive state	7-19
Setting the listening and dispatching properties	7-20
Setting the server engine properties.	7-21
Setting the server connection manager properties.	7-21
Manager properties	7-21
Listener properties.	7-22
Dispatcher properties	7-22
When to use these properties	7-22
Adapter activators	7-24
Processing requests	7-25

Chapter 8

Managing threads and connections **8-1**

Using threads with VisiBroker.	8-1
What thread policies does VisiBroker provide?	8-2
Thread pooling policy	8-2
Thread-per-session policy	8-7
What connection management does VisiBroker provide?	8-8
Setting dispatch policies and properties	8-9
Thread pooling	8-10
Threads-per-session.	8-10
Coding considerations	8-10

Chapter 9	
Using the tie mechanism	9-1
How does the tie mechanism work?	9-1
Example program	9-2
Location of an example program using the tie mechanism	9-2
Changes to the server class	9-2
Changes to the AccountManager.	9-3
Changes to the Account class	9-4
Building the tie example.	9-4

Part III

Client concepts

Chapter 10	
Client basics	10-1
Initializing the ORB	10-1
Binding to objects	10-1
Action performed during the bind process	10-2
Invoking operations on an object	10-3
Manipulating object references.	10-3
Converting a reference to a string	10-3
Obtaining object and interface names	10-4
Determining the type of an object reference.	10-4
Determining the location and state of bound objects	10-4
Narrowing object references	10-5
Widening object references	10-5
Using quality of service	10-6
Understanding Quality of Service	10-6
Policy overrides and effective policies	10-6
QoS interfaces.	10-6
org.omg.CORBA.Object	10-6
com.inprise.vbroker.CORBA.Object	10-6
org.omg.CORBA.PolicyManager	10-7
org.omg.CORBA.PolicyCurrent.	10-7
org.omg.Messaging.RebindPolicy	10-8
com.inprise.vbroker.QoSExt.RelativeConnectionTimeoutPolicy	10-8
com.inprise.vbroker.QoSExt.DeferBindPolicy	10-8
com.inprise.vbroker.QoSExt.ExclusiveConnectionPolicy	10-9
com.inprise.vbroker.QoSExt.SyncScopePolicy	10-9
QoS exceptions	10-9
QoS example	10-10

Part IV

Configuration and management

Chapter 11	
Using the VisiBroker Console	11-1
What is the VisiBroker Console?	11-1
Starting the VisiBroker Console	11-2
Configuring the Console	11-2
Setting preferences	11-3
Viewing system information.	11-4
Navigating the VisiBroker Console	11-5
Menu bar	11-5
Toolbar	11-5
Status bar.	11-5
Pull down or context menus	11-5
Navigation pane.	11-6
Content pane.	11-6
Supported ORB Services	11-6
Location Service	11-6
Naming Services.	11-6
Interface Repositories.	11-7
Implementation Repositories	11-7
Server Manager	11-7
Gatekeeper	11-7

Chapter 12	
Using the ORB Services browsers	12-1
Introduction	12-1
Location Service.	12-1
Accessing the Location Service browser	12-2
Refreshing the active object list	12-3
Naming Services	12-3
Accessing the Naming Services	12-4
Browsing the Naming Service	12-5
Browsing the VisiBroker Naming Service clusters	12-5
Browsing the VisiBroker Naming Service federations	12-6
Implementation Repositories	12-7
Accessing the Implementation Repositories	12-7
Interface Repositories.	12-8
Viewing an Interface Repository.	12-9
Accessing the Interface Repositories	12-10
Browsing the Interface Repositories.	12-11

Chapter 13	
Using the Server Manager	13-1
What is the Server Manager	13-1
Viewing the top-level container	13-2

Server Manager browser.	13-2
Using the VisiBroker 4.x example server. . .	13-2
Setting security for the Server Manager . .	13-4
Using the Server Manager browser	13-4
Viewing the contents of a server	13-4
Enabling the server.	13-5
Invoking methods	13-6
Setting properties.	13-7
Property types.	13-7
Specifying the property storage file. . .	13-8
Changing property settings	13-8

Chapter 14 Setting properties 14-1

Overview	14-1
Setting Visibroker properties	14-2
Shell/console environment variables . .	14-2
Windows registry.	14-2
Command line arguments.	14-3
Applet parameters	14-3
System properties	14-4
Programmatically via ORB.init	14-4
Property file via ORBpropStorage option .	14-4
Properties file	14-4
Default properties file	14-5
Property precedence under NT and Unix. . .	14-5
Property precedence for applets	14-5
VisiBroker for Java properties	14-6

Part V Tools and services

Chapter 15 Using IDL 15-1

Introduction to IDL	15-1
How the IDL compiler generates code	15-2
Example IDL specification.	15-2
Looking at the generated code	15-2
<interface name>Stub.java.	15-3
<interface name>.java	15-3
<interface name>Helper.java	15-3
<interface name>Holder.java	15-5
<interface name>Operations.java	15-5
<interface name>POA.java	15-5
<interface name>POATie.java	15-6
Defining interface attributes in the IDL . . .	15-7
Specifying oneway methods with no return value.	15-7

Specifying an interface in IDL that inherits from another interface	15-8
--	------

Chapter 16 Using the Smart Agent 16-1

What is the Smart Agent?	16-1
Locating Smart Agents	16-1
Locating objects through Agent cooperation	16-2
Cooperating with the OAD to connect with objects.	16-2
Starting a Smart Agent (osagent)	16-2
Verbose output	16-3
Disabling the agent	16-3
Ensuring Agent availability	16-3
Checking client existence	16-3
Working within ORB domains.	16-4
Connecting Smart Agents on different local networks	16-5
How Smart Agents detect each other	16-6
Working with multihomed hosts	16-6
Specifying interface usage for Smart Agents	16-7
Using point-to-point communications	16-8
Specifying a host as a runtime parameter .	16-8
Specifying an IP address with an environment variable	16-9
Specifying hosts with the agentaddr file . .	16-9
Ensuring object availability	16-10
Invoking methods on stateless objects . .	16-10
Achieving fault-tolerance for objects that maintain state	16-10
Replicating objects registered with the OAD. . .	16-10
Migrating objects between hosts	16-11
Migrating objects that maintain state . . .	16-11
Migrating instantiated objects	16-11
Migrating objects registered with the OAD	16-11
Reporting all objects and services.	16-12
Binding to Objects	16-12

Chapter 17 Using the Location Service 17-1

What is the Location Service?	17-1
Location Service components	17-3
What is the Location Service agent?.	17-3
Obtaining names of all hosts running Smart Agents.	17-4
Finding all accessible interfaces	17-4

Obtaining references to instances of an interface	17-4
Obtaining references to like-named instances of an interface	17-5
What is a trigger?	17-5
Looking at trigger methods	17-5
Creating triggers	17-6
Looking at only the first instance found by a trigger	17-6
Querying an agent	17-7
Finding all instances of an interface	17-7
Finding everything known to Smart Agents	17-8
Writing and registering a trigger handler . . .	17-10
Implementing and registering a trigger handler	17-10

Chapter 18

Using the Naming Service 18-1

Overview	18-1
Understanding the namespace	18-2
Naming contexts	18-3
Naming context factories	18-4
Names and NameComponent	18-4
Name resolution	18-5
Stringified names	18-5
Simple and complex names	18-5
Running the Naming Service.	18-6
Installing the Naming Service.	18-6
Configuring the Naming Service	18-6
Starting the Naming Service	18-6
Starting the Naming Service with vbj	18-7
Invoking the Naming Service from the Command Line	18-7
Configuring nsutil	18-7
Running nsutil	18-8
Closing nsutil	18-8
Bootstrapping a Naming Service.	18-9
Calling resolve_initial_references.	18-9
Using -DSVCnameroot	18-9
Using -DORBInitRef	18-9
Using a corbaloc URL	18-10
Using a corbaname URL	18-10
-DORBDefaultInitRef	18-10
Using -DORBDefaultInitRef with a corbaloc URL	18-10
Using -DORBDefaultInitRef with corbaname	18-10
NamingContext	18-11
NamingContextExt	18-12

Default naming contexts	18-12
Obtaining the default naming context . . .	18-12
Naming Service Properties.	18-13
Pluggable backing store	18-14
Types of backing stores	18-14
In-memory adaptor	18-14
JDBC adaptor.	18-14
DataExpress adaptor	18-14
JNDI adaptor	18-15
Configuration and use	18-15
Properties file.	18-15
JDBC Adaptor properties	18-16
DataExpress Adaptor properties	18-18
JNDI adaptor properties	18-18
Caching facility.	18-18
Clusters	18-20
Clustering criteria	18-20
Cluster and ClusterManager interfaces. .	18-20
Creating a cluster	18-22
Explicit and implicit clusters	18-22
Load balancing	18-23
Failover.	18-23
Configuring the Naming Service for fault tolerance	18-24
Import statements for Java.	18-24
Sample programs	18-25
Binding a name in Java	18-25

Chapter 19

Using the Event Service 19-1

Overview	19-1
Proxy consumers and suppliers	19-2
OMG common object services specification	19-3
Communication models	19-4
Push model	19-5
Pull model	19-5
Using event channels.	19-6
Example push supplier and consumer	19-8
Running the Push model example.	19-8
Running the Pull model example	19-8
PullSupply	19-9
Executing PullSupply	19-10
PullConsume.	19-12
Executing PullConsume	19-12
Starting the Event Service	19-14
Setting the queue length	19-14
In-process event channel.	19-15
Java usage	19-16
Java EventLibrary class	19-16

Java example	19-16
Import statements for Java	19-16
Interface reference	19-17
EventChannel.	19-17
EventLibrary (Java)	19-17
EventLibrary methods	19-17
ConsumerAdmin.	19-18
SupplierAdmin.	19-19
ProxyPullConsumer	19-19
ProxyPushConsumer	19-20
ProxyPullSupplier	19-20
ProxyPushSupplier.	19-20
PullConsumer	19-21
PushConsumer	19-21
PullSupplier.	19-22
PullSupplier methods	19-22
PushSupplier	19-23

Chapter 20

Using the Object Activation Daemon 20-1

Automatic activation of objects and servers . .	20-1
Locating the implementation repository data. .	20-2
Activating servers	20-2
Starting the Object Activation Daemon	20-2
Starting the Object Activation Daemon on a	
Windows platform	20-2
Starting the Object Activation Daemon on a	
UNIX platform	20-3
Using the Object Activation Daemon utilities .	20-4
Converting interface names to repository IDs	
20-4	
Listing objects with oadutil list	20-5
Description	20-6
Registering objects with oadutil	20-6
Example 1: Specifying repository ID . .	20-8
Example 2: Specifying IDL interface name. .	20-8
Remote registration to an OAD	20-8
Accessing a server without using the Smart	
Agent.	20-8
Distinguishing between multiple instances of an	
object.	20-9
Setting activation properties using the	
CreationImplDef class	20-9
Dynamically changing an ORB implementation	
20-10	
OAD Registration using	
OAD::reg_implementation	20-10
Example of object creation and registration	
20-11	

Arguments passed by the OAD	20-12
Un-registering objects	20-12
Un-registering objects using the oadutil tool. . .	20-12
Unregistration example	20-13
Un-registering with the OAD operations. .	20-13
Displaying the contents of the implementation	
repository.	20-14
IDL interface to the OAD	20-14

Chapter 21

Using interface repositories 21-1

What is an interface repository?	21-1
What does an interface repository contain? .	21-2
How many interface repositories can you have?.	21-2
Creating and viewing an interface repository with	
irep	21-3
Creating an interface repository with irep .	21-3
Viewing the contents of the interface repository.	
21-4	
Updating an interface repository with idl2ir. .	21-5
Understanding the structure of the interface	
repository	21-5
Identifying objects in the interface repository . .	21-6
Types of objects that can be stored in the	
interface repository	21-7
Inherited interfaces	21-8
Accessing an interface repository	21-8
Example programs	21-9

Part VI

Advanced concepts

Chapter 22

Using the Dynamic Invocation Interface 22-1

What is the Dynamic Invocation Interface? . .	22-1
Introducing the main DII concepts	22-2
Using request objects	22-2
Encapsulating arguments with the Any type.	
22-3	
Options for sending requests	22-4
Options for receiving replies	22-4
Steps for invoking object operations	
dynamically	22-4
Location of example programs for using the DII	
22-5	

Using the idl2java compiler	22-5
Obtaining a generic object reference	22-5
Creating and initializing a request	22-6
Request interface	22-6
Ways to create and initialize a DII request	22-6
Using the create_request method	22-7
Using the _request method	22-7
Example of creating a Request object	22-8
Setting arguments for the request	22-8
Implementing a list of arguments with the	
NVList	22-8
Setting input and output arguments with the	
NamedValue Class	22-9
Passing type safely with the Any class	22-10
Representing argument or attribute types with	
the TypeCode class	22-10
Sending DII requests and receiving results	22-13
Invoking a request	22-13
Sending a deferred DII request with the	
send_deferred method	22-14
Sending an asynchronous DII request with the	
send_oneway method	22-14
Sending multiple requests	22-15
Receiving multiple requests	22-15
Using the interface repository with the DII	22-16

Chapter 23	
Using the Dynamic Skeleton Interface	
23-1	
What is the Dynamic Skeleton Interface?	23-1
Using the idl2java compiler	23-2
Steps for creating object implementations	
dynamically	23-2
Location of an example program for using the	
DSI	23-2
Extending the DynamicImplementation class	23-3
Example of designing objects for dynamic	
requests	23-3
Specifying repository ids	23-5
Looking at the ServerRequest class	23-6
Implementing the Account object	23-6
Implementing the AccountManager object	23-7
Processing input parameters	23-7
Setting the return value	23-7
Server implementation	23-8

Chapter 24	
Using interceptors	
24-1	

Overview	24-1
Interceptor interfaces and managers	24-2
Client interceptors	24-2
BindInterceptor	24-2
ClientRequestInterceptor	24-3
Server interceptors	24-4
POALifeCycleInterceptor	24-4
ActiveObjectLifeCycleInterceptor	24-4
ServerRequestInterceptor	24-4
IORCreationInterceptor	24-5
Service Resolver interceptor	24-5
Default interceptor classes	24-6
Registering interceptors with the VisiBroker	
ORB	24-6
Creating interceptor objects	24-7
Loading interceptors	24-7
Example interceptors	24-7
Example code	24-8
Client-server interceptors example	24-8
ServiceResolverInterceptor example	24-9
Code listings	24-10
Passing information between your interceptors	24-16

Chapter 25	
Using object wrappers	
25-1	
Overview	25-1
Typed and un-typed object wrappers	25-2
Special idl2java requirements	25-2
Example applications	25-2
Un-typed object wrappers	25-2
Using multiple, un-typed object wrappers	25-3
Order of pre_method invocation	25-4
Order of post_method invocation	25-4
Using un-typed object wrappers	25-4
Implementing an un-typed object wrapper	
factory	25-4
Implementing an un-typed object wrapper	
pre_method and post_method parameters	25-6
Creating and registering un-typed object	
wrapper factories	25-6
Removing un-typed object wrappers	25-8
Typed object wrappers	25-8
Using multiple, typed object wrappers	25-9
Order of invocation	25-10
Typed object wrappers with co-located client	
and servers	25-11

Using typed object wrappers	25-11
Implementing typed object wrappers	25-11
Registering typed object wrappers for a client	25-12
Registering typed object wrappers for a server	25-13
Removing typed object wrappers	25-14
Combined use of un-typed and typed object wrappers	25-14
Command-line arguments for typed wrappers	25-15
Initializer for typed wrappers.	25-15
Command-line arguments for un-typed wrappers.	25-16
Initializers for un-typed wrappers	25-17
Executing the sample applications	25-18
Turning on timing and tracing object wrappers.	25-18
Turning on caching and security object wrappers.	25-19
Turning on typed and un-typed wrappers.	25-19
Executing a co-located client and server	25-19

Chapter 26

Using RMI over IIOP 26-1

Overview	26-1
java2iiop and java2idl tools	26-1
Using java2iiop.	26-1
Supported interfaces	26-2
Running java2iiop	26-2
Reverse mapping of Java classes to IDL	26-2
Completing the development process	26-3
RMI-IIOP Bank example	26-3
Supported data types	26-5
Mapping primitive data types	26-5
Mapping complex data types	26-6
Interfaces	26-6
Arrays	26-6

Chapter 27

Using the dynamically managed types 27-1

Overview	27-1
DynAny types	27-1
Usage restrictions.	27-2

Creating a DynAny	27-2
Initializing and accessing the value in a DynAny	27-3
Constructed data types.	27-3
Traversing the components in a constructed data type	27-3
DynEnum	27-3
DynStruct.	27-4
DynUnion	27-4
DynSequence and DynArray	27-4
Example IDL.	27-4
Example client application.	27-5
Example server application	27-6

Chapter 28

Using valuetypes 28-1

Understanding valuetypes.	28-1
Concrete valuetypes.	28-2
Valuetype derivation	28-2
Sharing semantics	28-2
Factories	28-2
Abstract valuetypes	28-2
Implementing valuetypes	28-3
Defining your valuetypes	28-3
Compiling your IDL file	28-3
Inheriting the valuetype base class	28-4
Implementing the Factory class	28-4
Registering your Factory with the ORB.	28-5
Implementing factories.	28-5
Factories and valuetypes	28-6
Registering valuetypes	28-6
Boxed valuetypes	28-7
Abstract interfaces	28-7
Custom valuetypes	28-8
Truncatable valuetypes.	28-9

Chapter 29

Using URL naming 29-1

URL Naming Service.	29-1
Registering objects	29-2
Locating an object by URL.	29-4

Chapter 30

Bidirectional Communication 7

Using bidirectional IIOP	7
Bidirectional ORB properties	8
About the examples.	9

Enabling bidirectional IIOP for existing applications 9

Explicitly enabling bidirectional IIOP 9

Security considerations 11

Part VII

Backward compatibility

Chapter 31

Using the BOA with VisiBroker 4.x 30-1

Compiling your BOA code with VisiBroker 4.x 30-1

Supporting BOA options 30-1

Limitations in using the BOA. 30-2

Using object activators 30-2

Naming objects under the BOA 30-2

 Object names 30-2

Chapter 32

Migrating VisiBroker code 31-1

Migrator. 31-1

 Changes to package name prefixes. 31-2

 Changes to class names 31-2

 Changes to API calls 31-3

 Changes from BOA to POA 31-3

 Changes in use of interceptors 31-3

Invoking the migrator 31-4

Using migrated code 31-4

Manually Migrating BOA to POA 31-5

 Looking at an example 31-5

 Obtaining a reference to the root POA . 31-5

 Setting the POA policies 31-5

 Defining the servant. 31-6

 Activating the POA manager. 31-6

 Waiting for incoming requests 31-7

 Looking at the other files 31-7

 Mapping BOA types to POA policies 31-7

Migrating to new package names. 31-7

Migrating to new class names 31-8

Migrating to new API calls 31-8

Migrating interceptors 31-9

 Using VisiBroker 3.x interceptors 31-9

Chapter 33

Using object activators 32-1

Deferring object activation. 32-1

Activator interface 32-1

Using the service activation approach 32-2

 Deferring object activation using service activators 32-3

 Example of deferred object activation for a service. 32-3

odb.idl interface.32-4
Implementing a service activator32-5
Instantiating the service activator. . . .32-6
Using a service activator to activate an object
32-6

Appendix A
CORBA exceptions **A-1**

Glossary **G-1**

Index **I-1**

Tables

3.1	Summary of log files produced on Windows platforms	3-3	21.1	Objects used to identify and classify interface repository objects	21-6
4.1	Command-line arguments for client applications.	4-12	21.2	Objects that can be stored in the interface repository	21-7
5.1	CORBA-defined system exceptions	5-1	21.3	Interfaces inherited by many IR objects .	21-8
7.1	Portable Object Adapter terminology . . .	7-2	22.1	NamedValue methods	22-9
10.1	Methods stringification and de-stringification	10-3	22.2	TypeCode kinds and parameters	22-10
10.2	Methods obtaining interface and object names	10-4	24.2	Results of executing the example interceptor .	24-8
10.3	Methods determining the type of an object reference	10-4	25.1	Comparison of features for typed and un-typed object wrappers	25-2
10.4	Methods for determining location and state of object reference.	10-5	25.2	Common arguments for the pre_method and post_method methods	25-6
17.1	Obtaining references to objects that implement a given interface	17-4	25.3	Command-line properties for enabling or disabling BankWrappers.	25-15
17.2	References to like-named instances of an interface.	17-5	25.4	Command-line properties for enabling or disabling UtilityObjectWrappers	25-17
17.3	Trigger methods	17-5	26.1	Mapping Java types to IDL/IIOP	26-5
17.4	TriggerHandler interface method	17-6	27.1	Interfaces derived from DynAny that represent constructed data types	27-2
18.1	Naming service properties	18-13	32.1	Changes to package name prefixes	31-2
18.2	Default properties common to all adaptors. .	18-15	32.2	Changes to class names	31-2
18.3	Example of a JNDI adaptor configuration file	18-18	32.3	Changes to API calls	31-3
19.1	Connecting Suppliers to an EventChannel . .	19-7	32.4	Migrator options	31-4
19.2	Connecting Consumers to an EventChannel .	19-8	32.5	Driver options.	31-4
19.3	PullConsume commands	19-12	32.6	Class name changes	31-5
			32.7	Mapping BOA types to POA policies . .	31-7
			33.1	Files in the odb example for service activation	32-3
			A.1	CORBA exceptions and possible causes .	A-1

A.2 CORBA exception minor codes A-5

Figures

2.1	Client program acting on an object	2-2	17.3	Smart Agents on a network with instances of an interface	17-4
2.2	VisiBroker architecture	2-2	18.1	Binding, resolving, and using an object name from a naming context within a namespace	18-2
4.1	Developing the sample bank application	4-2	18.2	Naming scheme for an order entry system	18-3
4.2	Client and server programs deployed with VisiBroker ORBs	4-10	19.1	Supplier-Consumer communication model	19-2
7.1	Overview of the POA	7-2	19.2	Consumer and supplier proxy objects	19-3
7.2	Example servant manager function	7-12	19.3	Push model	19-4
7.3	Server engine overview	7-20	19.4	Pull model	19-6
8.1	Pool of threads is available	8-3	21.1	Interface repository object hierarchy for Bank.idl	21-6
8.2	Client application #1 sends a request	8-4	24.1	How interceptors work	24-1
8.3	Client application #2 sends a request	8-5	25.1	Single un-typed object wrapper	25-3
8.4	Client application #1 sends a second request	8-6	25.2	Multiple un-typed object wrappers	25-3
8.5	Object implementation using the thread-per-session policy	8-7	25.3	Single typed object wrapper registered	25-9
8.6	Second request comes in from the same client	8-8	25.4	Multiple, typed object wrappers registered	25-10
8.7	Binding to two objects in the same server process	8-9	25.5	Typed object wrapper invocation order	25-11
8.8	Binding to an object in a server process	8-9	33.1	Diagram showing the process of deferring activation for a service	32-3
10.1	Client interaction with the Smart Agent	10-2			
11.1	VisiBroker Console	11-2			
12.8	Console's Interface Repositories browser with the repository expanded	12-11			
13.3	Server Manager browser with selected server object	13-5			
16.1	Running separate ORB domains simultaneously	16-4			
16.2	Two Smart Agents on separate local networks	16-5			
16.3	Smart Agent on a multihomed host	16-6			
16.4	Setting the OSAGENT_ADDR environment variable using the C shell	16-9			
17.1	Using the Smart Agent to find instances of objects	17-2			
17.2	Use of interface repository IDs and instance names	17-3			

Introduction

VisiBroker allows you to develop and deploy distributed object-based applications, as defined in the Common Object Request Broker (CORBA) specification.

The VisiBroker for Java *Programmer's Guide* provides you information on how to get started with the VisiBroker fundamentals, use the VisiBroker Console to simplify certain functions, and work with the more advanced features. It is written for Java programmers who are familiar with object-oriented development.

This chapter highlights the latest features, and identifies typographical and platform conventions used throughout the manual. It also tells you where to find additional information about Common Object Request Broker Architecture (CORBA) and the remaining VisiBroker for Java documentation set, and how to contact Inprise developer support.

What's new in this manual

This manual has been updated to reflect the latest VisiBroker release. The new features and enhancements include:

- **CORBA 2.3 compliance:** VisiBroker for Java is fully compliant with the CORBA specification (version 2.3) from the Object Management Group (OMG). For more details, refer to the CORBA specification located at <http://www.omg.org/.qqq--> is this still true, or should it be 2.4?
- **Ability to establish exclusive connections:** Information about the `ExclusiveConnectionPolicy` and about an example is provided in “com.inprise.vbroker.QoSExt.ExclusiveConnectionPolicy” on page 10-9.
- **Ability to define the level of synchronization of a request with its target:** Information about the `SyncScopePolicy` is provided in “com.inprise.vbroker.QoSExt.SyncScopePolicy” on page 10-9.

- Chapter 14, “Setting properties” has been expanded to include additional information about precedence of properties.

Other updates include new sample code snippets to reflect the new VisiBroker features. New interfaces and methods are covered in the *VisiBroker for Java Reference*.

What's new in Visibroker 4.1

The VisiBroker 4.1 features and enhancements include:

- **Abstract base support:** This feature provides CORBA 2.3.1 compliant support for VisiBroker 4.x and ensures that its CORBA 2.3.1 compliant Interface Repository is compatible with VisiBroker 3.x clients.
- **Class downloading:** This feature allows the client to receive or find implementations of some classes that are not stored locally on the system. Depending on the security policy configured for your server, clients will be able to download classes from one source but will be denied the ability to download them from another source.
- **Redesigned Interface Repository:** This feature has been enhanced to make sure that it is compatible with the VisiBroker for Java 3.4 Interface Repository.
- **Connection timeout:** This feature allows you to indicate a timeout after which attempts to connect to an object using one of the available endpoints will be aborted.
- **Naming Service:** The new VisiBroker Naming Service provides clustering and fault tolerance features, and persistence is handled differently. Clustering allows you to associate a number of bindings with a single name. Fault tolerance features include failover and load balancing. To maintain persistence, you can now store the namespace in a relational database. Previous versions stored the namespace in a flat logging file. The `corbaloc` and `corbaname` provide stringified object references which can be used in an Internet environment. This allows you to refer to objects by a URL. See Chapter 18, “Using the Naming Service,” for a description of how to use the Naming Service.
- **Portable Object Adaptor (POA):** The POA offers portability on the server side. This feature replaces the Basic Object Adapter (BOA). Although BOA is being deprecated, VisiBroker 4.0 will still support BOA functionality. See Chapter 7, “Using POAs,” for an explanation of how to use the POA.
- **Objects by value (OBV) or Value types:** Previous versions of CORBA allowed you to pass objects between clients and servers by reference. However, CORBA 2.3 allows you to pass objects by value between clients and servers using VisiBroker. OBV is interoperable with other 2.3-compliant ORBs. See Chapter 28, “Using valuetypes,” for more information on this feature.
- **Property Management:** This feature provides you with a way to centralized management of properties. Using the Property Management, you can get/set property values, and register yourself as an observer so that the Property Manager

can update and query the observer. See Chapter 14, “Setting properties,” for more information on the Property Management.

- **Interceptors and object wrappers:** This feature has been upgraded to CORBA 2.3 specifications. The ORB provides a set of APIs known as interceptors which provide a way to plug in additional ORB behavior such as support for transactions and security, which may be defined on either the client or server side. One of the main difference in this release is that now the interceptors have scope. See Chapter 24, “Using interceptors,” for more information on how to use the VisiBroker interceptor.
- **Quality of Service (QoS):** This feature, which implements the CORBA 2.3 Messaging Specification, allows you to define properties that influence how connections are made. You perform client-side policy management by setting properties that are associated with connections or client/server pairs. See “Using quality of service” on page 10-6 for a description of the VisiBroker QoS features.
- **Naming Service:** VisiBroker 4.1 has a combined JDBC adaptor. The JDBC adaptor and the OptJDBC adaptor have been combined to create a new optimized adaptor. If you set the `vbroker.naming.backingStoreType` property to either JDBC or OptJDBC, VisiBroker will use the combined adaptor, which is backwards compatible with both the VisiBroker 4.0 JDBC and OptJDBC adaptors. There are no backward compatibility issues with the InMemory adaptor. The 4.5 DataExpress (DX) adaptor is backward compatible with the 4.0 DataExpress. If the same backingstore is used with the VisiBroker 4.5 name server, the 4.0 datastore schema will automatically be converted into the 4.5 name server schema when you start the VisiBroker 4.5 name server for the first time. If the DataExpress adaptor is used, you will also need to have the JDataStore JDBC driver installed and set in your classpath before you start the 4.5 name server in order to convert from 4.0 to 4.5 schema.

What's new in Visibroker 4.5

New features and enhancements in Visibroker 4.5 include:

- **Bidirectional support:** This feature makes it possible for a server to asynchronously connect with a client. For more information about this feature, see the *VisiBroker for Java Programmer's Guide*. A number of policies have been added to support this feature.
- **SyncScope support:** Support for CORBA 4.5-compliant SyncScope capabilities is now provided (see “`com.inprise.vbroker.QoSExt.SyncScopePolicy`” on page 10-9).
- **Ability to create exclusive connections.** For more information see “`com.inprise.vbroker.QoSExt.ExclusiveConnectionPolicy`” on page 10-9.
- Windows registry entries and environment variables `VBROKER_JAVAVM` and `VBROKER_TAG` are no longer used.

What's in this guide?

This VisiBroker for Java *Programmer's Guide* provides detailed information on developing distributed object-based applications using VisiBroker for Java. It contains the following sections:

- **Part I, "Basic Concepts."** This part presents an introduction to VisiBroker for Java. It also includes an overview of the CORBA model and a quick start example designed to introduce you to the VisiBroker development principles and the handling of exceptions.
- **Part II, "Server concepts."** This part describes how to develop a VisiBroker server, use the Portable Object Adapter (POA), thread management, and the tie mechanism.
- **Part III, "Client concepts."** This part describes how to develop a VisiBroker client.
- **Part IV, "Configuration and management."** This part is designed to familiarize you with the configuration and management of VisiBroker ORB and its CORBA services, using the Console and its associated browsers. This allows you to perform many of your configuration tasks in one location that previously were performed on the command line. From the Console, you can access browsers for the ORB services and repositories, the Server Manager, and the Gatekeeper. From this central location, you can easily view, monitor, and manage VisiBroker Services, servers and objects. The configuration of VisiBroker using properties files is also described.
- **Part V, "Tools and services."** This part describes the IDL compiler, the Smart Agent, the Location, Event, and Naming services, the Object Activation Daemon, and the Interface Repository.
- **Part VI, "Advanced concepts."** This part describes advanced concepts such as the Dynamic Invocation Interface, the Dynamic Skeleton Interface, Interceptors, Object Wrappers, the DynAny class, and ValueTypes.
- **Part VII, "Backward compatibility."** This part describes compatibility issues between previous releases of VisiBroker and the current one.
- **Appendix A, "CORBA exceptions."** Contains additional information about CORBA exceptions that can be thrown by the VisiBroker ORB, and explains possible causes for VisiBroker to throw them.
- **"Glossary."** Provides a glossary of commonly used terms.

Manual conventions

This section identifies this manual's typographical and platform conventions.

Typographic conventions

This manual uses the following conventions:

Convention	Used for
boldface	Bold type indicates that syntax should be typed exactly as shown. For UNIX, used to indicate database names, filenames, and similar terms.
<i>italics</i>	Italics indicates information that the user or application provides, such as variables in syntax diagrams. Also used to introduce new terms.
<code>computer</code>	Computer typeface is used for sample command lines and code.
UPPERCASE	Uppercase letters indicate SQL statements and terms. For Windows, used to indicate database names, filenames, and similar terms.
[]	Brackets indicate optional items.
{ }	Curly brackets are used in the more complex syntax statements to show a required item.
...	An ellipsis indicates the continuation of previous lines of code or that the previous argument can be repeated.
	A vertical bar separates two mutually exclusive choices.

Platform conventions

This manual uses the following conventions—where necessary—to indicate that information is platform-specific:

Windows	All Windows platforms including Windows 3.1, Windows NT, Windows 95, and Windows 98
WinNT	Windows NT only
Win95	Windows 95 only
Win98	Windows 98 only
Win2000	Windows 2000 only
UNIX	All UNIX platforms
Solaris	Solaris only
Linux	Linux only
IBM-AIX	IBM-AIX only
HP-UX	HP-UX only

Where to find additional information

For more information about VisiBroker for Java, refer to these information sources:

- VisiBroker for Java *Release Notes* contain late-breaking information about the current release of VisiBroker for Java.
- VisiBroker for Java *Installation Guide*. This guide contains the instructions for installing VisiBroker for Java on Windows and UNIX, and information for system administrators who are deploying distributed applications built using VisiBroker.
- VisiBroker for Java *Reference*. This manual contains information on VisiBroker commands and Java interfaces.
- VisiBroker for Java *VisiBroker Gatekeeper Guide*. This guide provides information about how to configure and use the Gatekeeper.
- The Inprise web site also provides a variety of useful information for developers and others who are interested in evaluating our products and ORB technology. From the web site you can view information specific to developers using VisiBroker ORBs. This includes a section listing Frequently Asked Questions (FAQ) and their answers. Visit the Inprise web site at <http://www.borland.com/visibroker/>.

For more information about the CORBA specification, refer to *The Common Object Request Broker: Architecture and Specification*. This document is available from the Object Management Group and describes the architectural details of CORBA. You can access the CORBA specification at the OMG web site: <http://www.omg.org/>.

Contacting Inprise developer support

Inprise offers a variety of support options. These include free services on the Internet, where you can search our extensive information base and connect with other users of Inprise products. In addition, you can choose from several categories of telephone support, ranging from support on installation of the Inprise product to fee-based consultant-level support and detailed assistance.

For more information about Inprise developer support services, please see our web site at <http://www.borland.com/devsupport/>, call Inprise Assist at 800-523-7070, or contact our Sales Department at 800-632-2864. For customers outside of the United States of America, please see our web site at <http://www.borland.com/bww/intlcust.html>.

When you contact developer support, you will be asked to provide the following information:

- Your Access ID number
- Product name and version (for example, VisiBroker for C++, version 4.0)
- Operating system and version (for example, Windows NT Server 4.0 with Service Pack 5)

- Your desired priority (low, medium, high)
- Brief description of the problem
- Details of any error messages or exceptions raised

Basic Concepts

This part of the VisiBroker for Java *Programmer's Guide* includes these chapters.

- Chapter 2 "Understanding the CORBA model"
- Chapter 3 "Setting up your environment"
- Chapter 4 "Developing an example application with VisiBroker"
- Chapter 5 "Handling exceptions"

Understanding the CORBA model

This chapter introduces VisiBroker for Java, a complete implementation of the CORBA 2.3 specification. This chapter describes VisiBroker features and components.

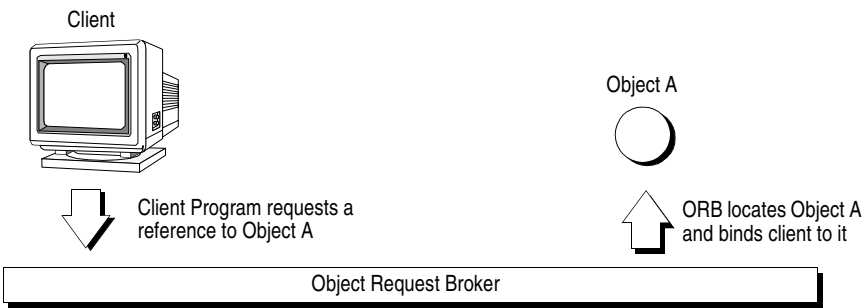
What is CORBA?

The Common Object Request Broker Architecture (CORBA) allows distributed applications to interoperate (application to application communication), regardless of what language they are written in or where these applications reside.

The CORBA specification was adopted by the Object Management Group to address the complexity and high cost of developing distributed object applications. CORBA uses an object-oriented approach for creating software components that can be reused and shared between applications. Each object encapsulates the details of its inner workings and presents a well defined interface, which reduces application complexity. The cost of developing applications is reduced, because once an object is implemented and tested, it can be used over and over again.

The Object Request Broker (ORB) in Figure 2.1 connects a client application with the objects it wants to use. The client program does not need to know whether the object implementation it is in communication with resides on the same computer or is located on a remote computer somewhere on the network. The client program only needs to know the object's name and understand how to use the object's interface. The ORB takes care of the details of locating the object, routing the request, and returning the result.

Figure 2.1 Client program acting on an object

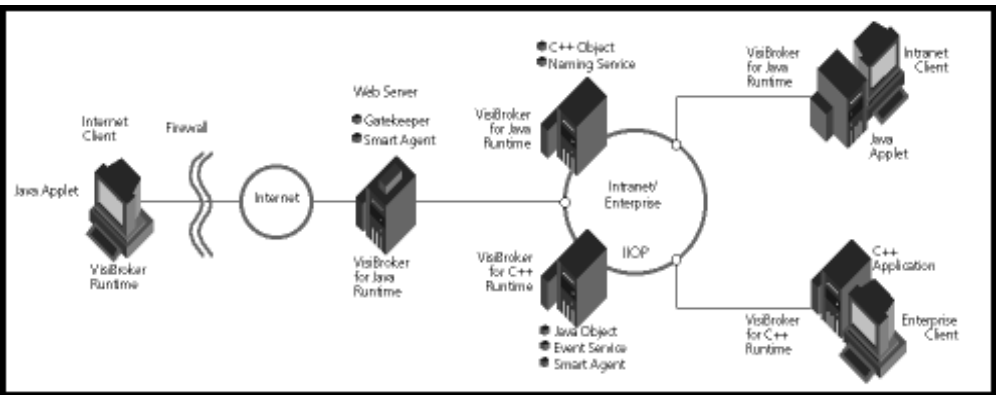


Note The ORB itself is not a separate process. It is a collection of libraries and network resources that integrates within end-user applications, and allows your client applications to locate and use objects.

What is VisiBroker?

VisiBroker for Java provides a complete CORBA 2.3 ORB runtime and supporting development environment for building, deploying, and managing distributed Java applications that are open, flexible, and inter-operable. Objects built with VisiBroker for Java are easily accessed by Web-based applications that communicate using OMG’s Internet Inter-ORB Protocol (IIOP) standard for communication between distributed objects through the Internet or through local intranets. VisiBroker has a built-in implementation of IIOP that ensures high-performance and inter-operability.

Figure 2.2 VisiBroker architecture



VisiBroker for Java features

VisiBroker for Java has several key features as described in the following sections.

VisiBroker Smart Agent architecture

VisiBroker's Smart Agent (`osagent`) is a dynamic, distributed directory service that provides facilities for both client applications and object implementations. Multiple Smart Agents on a network cooperate to provide load balancing and high availability for client access to server objects. The Smart Agent keeps track of objects that are available on a network, and locates objects for client applications at invocation time. VisiBroker can determine if the connection between your client application and a server object has been lost, due to an error such as a server crash or a network failure. When a failure is detected, an attempt is automatically made to connect your client to another server on a different host, if it is so configured. For details on the Smart Agent, see Chapter 16, "Using the Smart Agent," and in Chapter 10, "Using quality of service."

Enhanced object discovery with the Location Service

VisiBroker provides a powerful Location Service—an extension to the CORBA specification—that enables you to access the information from multiple Smart Agents. Working with the Smart Agents on a network, the Location Service can see all the available instances of an object to which a client can bind. Using *triggers*, a callback mechanism, client applications can be instantly notified of changes to an object's availability. Used in combination with *interceptors*, the Location Service is useful for developing enhanced load balancing of client requests to server objects. See Chapter 17, "Using the Location Service," for more information.

Implementation and object activation support

VisiBroker's Object Activation Daemon (OAD) can be used to automatically start object implementations when clients need to use them. Additionally, VisiBroker provides functionality that enables you to defer object activation until a client request is received. You can defer activation for a particular object or an entire class of objects on a server. See Chapter 7, "Using POAs," for more information on servant managers.

Robust thread and connection management

VisiBroker provides native support for single and multithreading thread management. With VisiBroker's thread-per-session model, threads are automatically allocated on the server-per-client connection to service multiple requests, and then are terminated when the connection ends. With the thread pooling model, threads are allocated based on the amount of request traffic to the server object. This means

that a highly active client will be serviced by multiple threads—ensuring that the requests are quickly executed—while less active clients can share a single thread, and still have their requests immediately serviced.

VisiBroker’s connection management minimizes the number of client connections to the server. All client requests for objects residing on the same server are multiplexed over the same connection, even if they originate from different threads. Additionally, released client connections are recycled for subsequent reconnects to the same server, eliminating the need for clients to incur the overhead of new connections to the same server.

All thread and connection behavior is fully configurable. See Chapter 8, “Managing threads and connections,” for details on how VisiBroker manages threads and connections.

IDL compilers

VisiBroker for Java comes with two IDL compilers that make object development easier,

- `idl2java`: The `idl2java` compiler takes IDL files as input and produces the necessary client stubs and server skeletons (in Java).
- `idl2ir`: The `idl2ir` compiler takes an IDL file and populates an interface repository with its contents.

See Chapter 15, “Using IDL,” and Chapter 21, “Using interface repositories,” for details on these compilers.

Dynamic invocation with DII and DSI

For dynamic invocation, VisiBroker provides implementations of both the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI). The DII allows client applications to dynamically create requests for objects that were not defined at compile time. The DII is covered in Chapter 22, “Using the Dynamic Invocation Interface.” The DSI allows servers to dispatch client operation requests to objects that were not defined at compile time. See Chapter 23, “Using the Dynamic Skeleton Interface,” for complete details.

Interface and implementation repositories

The Interface Repository (IR) is an online database of meta information about ORB objects. Meta information stored for objects includes information about modules, interfaces, operations, attributes, and exceptions. Chapter 21, “Using interface repositories,” covers how to start an instance of the Interface Repository, add information to an interface repository from an IDL file, and extract information from an interface repository.

The Implementation Repository is an online database of meta information about implementations of ORB objects. The Object Activation Daemon is VisiBroker's interface to the Implementation Repository that is used to automatically activate the implementation when a client references the object. See Chapter 20, "Using the Object Activation Daemon."

Server-side portability

VisiBroker supports the CORBA Portable Object Adapter (POA), which is a replacement to the Basic Object Adapter (BOA). The POA shares some of the same functionality as the BOA, such as activating objects, support for transient or persistent objects, and so forth. The POA also has new features, such as the POA Manager and Servant Manager which creates and manages instances of your objects. See Chapter 7, "Using POAs," for more information.

Customizing the ORB with interceptors and object wrappers

VisiBroker's interceptors enable developers to view under-the-cover communications between clients and servers. Interceptors can be used to extend the ORB with customized client and server code that enables load balancing, monitoring, or security to meet specialized needs of distributed applications. See Chapter 24, "Using interceptors," for information.

VisiBroker's object wrappers allow you to define methods that are called when a client application invokes a method on a bound object or when a server application receives an operation request. See Chapter 25, "Using object wrappers," for information.

Backing stores in the Naming Service

The new interoperable Naming Service integrates with pluggable backing stores to make its state persistent. This ensures easy fault tolerance and failover functionality in the Naming Service. See "Pluggable backing store" on page 18-14 for more information.

Web naming

The web naming feature allows you to associate a Uniform Resource Locator (URL) with an object, allowing references to that object to be obtained by specifying a URL. See Chapter 29, "Using URL naming," for more information.

Defining interfaces without IDL

VisiBroker’s `java2iiop` compiler lets you use the Java language to define interfaces instead of using the Interface Definition Language (IDL). You can use the `java2iiop` compiler if you have existing Java code that you wish to adapt to interoperate with CORBA distributed objects or if you do not wish to learn IDL. See Chapter 26, “Using RMI over IIOP,” for more details.

Gatekeeper (optional feature)

The VisiBroker Gatekeeper allows client programs to issue operation requests to objects that reside on a web server and to receive callbacks from those objects, all the while conforming to the security restrictions imposed by web browsers. The Gatekeeper also handles communication through firewalls and can be used as an HTTP daemon.

VisiBroker CORBA compliance

VisiBroker for Java is fully compliant with the CORBA specification (version 2.3) from the Object Management Group (OMG). For more details, refer to the CORBA specification located at <http://www.omg.org/>.

VisiBroker development environment

VisiBroker for Java is used in both the development and deployment phases. The VisiBroker development environment includes the following components:

- Administration and programming tools
- VisiBroker ORB

Programmer’s tools

The following tools are used during the development phase:

Tool	Purpose
<code>idl2ir</code>	This tool allows you to populate an interface repository with interfaces defined in an IDL file.
<code>idl2java</code>	This tool generates Java stubs and skeletons from an IDL file.
<code>java2iiop</code>	Generates Java stubs and skeletons from a Java file. This tool allows you to define your interfaces in Java, rather than in IDL.
<code>java2idl</code>	Generates an IDL file from a file containing Java bytecode.

CORBA services tools

The following tools are used to administer the VisiBroker ORB during development:

Tool	Purpose
irep	Used to manage the Interface Repository. See Chapter 21, “Using interface repositories.”
oad	Used to manage the Object Activation Daemon (OAD). See Chapter 20, “Using the Object Activation Daemon.”
nameserv	Used to start an instance of the Naming Service. See Chapter 18, “Using the Naming Service.”

Administration tools

The following tools are used to administer the VisiBroker ORB during development:

Tool	Purpose
oadutil list	Lists ORB object implementations registered with the OAD.
oadutil reg	Registers an ORB object implementation with the OAD.
oadutil unreg	Unregisters an ORB object implementation with the OAD.
osagent	Used to manage the Smart Agent. See Chapter 16, “Using the Smart Agent.”
osfind	Reports on objects running on a given network.

Java Development Environment

A Java development environment, such as Inprise JBuilder, is required for developing applets or applications that use the VisiBroker ORB. JavaSoft’s Java Developer’s Kit (JDK) also includes a Java runtime environment.

Sun Microsystems has made JavaSoft’s JDK—including its Java runtime environment—available for Solaris, Windows NT, and Windows 95 platforms. You can download the JDK from Sun Microsystems’ web site: <http://java.sun.com/>.

The JDK has also been ported to IBM AIX, OS/2, SGI IRIX, and HP-UX. These other versions can be downloaded from the respective hardware vendor’s web site. To see what is available for various platforms, visit Sun Microsystems’ JavaSoft web site: <http://java.sun.com/products/jdk/jdk-ports.html>.

Java Runtime Environment

A Java runtime environment is required for any end user who wishes to execute a Java application. A Java runtime environment is an engine that interprets and executes a Java application. Typically, Java runtime environments are bundled with Java development environments. See “Java Runtime Environment” on page 2-7 for details.

What's Required for VisiBroker?

In order to use certain tools and functionality in VisiBroker, specific versions of the JDK are required. The following section explains what is needed.

JDK 1.1 or later is required for the following functionality:

- Runtime use of UDP broadcasting to locate a Smart Agent. UDP broadcasting enables you to locate a Smart Agent anywhere on the LAN. Otherwise, you must set the `vbroker.agent.addrFile` property to locate the Smart Agent.
- Gatekeeper.
- Ability to tune some parameters of the TCP sockets.
- Compilers (`java2iiop` and `java2idl`).
- Runtime support for pass-by-value.
- OAD.

JDK 1.1.8 or later is required for the following functionality:

- To use Object by Value and RMI/IIOP.

Java-enabled Web Browser

Applets can be run in any Java-enabled web browser—such as Netscape Communicator, Netscape Navigator, or Microsoft's Internet Explorer. You can obtain these Java-enabled web browsers by navigating to one of the following URLs:

- <http://www.netscape.com/>
- <http://microsoft.com/ie/>

Interoperability with VisiBroker for C++

Applications created with VisiBroker for Java can communicate with object implementations developed with VisiBroker for C++, which is sold separately. Simply use the same IDL you used to develop your Java application as input to the VisiBroker IDL compiler, supplied with VisiBroker for C++. You may then use the resulting C++ skeletons to develop the object implementation.

Also, object implementations written with VisiBroker for Java will work with clients written in VisiBroker for C++. In fact, a server written with VisiBroker for Java will work with *any* CORBA-compliant client; a client written with VisiBroker for Java will work with *any* CORBA-compliant server.

Interoperability with other ORB products

CORBA-compliant software objects communicate using the Internet Inter-ORB Protocol (IIOP) and are fully interoperable, even when they are developed by different vendors who have no knowledge of each other's implementations. VisiBroker's use of IIOP allows client and server applications you develop with VisiBroker to interoperate with a variety of ORB products from other vendors.

IDL to Java mapping

VisiBroker for Java conforms with the *OMG IDL/Java Language Mapping Specification*. See the *VisiBroker for Java Reference* for a summary of VisiBroker's current IDL to Java language mapping, as implemented by the `idl2java` compiler. For each IDL construct there is a section that describes the corresponding Java construct, along with code samples.

For more information about the mapping specification, refer to the *OMG IDL/Java Language Mapping Specification*.

Setting up your environment

Before using VisiBroker, you must set several environment variables.

Setting the Path environment variable

Note The `PATH` environment variable is set automatically during installation to include the `bin` directory of the VisiBroker distribution. Installation instructions are provided in the *VisiBroker Installation Guide*.

If you choose to explicitly set the `PATH` environment variable, the following sections explain how to do so.

Updating the PATH on a Windows platform

Assuming that the VisiBroker distribution was installed in `c:\`, you can set your `PATH` with the following DOS command. Alternatively, you may want to set the `PATH` in your `autoexec.bat` file.

```
prompt> set PATH=c:\inprise\vbroker\bin;%PATH%
```

Updating the PATH on a Windows NT platform

Although the DOS `set` command can be used to set environment variables in Windows NT, you may find it easier to use the System control panel to automatically set the `PATH`. Assuming that the VisiBroker distribution is installed in `c:\inprise\vbroker`, open the System control panel, choose “PATH” as the variable to edit, and add the following to the `PATH`:

```
c:\inprise\vbroker\bin;
```

Changes made to environment variables with the System control panel will not be reflected in currently running applications, but all subsequently launched applications and DOS prompts will use the new settings.

Setting the Path on a UNIX platform

If you are using `csh` and you installed VisiBroker in `/usr/local/vbroker`, you can update the `PATH` environment using the following command:

```
prompt> setenv PATH /usr/local/vbroker/bin:$PATH
```

If you are using the Bourne shell and you have installed VisiBroker in `/usr/local/vbroker` you can update the `PATH` environment variable using the following commands:

```
prompt> PATH=$PATH:/usr/local/vbroker/bin
prompt> export PATH
```

CLASSPATH

This environment variable defines the location of the various Java packages used on your system. The `CLASSPATH` variable does not need to be set when installing or configuring VisiBroker.

Setting the VBROKER_ADM environment variable

The `VBROKER_ADM` environment variable defines the administration directory where important configuration information for VisiBroker's interface repository, Object Activation Daemon, and Smart Agent are stored.

Setting VBROKER_ADM on a Windows platform

The `VBROKER_ADM` environment variable is set in the Windows registry when you install VisiBroker. You can change the registry setting by using the `vregedit` tool.

You can override the registry by setting the `VBROKER_ADM` environment variable. Assuming you want your own directory `c:\my\adm` to be used, you could set the `VBROKER_ADM` environment variable as follows:

```
prompt> set VBROKER_ADM=c:\my\adm
```

Setting VBROKER_ADM on a UNIX platform

If you are using `csh` and you installed VisiBroker in `/usr/local`, set the `VBROKER_ADM` environment variable as follows:

```
prompt> setenv VBROKER_ADM /usr/local/vbroker/adm
```

If you are using the Bourne shell and you installed VisiBroker in `/usr/local`, set the `VBROKER_ADM` environment variable as follows:

```
prompt> VBROKER_ADM=/usr/local/vbroker/adm
prompt> export VBROKER_ADM
```

Setting the OSAGENT_PORT environment variable

The `OSAGENT_PORT` environment variable defines the port number under which the Smart Agent will listen. Although you can set the port number to any value from 5000 to 65355, by default, the Smart Agent listens on port number 14000.

The `OSAGENT_PORT` variable is automatically set in the Windows registry when you install VisiBroker. You can change the registry setting by using the `vregedit` tool.

You can override the registry by setting the `OSAGENT_PORT` environment variable. Assuming you want the Smart Agent to listen on port number 10000, you could set the `OSAGENT_PORT` environment variable as follows:

```
prompt> set OSAGENT_PORT=10000
```

If you are using `csh` and you want the Smart Agent to listen on port number 10000, set the `OSAGENT_PORT` environment variable as follows:

```
prompt> setenv OSAGENT_PORT 10000
```

If you are using the Bourne shell and you want the Smart Agent to listen on port number 10000, set the `OSAGENT_PORT` environment variable as follows:

```
prompt> OSAGENT_PORT=10000
prompt> export OSAGENT_PORT
```

Logging output

Many VisiBroker tools offer a verbose mode that displays information about the tool as it executes. In addition, any application that is linked with the VisiBroker library may also produce output. On UNIX systems, this output is written to the console. On Windows systems, this output is written to one of several log files.

Table 3.1 summarizes the names of the various log files that may be produced on Windows.

Table 3.1 Summary of log files produced on Windows platforms

File Name	Description
<code>oad.log</code>	Produced by Object Activation Daemon when started with the <code>-v</code> flag.
<code>osagent.log</code>	Produced by the Smart Agent when started with the <code>-v</code> flag.

The location of these log files is determined by the following rules:

- 1 An attempt will be made to write the file to the log directory within the directory pointed to by the `VBROKER_ADM` variable. The following example shows how to set the `VBROKER_ADM` variable on Windows to a specific directory:

```
SET VBROKER_ADM=c:\my\adm\log
```

- 2 If the application or tool does not have write permission for this directory, an attempt will then be made to write the file to the directory `\inprise\vbroker\log` on the drive from which the application or tool was started.

If step 2 fails, an attempt is made to write the file to the current directory.

Developing an example application with VisiBroker

This chapter uses an example application to describe the development process for creating distributed, object-based applications.

The code for the example application is provided in the `bank_agent.html` file under the `examples/basic/bank_agent` directory where the VisiBroker for Java package was installed. If you do not know the location of the VisiBroker for Java package, see your system administrator.

Development process

When you develop distributed applications with VisiBroker, you must first identify the objects required by the application. You will then usually follow these steps:

- 1 Write a specification for each object using the Interface Definition Language (IDL).

IDL is the language that an implementer uses to specify the operations that an object will provide and how they should be invoked. In this example, we define, in IDL, the `Account` interface with a `balance()` method and the `AccountManager` interface with an `open()` method.

- 2 Use the IDL compiler to generate the client stub code and server POA servant code.

Using the `idl2java` compiler, we'll produce client-side stubs (which provide the interface to the `Account` and the `AccountManager` objects' methods) and server-side classes (which provides classes for the implementation of the remote objects).

- 3 Write the client program code.

To complete the implementation of the client program, initialize the ORB, bind to the `Account` and the `AccountManager` objects, invoke the methods on these object, and print out the balance.

4 Write the server object code.

To complete the implementation of the server object code, we must derive from the `AccountPOA` and `AccountManagerPOA` classes, provide implementations of the interface's methods, and implement the server's `main` routine.

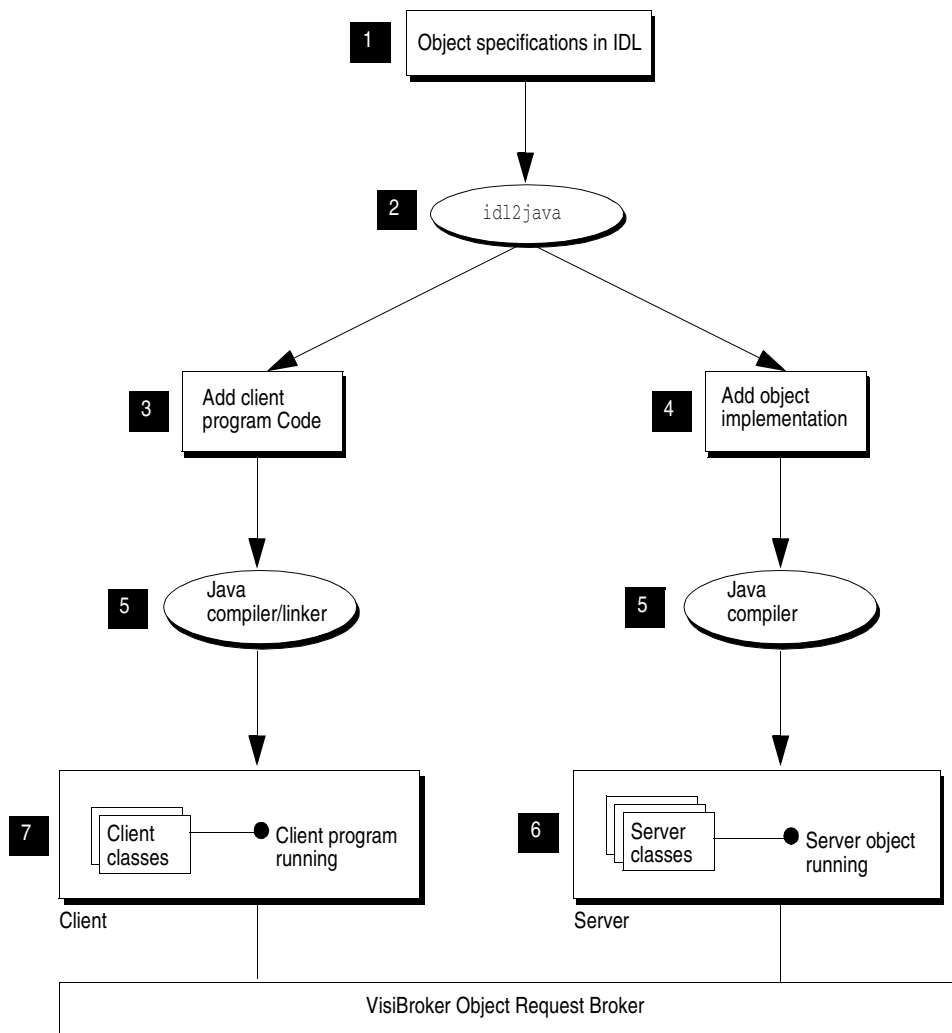
5 Compile the client and server code.

To create the client program, compile the client program code with the client stub. To create the `Account` server, compile the server object code with the server skeleton.

6 Start the server.

7 Run the client program.

Figure 4.1 Developing the sample bank application



Step 1: Defining object interfaces

The first step to creating an application with VisiBroker is to specify all of your objects and their interfaces using the OMG's Interface Definition Language (IDL). The IDL can be mapped to a variety of programming languages. The IDL mapping for Java is summarized in the VisiBroker for Java *Reference*.

You then use the `idl2java` compiler to generate stub routines and servant code from the IDL specification. The stub routines are used by your client program to invoke operations on an object. You use the servant code, along with code you write, to create a server that implements the object. The code for the client and object, once completed, is used as input to your Java compiler to produce a client Java applet or application and an object server.

Writing the account interface in IDL

IDL has a syntax similar to C++ and can be used to define modules, interfaces, data structures, and more.

IDL sample 4.1 shows the contents of the `Bank.idl` file for the `bank_agent` example. The `Account` interface provides a single method for obtaining the current balance. The `AccountManager` interface creates an account for the user if one does not already exist.

IDL sample 4.1 `Bank.idl` file provides the `Account` interface definition

```
module Bank{
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

Step 2: Generating client stubs and server servants

The interface specification you create in IDL is used by VisiBroker's `idl2java` compiler to generate Java classes for the client program, and skeleton code for the object implementation. The Java classes are used by the client program for all method invocations. You use the skeleton code, along with code you write, to create the server that implements the objects.

The code for the client program and server object, once completed, is used as input to your Java compiler to produce the client and server executables classes. These steps are shown in Figure 4.1 on page 4-2.

Because the `Bank.idl` file requires no special handling, it can be compiled with the following command.

```
prompt> idl2java Bank.idl
```

For more information on the command-line options for the `idl2java` compiler, see “Using IDL” on page 15-1.

Files produced by the idl compiler

Because Java allows only one public interface or class per file, compiling the IDL file will generate several `.java` files. These files are stored in a generated sub-directory called `Bank` which is the module name specified in the IDL and is the package to which the generated files belong. The following files are generated:

- `_AccountManagerStub.java`: Stub code for the `AccountManager` object on the client side.
- `_AccountStub.java`: Stub code for the `Account` object on the client side.
- `Account.java`: The `Account` interface declaration.
- `AccountHelper.java`: Declares the `AccountHelper` class, which defines helpful utility methods.
- `AccountHolder.java`: Declares the `AccountHolder` class, which provides a holder for passing `Account` objects.
- `AccountManager.java`: The `AccountManager` interface declaration.
- `AccountManagerHelper.java`: Declares the `AccountManagerHelper` class, which defines helpful utility methods.
- `AccountManagerHolder.java`: Declares the `AccountManagerHolder` class, which provides a holder for passing `AccountManager` objects.
- `AccountManagerOperation.java`: This interface provides declares the method signatures defined in the `AccountManager` interface in the `Bank.idl` file.
- `AccountManagerPOA.java`: POA servant code (implementation base code) for the `AccountManager` object implementation on the server side.
- `AccountManagerPOATie.java`: Class used to implement the `AccountManager` object on the server side using the tie mechanism, described in Chapter 9, “Using the tie mechanism.”
- `AccountOperations.java`: This interface provides declares the method signatures defined in the `Account` interface in the `Bank.idl` file
- `AccountPOA.java`: POA servant code (implementation base code) for the `Account` object implementation on the server side.
- `AccountPOATie.java`: Class used to implement the `Account` object on the server side using the tie mechanism, described in Chapter 9, “Using the tie mechanism.”

For more information about the Helper, Holder, and Operations classes, see the *VisiBroker for Java Reference*.

Step 3: Implementing the client

Many of the classes used in implementing the bank client are contained in the `Bank` package generated by the `idl2java` compiler as shown in the previous example. The `Client.java` file illustrates this example and is included in the `bank_agent` directory. Normally you would create this file.

Client.java

The `Client` class implements the client application which obtains the current balance of a bank account. The bank client program performs these steps:

- 1 Initializes the ORB.
- 2 Binds to an `AccountManager` object.
- 3 Obtains an `Account` object by invoking `open` on the `AccountManager` object.
- 4 Obtains the balance by invoking `balance` on the `Account` object.

Code sample 4.1 Client side program

```
public class Client {
    public static void main(String[] args) {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // Get the manager Id
        byte[] managerId = "BankManager".getBytes();
        // Locate an account manager. Give the full POA name and the servant ID.
        Bank.AccountManager manager =
            Bank.AccountManagerHelper.bind(orb, "/bank_agent_poa", managerId);
        // use args[0] as the account name, or a default.
        String name = args.length > 0 ? args[0] : "Jack B. Quick";
        // Request the account manager to open a named account.
        Bank.Account account = manager.open(name);
        // Get the balance of the account.
        float balance = account.balance();
        // Print out the balance.
        System.out.println("The balance in " + name + "'s account is $" + balance);
    }
}
```

Binding to the AccountManager object

Before your client program can invoke the `open(String name)` method, it must first use the `bind()` method to establish a connection to the server that implements the `AccountManager` object. The implementation of the `bind()` method is generated automatically by the `idl2java` compiler. The `bind()` method requests the ORB to locate and establish a connection to the server. If the server is successfully located and a connection is established, a proxy object is created to represent the server's `AccountManagerPOA` object. An object reference to the `AccountManager` object is returned to your client program.

Obtaining an Account object

Next your client class needs to call the `open()` method on the `AccountManager` object to get an object reference to the `Account` object for the specified customer name.

Obtaining the balance

Once your client program has established a connection with an `Account` object, the `balance()` method can be used to obtain the balance. The `balance()` method on the client side is actually a stub generated by the `idl2java` compiler that gathers all the data required for the request and sends it to the server object.

AccountManagerHelper.java

This file is located in the `Bank` package. It contains an `AccountManagerHelper` object and defines several methods for binding to the server that implements this object. The `bind()` class method contacts the specified POA manager to resolve the object. Our example application uses the version of the `bind` method that accepts an object name, but the client may optionally specify a particular host and special bind options. For more information about Helper classes, see the *VisiBroker for Java Reference*.

Code sample 4.2 Portion of the `AccountManagerHelper.java` file

```
package Bank;
public final class AccountManagerHelper {
    . . .
    public static Bank.AccountManager bind(org.omg.CORBA.ORB orb) {
        return bind(orb, null, null, null);
    }
    . . .
}
```

Other methods

Several other methods are provided that allow your client program to manipulate an `AccountManager` object reference. Many of these are not used in the example client application, but they are described in detail in the *VisiBroker for Java Reference*.

Step 4: Implementing the server

Just as with the client, many of the classes used in implementing the bank server are contained in the `Bank` package generated by the `idl2java` compiler. The `Server.java` file is a server implementation included for the purposes of illustrating this example. Normally you, the programmer, would create this file.

Server.java

This file implements the Server class for the server side of our banking example. The server program does the following:

- Initializes the Object Request Broker.
- Creates a Portable Object Adapter with the required policies.
- Creates the account manager servant object.
- Activates the servant object.
- Activates the POA manager (and the POA).
- Waits for incoming requests.

Code sample 4.3 Server side program

```
public class Server {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // get a reference to the root POA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Create policies for our persistent POA
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
            };
            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA( "bank_agent_poa", rootPOA.the_POAManager(),
                policies );
            // Create the servant
            AccountManagerImpl managerServant = new AccountManagerImpl();
            // Decide on the ID for the servant
            byte[] managerId = "BankManager".getBytes();
            // Activate the servant with the ID on myPOA
            myPOA.activate_object_with_id(managerId, managerServant);
            // Activate the POA manager
            rootPOA.the_POAManager().activate();
            System.out.println(myPOA.servant_to_reference(managerServant) + " is ready.");
            // Wait for incoming requests
            orb.run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Step 5: Building the example

The `examples` directory of your VisiBroker release contains a `vbmake.bat` for this example and other VisiBroker examples.

Compiling the example

Windows Assuming the VisiBroker distribution was installed in `C:\vbroker`, type the following to compile the example:

```
prompt> C:
prompt> cd vbroker\examples\basic\bank_agent
prompt> vbmake
```

The command `vbmake` is a batch file which runs the `idl2java` compiler and then compiles each file.

If you encounter some problems while running `vbmake`, check that your path environment variable points to the `bin` directory where you installed the VisiBroker software.

UNIX Assuming the VisiBroker distribution was installed in `/usr/local`, type the following to compile the example:

```
prompt> cd /usr/local/vbroker/examples/basic/bank_agent
prompt> make java
```

In this example, `make` is the standard UNIX facility. If you do not have it in your `PATH`, see your system administrator.

Step 6: Starting the server and running the example

Now that you have compiled your client program and server implementation, you are ready to run your first VisiBroker application.

Starting the Smart Agent

Before you attempt to run VisiBroker client programs or server implementations, you must first start the Smart Agent on at least one host in your local network.

The basic command for starting the Smart Agent is as follows:

```
prompt> osagent
```

If you're running Windows NT and you want to start the Smart Agent as an NT Service, you need to register the ORB Services as NT Services during installation. See the VisiBroker *Installation Guide* for instructions on installing this product. If the Services were registered, you then are able to start the Smart Agent as an NT Service through the Services Control Panel.

The Smart Agent is described in detail in Chapter 16, "Using the Smart Agent."

Starting the server

Open a DOS prompt window and start your server by using the following DOS command:

```
prompt> start vbj Server
```

UNIX Start your Account server by typing

```
prompt> vbj Server&
```

Running the client

Windows Open a separate DOS prompt window and start your client by using the following DOS command:

```
prompt> vbj Client
```

UNIX To start your client program, type

```
prompt> vbj Client
```

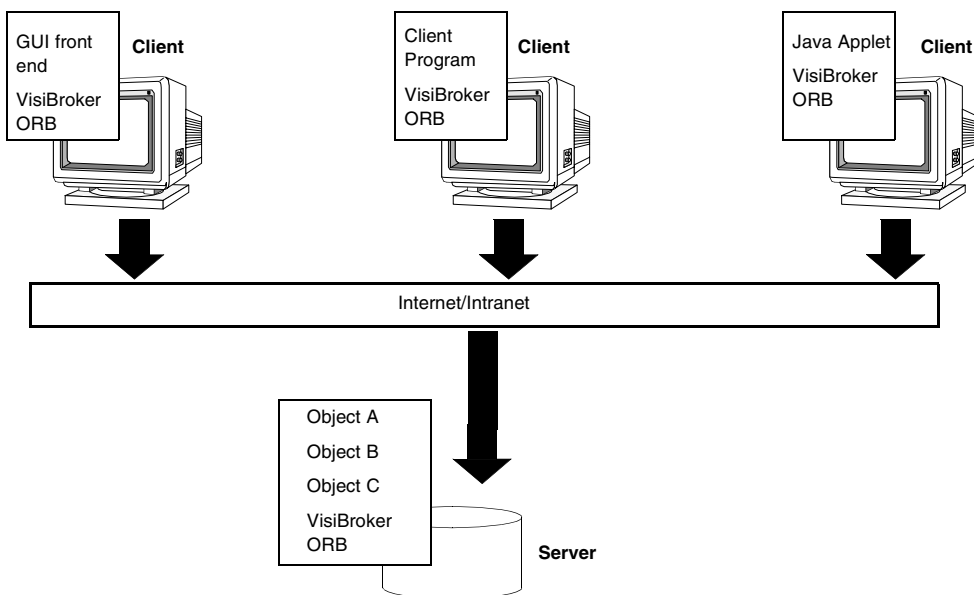
You should see output similar to that shown below (the account balance is computed randomly).

```
The balance in the account is $168.38.
```

Deploying applications with VisiBroker

VisiBroker is also used in the deployment phase. This phase occurs when a developer has created client programs or server applications that have been tested and are ready for production. At this point a system administrator is ready to deploy the client programs on end-users' desktops or server applications on server-class machines.

For deployment, the VisiBroker ORB supports client programs on the front end. You must install the ORB on each machine that runs the client program. Clients (that make use of the ORB) on the same host share the ORB. The VisiBroker ORB also supports server applications on the middle tier. You must install the full ORB on each machine that runs the server application. Server applications or objects (that make use of the ORB) on the same server machine share the ORB. Clients may be GUI front ends, applets, or client programs. Server implementations contain the business logic on the middle tier.

Figure 4.2 Client and server programs deployed with VisiBroker ORBs

VisiBroker for Java applications

Deploying applications

In order to deploy applications developed with VisiBroker for Java, you must first set up a runtime environment on the host where the application is to be executed and ensure that the necessary support services are available on the local network.

The runtime environment required for applications developed with VisiBroker for Java includes these components,

- Java Runtime Environment.
- VisiBroker Java packages archived in the `vbjorb.jar` file, located in the `lib` subdirectory where you installed VisiBroker.
- Availability of the support services required by the application.

A Java Runtime Environment must be installed on the host where the deployed application is to execute, and the VisiBroker Java packages must be installed on the host where the deployed application is to execute.

Environment variables

When you use the `vbj` executable, the environmental variables are automatically set up for you.

If the deployed application is to use a Smart Agent (`osagent`) on a particular host, you must set the `OSAGENT_ADDR` environment variable before running the application. You can use the `vbroker.agent.addr` property as a command-line argument to specify a hostname or IP address, as described in Table 4.1.

If the deployed application is to use a particular UDP port when communicating with a Smart Agent (`osagent`), you must set the `OSAGENT_PORT` environment variable before running the application. You can use the `vbroker.agent.port` command-line argument to specify the UDP port number.

For more information about environment variables, see Chapter 3, “Setting up your environment.”

Support service availability

A Smart Agent (`osagent`) must be executing somewhere on the network where the deployed application is to be executed. Depending on the requirements of the application being deployed, you may need to ensure that other VisiBroker runtime support services are available, as well. These services include:

Support services	Needed when:
Object Activation Daemon (<code>oad</code>)	A deployed application is a server that implements object which needs to be started on demand.
Interface Repository (<code>irep</code>)	A deployed application uses either the dynamic skeleton interface or dynamic implementation interface. See Chapter 21, “Using interface repositories,” for a description of these interfaces.
Gatekeeper	A deployed application needs to execute in an environment that uses firewalls for network security.

Using `vbj`

You can use the `vbj` command to start your application and enter command-line arguments that control the behavior of your application.

```
vbj -Dvbroker.agent.port=10000 <class>
```

Executing client applications

A client application is one that uses ORB objects, but does not offer any ORB objects of its own to other client applications. A client may be started with the `vbj` command, or from within a Java-enabled web browser.

The following table summarizes the command-line arguments that may be specified for a client application.

Table 4.1 Command-line arguments for client applications

Options	Description
-DORBagentAddr=<hostname ip_address>	Specifies the hostname or IP address of the host running the Smart Agent this client should use. If a Smart Agent is not found at the specified address or if this option is not specified, broadcast messages will be used to locate a Smart Agent.
-DORBagentPort= <port_number>	Specifies the port number of the Smart Agent. This option can be useful if multiple ORB domains are required. If not specified, a default port number of 14000 will be used.
-DORBmbufSize= <buffer_size>	Specifies the size of the intermediate buffer used by VisiBroker for operation request processing. To improve performance, the ORB does more complex buffer management than in previous versions of VisiBroker. The default size of send and receive buffers is 4 K. If data sent or received is larger than the default, new buffers will be allocated for each request/reply. If your application frequently sends data larger than 4 K and you wish to take advantage of buffer management, you may use this system property to specify a larger number of bytes for a default buffer size.
-DORBtcpNoDelay=<false true>	When set to true, all network connections will send data immediately. The default is false, which allows a network connection to send data in batches, as the buffer fills.
-DORBconnectionMax=<number>	Specifies the maximum number of connections allowed for an object implementation when OAid TSession is selected. If you do not specify, the default is unlimited.
-DORBconnectionMaxIdle= <number>	Specifies the number of milliseconds which a network connection can be idle before being shutdown by VisiBroker. By default, this is set to 0 which means that connections will never time-out. This option should be set for Internet applications.

Executing server applications

A server application is one that offers one or more ORB objects to client applications. A server application may be started with the `vbj` command or it may be activated by the Object Activation Daemon (`oad`).

The following table summarizes the command-line arguments that may be specified for a server application.

Options	Description
-DOAipAddr <hostname ip_address>	Specifies the hostname or IP address to be used for the Object Adaptor. Use this option if your host has multiple network interfaces and the BOA is associated with only one of those interfaces. If no option is specified, the host's default address is used.
-DOAport <port_number>	Specifies the port number to be used by the object adaptor when listening for a new connection.
-DOAid <TPool TSession>	Specifies the thread policy to be used by the BOA. The default is TPool unless you are in backward compatibility mode; if you are in backward compatibility, the default is TSession.
-DOAthreadMax <#>	Specifies the maximum number of threads allowed when OAid TPool is selected. If you do not specify or you Specify 0, this selects unlimited number of threads or, to be more precise, a number of threads limited only by your system resources.
-DOAthreadMin <#>	Specifies the minimum number of threads available in the thread pool. If you do not specify, the default is zero. You can specify this only when OAid TPool is selected.
-DOAthreadMaxIdle	This specifies the time which a thread can exist without servicing any requests. Threads that idle beyond the time specified can be returned to the system. By default, this is set to 300 seconds.
-DOAconnectionMax <#>	Specifies the maximum number of connections allowed when OAid TSession is selected. If you do not specify, the default is unlimited.
-DOAconnectionMaxIdle	This specifies the time which a connection can idle without any traffic. Connections that idle beyond this time can be shutdown by VisiBroker. By default, this is set to 0 which means that connections will never automatically time-out. This option should be set for Internet applications.

Handling exceptions

Exceptions in the CORBA model

The exceptions in the CORBA model include both system and user exceptions. The CORBA specification defines a set of *system* exceptions that can be raised when errors occur in the processing of a client request. Also, system exceptions are raised in the case of communication failures. System exceptions can be raised at any time and they do not need to be declared in the interface. You can define *user* exceptions in IDL for objects you create and specify the circumstances under which those exceptions are to be raised. They are included in the method signature. If an object raises an exception while handling a client request, the ORB is responsible for reflecting this information back to the client.

System exceptions

System exceptions are usually raised by the ORB, though it is possible for object implementations to raise them through interceptors discussed in Chapter 24, “Using interceptors.” When the ORB raises a `SystemException`, it will be one of the CORBA-defined error conditions shown in Table 5.1.

Table 5.1 CORBA-defined system exceptions

Exception name	Description
BAD_CONTEXT	Error processing context object.
BAD_INV_ORDER	Routine invocations out of order.
BAD_OPERATION	Invalid operation.
BAD_PARAM	An invalid parameter was passed.
BAD_TYPECODE	Invalid typecode.
COMM_FAILURE	Communication failure.
DATA_CONVERSION	Data conversion error.

Table 5.1 CORBA-defined system exceptions (continued)

Exception name	Description
FREE_MEM	Unable to free memory.
IMP_LIMIT	Implementation limit violated.
INITIALIZE	ORB initialization failure.
INTERNAL	ORB internal error.
INTF_REPOS	Error accessing interface repository.
INV_FLAG	Invalid flag was specified.
INV_INDENT	Invalid identifier syntax.
INV_OBJREF	Invalid object reference specified.
MARSHAL	Error marshalling parameter or result.
INVALID_TRANSACTION	Specified transaction was invalid (used in conjunction with ITS/OTS).
NO_IMPLEMENT	Operation implementation not available.
NO_MEMORY	Dynamic memory allocation failure.
NO_PERMISSION	No permission for attempted operation.
NO_RESOURCES	Insufficient resources to process request.
NO_RESPONSE	Response to request not yet available.
OBJ_ADAPTOR	Failure detected by object adaptor.
OBJECT_NOT_EXIST	Object is not available.
PERSIST_STORE	Persistent storage failure.
TRANSIENT	Transient failure.
TRANSACTION_REQUIRED	Transaction is required (used in conjunction with ITS/OTS).
TRANSACTION_ROLLEDBACK	Transaction was rolled back(used in conjunction with ITS/OTS).
TIMEOUT	Request timeout.
UNKNOWN	Unknown exception.

Code sample 5.1 SystemException class

```
public abstract class org.omg.CORBA.SystemException extends java.lang.RuntimeException {
    protected SystemException(java.lang.String reason,
        int minor, CompletionStatus completed) { . . . }
    public String toString() { . . . }
    public CompletionStatus completed;
    public int minor;
}
```

Obtaining completion status

System exceptions have a completion status that tells you whether or not the operation that raised the exception was completed. The `CompletionStatus` enumerated values are shown following. `COMPLETED_MAYBE` is returned when the status of the operation cannot be determined.

IDL sample 5.1 CompletionStatus values

```
enum CompletionStatus {
    COMPLETED_YES = 0;
    COMPLETED_NO = 1;
    COMPLETED_MAYBE = 2;
};
```

Catching system exceptions

Your applications should enclose the ORB and remote calls in a try catch block. Code sample 5.2 illustrates how the account client program, discussed in Chapter 4, “Developing an example application with VisiBroker,” prints an exception.

Code sample 5.2 Printing an exception

```
public class Client {
    public static void main(String[] args) {
        try {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            byte[] managerId = "BankManager".getBytes();
            Bank.AccountManager manager =
                Bank.AccountManagerHelper.bind(orb, "/bank_agent_poa", managerId);
            String name = args.length > 0 ? args[0] : "Jack B. Quick";
            Bank.Account account = manager.open(name);
            float balance = account.balance();
            System.out.println("The balance in " + name + "'s account is $" + balance);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

If you were to execute the client program with these modifications and without a server present, the following output would indicate that the operation did not complete and the reason for the exception.

```
prompt>vbj Client
org.omg.CORBA.OBJECT_NOT_EXIST:
    Could not locate the following POA:
    poa name : /bank_agent_poa
    minor code: 0    completed: No
```

Downcasting exceptions to a system exception

You can modify the account client program to attempt to downcast any exception that is caught to a `SystemException`. Code sample 5.3 shows how you might modify the client program. Code sample 5.4 shows how the output would appear if a system exception occurred.

Code sample 5.3 Downcasting an exception to a system exception

```

public class Client {
    public static void main(String[] args) {
        try {
            // Initialize the ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // Bind to an account
            Account account = AccountHelper.bind(orb, "/bank_poa",
                "BankAccount".getBytes());
            // Get the balance of the account
            float balance = account.balance();
            // Print the account balance
            System.out.println("The account balance is $" + balance);
            catch(Exception e) {
                if (e instanceof org.omg.CORBA.SystemException) {
                    System.err.println("System Exception occurred:");
                } else {
                    System.err.println("Not a system exception");
                }
                System.err.println(e);
            }
        }
    }
}

```

Code sample 5.4 Output from the system exception

```

System Exception occurred:
in thread "main" org.omg.CORBA.OBJECT_NOT_EXIST minor code: 0 completed: No

```

Catching specific types of system exceptions

Rather than catching all types of exceptions, you may choose to specifically catch each type of exception that you expect. Code sample 5.5 shows this technique.

Code sample 5.5 Catching specific types of exceptions

```

public class Client {
    public static void main(String[] args) {
        try {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            byte[] managerId = "BankManager".getBytes();
            Bank.AccountManager manager =
                Bank.AccountManagerHelper.bind(orb, "/bank_agent_poa", managerId);
            String name = args.length > 0 ? args[0] : "Jack B. Quick";
            Bank.Account account = manager.open(name);
            float balance = account.balance();
            System.out.println("The balance in " + name + "'s account is $" + balance);
        } catch(org.omg.CORBA.SystemException e) {
            System.err.println("System Exception occurred:");
            System.err.println(e);
        }
    }
}

```

User exceptions

When you define your object's interface in IDL you can specify the user exceptions that the object may raise. Code sample 5.6 shows the `UserException` code from which the `idl2java` compiler will derive the user exceptions you specify for your object.

Code sample 5.6 UserException class

```
public abstract class UserException extends java.lang.Exception {
    protected UserException();
    protected UserException(String reason);
}
```

Defining user exceptions

Suppose that you want to enhance the account application, introduced in Chapter 4, "Developing an example application with VisiBroker," so that the `account` object will raise an exception. If the `account` object has insufficient funds, you want a user exception named `AccountFrozen` to be raised. The additions required to add the user exception to the IDL specification for the `Account` interface are shown in bold.

IDL sample 5.2 Defining user exceptions

```
// Bank.idl
module Bank {
    interface Account {
        exception AccountFrozen {
        };
        float balance() raises(AccountFrozen);
    };
};
```

The `idl2java` compiler will generate the following code for a `AccountFrozen` exception class.

Code sample 5.7 AccountFrozen class generated by the idl compiler

```
package Bank;
public interface Account extends com.inprise.vbroker.CORBA.Object,
    Bank.AccountOperations, org.omg.CORBA.portable.IDLEntity {
}

package Bank;
public interface AccountOperations {
    public float balance () throws Bank.AccountPackage.AccountFrozen;
}

package Bank.AccountPackage;
public final class AccountFrozen extends org.omg.CORBA.UserException {
    public AccountFrozen () { . . . }
    public AccountFrozen (java.lang.String _reason) { . . . }
    public synchronized java.lang.String toString() { . . . }
}
```

Modifying the object to raise the exception

The `AccountImpl` object must be modified to use the exception by raising the exception under the appropriate error conditions.

Code sample 5.8 Modifying the object implementation to raise an exception

```
public class AccountImpl extends Bank.AccountPOA {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() throws AccountFrozen {
        if (_balance < 50) {
            throws AccountFrozen();
        } else {
            return _balance;
        }
    }
    private float _balance;
}
```

Catching user exceptions

When an object implementation raises an exception, the ORB is responsible for reflecting the exception to your client program. Checking for a `UserException` is similar to checking for a `SystemException`. To modify the account client program to catch the `AccountFrozen` exception, make modifications like those shown in Code sample 5.9.

Code sample 5.9 Catching a `UserException`

```
public class Client {
    public static void main(String[] args) {
        try {
            // Initialize the ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // Bind to an account
            Account account = AccountHelper.bind(orb, "/bank_poa",
                "BankAccount".getBytes());

            // Get the balance of the account
            float balance = account.balance();

            // Print the account balance
            System.out.println("The account balance is $" + balance);
        }
        // Check for AccountFrozen exception
        catch(Account.AccountFrozen e) {
            System.err.println("AccountFrozen returned:");
            System.err.println(e);
        }
        // Check for system errors
        catch(org.omg.CORBA.SystemException sys_excep) {
            ...
        }
    }
}
```

Adding fields to user exceptions

You can associate values with user exceptions. Code sample 5.10 shows how to modify the IDL interface specification to add a reason code to the `AccountFrozen` user exception. The object implementation that raises the exception is responsible for setting the reason code. The reason code is printed automatically when the exception is put on the output stream.

Code sample 5.10 Adding a reason code to the `AccountFrozen` exception

```
// Bank.idl
module Bank {
    interface Account {
        exception AccountFrozen {
            int reason;
        };
        float balance() raises(AccountFrozen);
    };
};
```


Server concepts

This part of the VisiBroker for Java *Programmer's Guide* includes these chapters.

- Chapter 6 “Server basics”
- Chapter 7 “Using POAs”
- Chapter 8 “Managing threads and connections”
- Chapter 9 “Using the tie mechanism”

Server basics

This chapter outlines the tasks that are necessary to set up a server to receive client requests.

Overview

The basic steps that you'll perform in setting up your server are:

- Initialize the ORB
- Create and setup the POA
- Activate the POA Manager
- Activate objects
- Wait for client requests

This chapter describes each task in a global manner to give you an idea of what you must consider. The specifics of each step are dependent on your individual requirements.

Initializing the ORB

As stated in the previous chapter, the ORB provides a communication link between client requests and object implementations. Each application must initialize the ORB before communicating with it.

Code sample 6.1 Initializing the ORB

```
// Initialize the ORB.  
org.omg.CORBA.ORB orb=org.omg.CORBA.ORB.init(args,null);
```

Creating the POA

Early versions of the CORBA object adapter (the *Basic Object Adapter*, or *BOA*) didn't permit portable object server code. A new specification was developed by the OMG to address these issues and the *Portable Object Adapter* (or *POA*) was created.

Note A discussion of the POA can be quite extensive. This section introduces you to some of the basic features of the POA. For detailed information, see “Using POAs” on page 7-1 and the OMG specification.

In basic terms, the POA (and its components) determine which *servant* should be invoked when a client request is received, and then invokes that servant. A servant is a programming object that provides the implementation of an *abstract object*. A servant is not a CORBA object.

One POA (called the *root POA*) is supplied by each ORB. You can create additional POAs and configure them with different behaviors. You can also define the characteristics of the objects the POA controls.

The steps to setting up a POA with a servant include:

- Obtaining a reference to the root POA
- Defining the POA policies
- Creating a POA as a child of the root POA
- Creating a servant and activating it
- Activating the POA through its manager

Some of these steps may be different for your application.

Obtaining a reference to the root POA

All server applications must obtain a reference to the root POA to manage objects or to create new POAs.

Code sample 6.2 Obtaining a reference to the root POA

```
//2. Get a reference to the root POA
org.omg.CORBA.Object obj = orb.resolve_initial_reference("RootPOA");
// Narrow the object reference to a POA reference
POA rootPoa = org.omg.PortableServer.POAHelper.narrow(obj);
```

You can obtain a reference to the root POA by using `resolve_initial_references`. `resolve_initial_references` returns a value of type `CORBA::Object`. You are responsible for narrowing the returned object reference to the desired type, which is `PortableServer::POA` in the above example.

You can then use this reference to create other POAs, if needed.

Creating the child POA

The root POA has a predefined set of *policies* that cannot be changed. A policy is an object that controls the behavior of a POA and the objects the POA manages. If you need a different behavior, such as different lifespan policy, you'll need to create a new POA.

POAs are created as children of existing POAs using `create_POA`. You can create as many POAs as you think are required.

Note Child POAs do not inherit the policies of their parent POAs.

In the following example, a child POA is created from the root POA and has a persistent lifespan policy. The POA Manager for the root POA is used to control the state of this child POA. More information on POA Managers are described later in this chapter.

Code sample 6.3 Creating the policies and the child POA

```
// Create policies for our persistent POA
org.omg.CORBA.Policy[] policies = {
    rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
};
// Create myPOA with the right policies
POA myPOA = rootPOA.create_POA( "bank_agent_poa", rootPOA.the_POAManager(), policies );
```

Implementing servant methods

IDL has a syntax similar to C++ and can be used to define modules, interfaces, data structures, and more. When you compile an IDL that contains an interface, a class is generated which serves as the base class for your servant. For example, in the `Bank.IDL` file, an `AccountManager` interface is described.

Code sample 6.4 Interfaces described in `Bank.IDL`

```
module Bank{
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open (in string name);
    };
};
```

An `AccountManagerPOA.java` is created and serves as the skeleton code (implementation base code) for the `AccountManager` object implementation on the server side, which is shown below.

Code sample 6.5 `AccountManagerImpl` code

```
import org.omg.PortableServer.*;
import java.util.*;
public class AccountManagerImpl extends Bank.AccountManagerPOA {
    public synchronized Bank.Account open(String name) {
```

```

// Lookup the account in the account dictionary.
Bank.Account account = (Bank.Account) _accounts.get(name);
// If there was no account in the dictionary, create one.
if(account == null) {
    // Make up the account's balance, between 0 and 1000 dollars.
    float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
    // Create the account implementation, given the balance.
    AccountImpl accountServant = new AccountImpl(balance);
    try {
        // Activate it on the default POA which is root POA for this servant
        account = Bank.AccountHelper.narrow(_default_POA().
            servant_to_reference(accountServant));
    } catch (Exception e) {
        e.printStackTrace();
    }
    // Print out the new account.
    System.out.println("Created " + name + "'s account: " + account);
    // Save the account in the account dictionary.
    _accounts.put(name, account);
}
// Return the account.
return account;
}
private Dictionary _accounts = new Hashtable();
private Random _random = new Random();
}

```

The **AccountManager** implementation must be created and activated in the server code. In this example, **AccountManager** is activated with `activate_object_with_id`, which passes the object ID to the *Active Object Map* where it is recorded. The *Active Object Map* is simply a table that maps IDs to servants. This approach ensures that this object is always available when the POA is active and is called *explicit object activation*.

Code sample 6.6 Creating and activating the servant

```

// Create the servant
AccountManagerImpl managerServant = new AccountManagerImpl();
// Decide on the ID for the servant
byte[] managerId = "BankManager".getBytes();
// Activate the servant with the ID on myPOA
myPOA.activate_object_with_id(managerId, managerServant);

```

Activating the POA

The last step is to activate the POA Manager associated with your POA. By default, POA Managers are created in a *holding* state. In this state, all requests are routed to a holding queue and are not processed. To allow requests to be dispatched, the *POA Manager* associated with the POA must be changed from the holding state to an active state. A POA Manager is simply an object that controls the state of the POA (whether requests are queued, processed, or discarded.) A POA Manager is associated with a POA during POA creation. You can specify a POA Manager to use, or let the system create a new one for you (enter `null` as the POA Manager name in `create_POA()`).

Code sample 6.7 Activating the POA manager

```
// Activate the POA manager
rootPOA.the_POAManager().activate();
```

Activating objects

In the preceding section, there was a brief mention of explicit object activation. There are several ways in which objects can be activated:

- **Explicit:** All objects are activated upon server start-up via calls to the POA
- **On-demand:** The servant manager activates an object when it receives a request for a servant not yet associated with an object ID
- **Implicit:** Objects are implicitly activated by the server in response to an operation by the POA, not by any client request
- **Default servant:** The POA uses the default servant to process the client request

A complete discussion of object activation is in Chapter 7, “Using POAs.” For now, just be aware that there are several means for activating objects.

Waiting for client requests

Once your POA is set up, you can wait for client requests by using `orb.run()`. This process will run until the server is terminated.

Code sample 6.8 Waiting for incoming requests

```
// Wait for incoming requests
orb.run();
```

Complete example

The following code shows the complete code described in this chapter.

Code sample 6.9 Complete server side code

```
// Server.java
import org.omg.PortableServer.*;
public class Server {
public static void main(String[] args) {
    try {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // get a reference to the root POA
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

        // Create policies for our persistent POA
        org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
        };
        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA( "bank_agent_poa", rootPOA.the_POAManager(),
            policies );

        // Create the servant
        AccountManagerImpl managerServant = new AccountManagerImpl();
        // Decide on the ID for the servant
        byte[] managerId = "BankManager".getBytes();
        // Activate the servant with the ID on myPOA
        myPOA.activate_object_with_id(managerId, managerServant);
        // Activate the POA manager
        rootPOA.the_POAManager().activate();
        System.out.println(myPOA.servant_to_reference(managerServant) + " is ready.");
        // Wait for incoming requests
        orb.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Using POAs

What is a Portable Object Adapter?

Portable Object Adapters replace Basic Object Adapters; they provide portability on the server side.

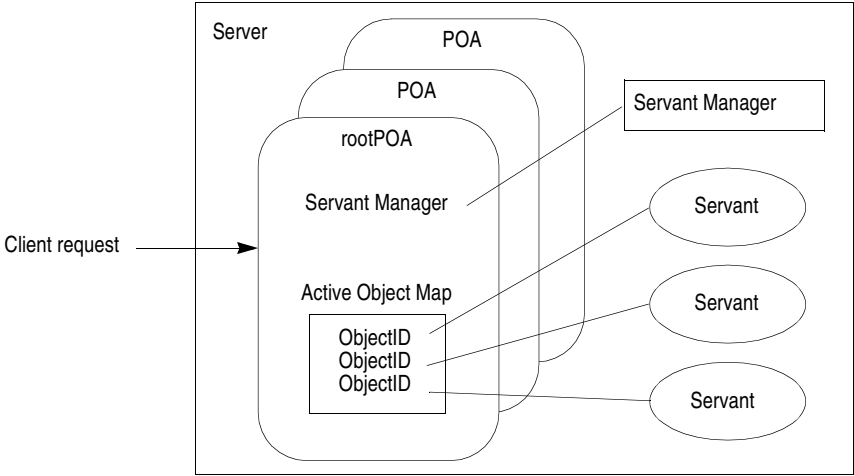
A POA is the intermediary between the implementation of an object and the ORB. In its role as an intermediary, a POA routes requests to servants and, as a result may cause servants to run and create child POAs if necessary.

Servers can support multiple POAs. At least one POA must be present, which is called the rootPOA. The rootPOA is created automatically for you. The set of POAs is hierarchical; all POAs have the rootPOA as their ancestor.

Servant managers locate and assign servants to objects for the POA. When an abstract object is assigned to a servant, it is called an active object and the servant is said to incarnate the active object. Every POA has one Active Object Map which keeps track of the object IDs of active objects and their associated active servants.

Note This chapter highlights only the major topics that relate to POAs. For a complete description, see the OMG specification.

Figure 7.1 Overview of the POA



POA terminology

Following are definitions of some terms with which you will become more familiar as you read through this chapter.

Table 7.1 Portable Object Adapter terminology

Term	Description
Active Object Map	Table that maps active CORBA objects (through their object IDs) to servants. There is one Active Object Map per POA.
adapter activator	Object that can create a POA on demand when a request is received for a child POA that does not exist.
etherealize	Remove the association between a servant and an abstract CORBA object.
incarnate	Associate a servant with an abstract CORBA object.
ObjectID	Way to identify a CORBA object within the object adapter. An ObjectID can be assigned by the object adapter or the application and is unique only within the object adapter in which it was created. Servants are associated with abstract objects through ObjectIDs.
persistent object	CORBA objects that live beyond the server process that created them.
POA manager	Object that controls the state of the POA; for example, whether the POA is receiving or discarding incoming requests.
Policy	Object that controls the behavior of the associated POA and the objects the POA manages.
rootPOA	Each ORB is created with one POA called the rootPOA. You can create additional POAs (if necessary) from the rootPOA.
servant	Any code that implements the methods of a CORBA object, but is not the CORBA object itself.

Table 7.1 Portable Object Adapter terminology (continued)

Term	Description
servant manager	An object responsible for managing the association of objects with servants, and for determining whether an object exists. More than one servant manager can exist.
transient object	A CORBA object that lives only within the process that created it.

Steps for creating and using POAs

Although the exact process can vary, following are the basic steps that occur during the POA lifecycle are:

- 1 Define the POA's policies.
- 2 Create the POA.
- 3 Activate the POA through its POA manager.
- 4 Create and activate servants.
- 5 Create and use servant managers.
- 6 Use adapter activators.

Depending on your needs, some of these steps may be optional. For example, you only have to activate the POA if you want it to process requests.

POA policies

Each POA has a set of policies that define its characteristics. When creating a new POA, you can use the default set of policies or use different values to suit your requirements. You can only set the policies when creating a POA; you can not change the policies of an existing POA. POAs do not inherit the policies from their parent POA.

The following sections lists the POA policies, their values, and the default value (used by the rootPOA).

Thread policy

The thread policy specifies the threading model to be used by the POA. The thread policy can have the following values:

ORB_CTRL_MODEL: (Default) The POA is responsible for assigning requests to threads. In a multi-threaded environment, concurrent requests may be delivered using multiple threads. Note that VisiBroker uses multithreading model.

SINGLE_THREAD_MODEL: The POA processes requests sequentially. In a multi-threaded environment, all calls made by the POA to servants and servant managers are thread-safe.

Lifespan policy

The lifespan policy specifies the lifespan of the objects implemented in the POA. The lifespan policy can have the following values:

TRANSIENT: (Default) A transient object activated by a POA cannot outlive the POA that created it. Once the POA is deactivated, an `OBJECT_NOT_EXIST` exception occurs if an attempt is made to use any object references generated by the POA.

PERSISTENT: A persistent object activated by a POA can outlive the process in which it was first created. Requests invoked on a persistent object may result in the implicit activation of a process, a POA and the servant that implements the object.

Object ID Uniqueness policy

The Object ID Uniqueness policy allows a single servant to be shared by many abstract objects. The Object ID Uniqueness policy can have the following values:

UNIQUE_ID: (Default) Activated servants support only one Object ID.

MULTIPLE_ID: Activated servants can have one or more Object IDs. The Object ID must be determined within the method being invoked at run time.

ID Assignment policy

The ID assignment policy specifies whether object IDs are generated by server applications or by the POA. The ID Assignment policy can have the following values:

USER_ID: Objects are assigned object IDs by the application.

SYSTEM_ID: (Default) Objects are assigned object IDs by the POA. If the **PERSISTENT** policy is also set, object IDs must be unique across all instantiations of the same POA.

Typically, **USER_ID** is for persistent objects, and **SYSTEM_ID** is for transient objects. If you want to use **SYSTEM_ID** for persistent objects, you can extract them from the servant or object reference.

Servant Retention policy

The Servant Retention policy specifies whether the POA retains active servants in the Active Object Map. The Servant Retention policy can have the following values:

RETAIN: (Default) The POA tracks object activations in the Active Object Map. **RETAIN** is usually used with `ServantActivators` or explicit activation methods on POA.

NON_RETAIN: The POA does not retain active servants in the Active Object Map. **NON_RETAIN** must be used with `ServantLocators`.

`ServantActivators` and `ServantLocators` are types of servant managers. For more information on servant managers, see “Using servants and servant managers” on page 7-12.

Request Processing policy

The Request Processing policy specifies how requests are processed by the POA.

USE_ACTIVE_OBJECT_MAP_ONLY: (Default) If the Object ID is not listed in the Active Object Map, an `OBJECT_NOT_EXIST` exception is returned. The POA must also use the `RETAIN` policy with this value.

USE_DEFAULT_SERVANT: If the Object ID is not listed in the Active Object Map or the `NON_RETAIN` policy is set, the request is dispatched to the default servant. If no default servant has been registered, an `OBJ_ADAPTER` exception is returned. The POA must also use the `MULTIPLE_ID` policy with this value.

USE_SERVANT_MANAGER: If the Object ID is not listed in the Active Object Map or the `NON_RETAIN` policy is set, the servant manager is used to obtain a servant.

Implicit Activation policy

The Implicit Activation policy specifies whether the POA supports implicit activation of servants. The Implicit Activation policy can have the following values:

IMPLICIT_ACTIVATION: The POA supports implicit activation of servants. Servants can be activated by converting them to an object reference with `org.omg.PortableServer.POA.servant_to_reference()` or by invoking `_this()` on the servant. The POA must also use the `SYSTEM_ID` and `RETAIN` policies with this value.

NO_IMPLICIT_ACTIVATION: (Default) The POA does not support implicit activation of servants.

Bind Support policy

The Bind Support policy (a VisiBroker-specific policy) controls the registration of POAs and active objects with the VisiBroker osagent. If you have several thousands of objects, it is not feasible to register all of them with the osagent. Instead, you can register the POA with the osagent. When a client request is made, the POA name and the object ID is included in the bind request so that the osagent can correctly forward the request.

The BindSupport policy can have the following values:

BY_INSTANCE: All active objects are registered with the osagent. The POA must also use the `PERSISTENT` and `RETAIN` policy with this value.

BY_POA: (Default) Only POAs are registered with the osagent. The POA must also use the `PERSISTENT` policy with this value.

NONE: Neither POAs nor active objects are registered with the osagent.

Creating POAs

To implement objects using the POA, at least one POA object must exist on the server. To ensure that a POA exists, a rootPOA is provided during the ORB initialization. This POA uses the default POA policies described earlier in this chapter.

Once the rootPOA is obtained, you can create child POAs that implement a specific server-side policy set.

POA naming convention

Each POA keeps track of its name and its full POA name (the full hierarchical path name.) The hierarchy is indicated by a slash (/). For example, /A/B/C means that POA C is a child of POA B, which in turn is a child of POA A. The first slash (see the previous example) indicates the rootPOA. If the Bind Support:BY_POA policy is set on POA C, then /A/B/C is registered with the osagent and the client binds with /A/B/C.

If your POA name contains escape characters or other delimiters, VisiBroker precedes these characters with a double backslash (\\) when recording the names internally. For example, if you have two POAs in a hierarchy like

```
org.omg.PortableServer.POA myPOA1 = rootPOA.create_POA("A/B",
    poaManager,
    policies);
org.omg.PortableServer.POA myPOA2 = myPOA1.create_POA("\t",
    poaManager,
    policies);
```

a client would bind using:

```
org.omg.CORBA.Object manager = ((com.inprise.vbroker.orb.ORB) orb).bind("/A\\B/\t",
    managerId,
    null,
    null);
```

Obtaining the rootPOA

The following code sample illustrates how a server application can obtain its rootPOA.

Code sample 7.1 Obtaining the rootPOA

```
// Initialize the ORB.
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
// get a reference to the rootPOA
org.omg.PortableServer.POA rootPOA =
    POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

Note The `resolve_initial_references` method returns a value of type `org.omg.CORBA.Object`. You are responsible for narrowing the returned object reference to the desired type, which is `org.omg.PortableServer.POA` in the previous example.

Setting the POA properties

Policies are not inherited from the parent POA. If you want a POA to have a specific characteristic, you must identify all the policies that are different from the default value. For more information about POA policies, see “POA policies” on page 7-3.

Code sample 7.2 Example of creating policies for a POA

```
org.omg.CORBA.Policy[] policies = {
    rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
};
```

Creating and activating the POA

A POA is created using `create_POA` on its parent POA. You can name the POA anything you like; however, the name must be unique with respect to all other POAs with the same parent. If you attempt to give two POAs the same name, a CORBA exception (`AdapterAlreadyExists`) is raised.

To create a new POA, use `create_POA` as follows:

```
POA create_POA(POA_Name, POAManager, PolicyList);
```

The POA manager controls the state of the POA (for example, whether it is processing requests). If `null` is passed to `create_POA` as the POA manager name, a new POA manager object is created and associated with the POA. Typically, you’ll want to have the same POA manager for all POAs. For more information about the POA manager, see “Managing POAs with the POA manager” on page 7-17.

POA managers (and POAs) are not automatically activated once created. Use `activate()` to activate the POA manager associated with your POA.

Code sample 7.3 Example of creating a POA

```
// Create policies for our persistent POA
org.omg.CORBA.Policy[] policies = {
    rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)};
// Create myPOA with the right policies
org.omg.PortableServer.POA myPOA =
    rootPOA.create_POA( "bank_agent_poa", rootPOA.the_POAManager(), policies );
```

Activating objects

When CORBA objects are associated with an active servant, if the POA's Servant Retention Policy is RETAIN, the associated object ID is recorded in the Active Object Map and the object is activated. Activation can occur in one of several ways:

- Explicit activation

The server application itself explicitly activates objects by calling `activate_object` or `activate_object_with_id`.

- On-demand activation

The server application instructs the POA to activate objects through a user-supplied servant manager. The servant manager must first be registered with the POA through `set_servant_manager`.

- Implicit activation

The server activates objects solely by in response to certain operations. If a servant is not active, there is nothing a client can do to make it active (for example, requesting for an inactive object does not make it active.)

- Default servant

The POA uses a single servant to implement all of its objects.

Activating objects explicitly

By setting `IdAssignmentPolicy::SYSTEM_ID` on a POA, objects can be explicitly activated without having to specify an object ID. The server invokes `activate_object` on the POA which activates, assigns and returns an object ID for the object. This type of activation is most common for transient objects. No servant manager is required since neither the object nor the servant is needed for very long.

Objects can also be explicitly activated using object IDs. A common scenario is during server initialization where the user invokes `activate_object_with_id` to activate all the objects managed by the server. No servant manager is required since all the objects are already activated. If a request for a non-existent object is received, an `OBJECT_NOT_EXIST` exception is raised. This has obvious negative effects if your server manages large numbers of objects.

Code sample 7.4 Example of explicit activation using `activate_object_with_id`

```
// Create the account manager servant.
Servant managerServant = new AccountManagerImpl(rootPoa);
// Activate the newly created servant.
testPoa.activate_object_with_id("BankManager".getBytes(), managerServant);
// Activate the POAs
testPoa.the_POAManager().activate();
```

Activating objects on demand

On-demand activation occurs when a client requests an object that does not have an associated servant. After receiving the request, the POA searches the Active Object Map for an active servant associated with the object ID. If none is found, the POA invokes `incarnate` on the servant manager which passes the object ID value to the servant manager. The servant manager can do one of three things:

- Find an appropriate servant which then performs the appropriate operation for the request
- Raise an `OBJECT_NOT_EXIST` exception that is returned to the client
- Forward the request to another object

The POA policies determine any additional steps that may occur. For example, if `RequestProcessingPolicy.USE_SERVANT_MANAGER` and `ServantRetentionPolicy.RETAIN` are enabled, the Active Object Map is updated with the servant and object ID association.

An example of on-demand activation is shown in Code sample 7.7 on page 7-13.

Activating objects implicitly

A servant can be implicitly activated by certain operations if the POA has been created with `ImplicitActivationPolicy.IMPLICIT_ACTIVATION`, `IdAssignmentPolicy.SYSTEM_ID` and `ServantRetentionPolicy.RETAIN`. Implicit activation can occur with:

- `POA.servant_to_reference` method
- `POA.servant_to_id` method
- `_this()` servant method

If the POA has `ObjectIdUniquenessPolicy.UNIQUE_ID` set, implicit activation can occur when any of the above operations are performed on an inactive servant.

If the POA has `ObjectIdUniquenessPolicy.MULTIPLE_ID` set, `servant_to_reference` and `servant_to_id` operations always perform implicit activation, even if the servant is already active.

Activating with the default servant

Use the `RequestProcessing.USE_DEFAULT_SERVANT` policy to have the POA invoke the same servant no matter what the object ID is. This is useful when little data is associated with each object.

Code sample 7.5 Example of activating all objects with the same servant

```
import org.omg.PortableServer.*;
public class Server {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
```

```
// get a reference to the rootPOA
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
// Create policies for our persistent POA
org.omg.CORBA.Policy[] policies = {
    rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
    rootPOA.create_request_processing_policy(
        RequestProcessingPolicyValue.USE_DEFAULT_SERVANT
    )
};
// Create myPOA with the right policies
POA myPOA = rootPOA.create_POA( "bank_default_servant_poa",
    rootPOA.the_POAManager(),
    policies );
// Create the servant
AccountManagerImpl managerServant = new AccountManagerImpl();

// Set the default servant on our POA
myPOA.set_servant(managerServant);
org.omg.CORBA.Object ref;
// Activate the POA manager
rootPOA.the_POAManager().activate();
// Generate the reference and write it out. One for each Checking and Savings
// account types. Note that we are not creating any
// servants here and just manufacturing a reference which is not
// yet backed by a servant.
try {
    ref = myPOA.create_reference_with_id("CheckingAccountManager".getBytes(),
        "IDL:Bank/AccountManager:1.0");
    // Write out checking object ID
    java.io.PrintWriter pw = new java.io.PrintWriter(
        new java.io.FileWriter("cref.dat") );
    pw.println(orb.object_to_string(ref));
    pw.close();
    ref = myPOA.create_reference_with_id("SavingsAccountManager".getBytes(),
        "IDL:Bank/AccountManager:1.0");
    // Write out savings object ID
    pw = new java.io.PrintWriter( new java.io.FileWriter("sref.dat") );
    System.gc();
    pw.println(orb.object_to_string(ref));
    pw.close();
} catch ( java.io.IOException e ) {
    System.out.println("Error writing the IOR to file ");
    return;
}
System.out.println("Bank Manager is ready.");
// Wait for incoming requests
orb.run();
} catch (Exception e) {
    e.printStackTrace();
}
}
```


Deactivating objects

A POA can remove a servant from its Active Object Map. This may occur, for example, as a form of garbage-collection scheme. When the servant is removed from the map, it is deactivated. You can deactivate an object using `deactivate_object()`. When an object is deactivated, it doesn't mean this object is lost forever. It can always be reactivated at a later time.

Code sample 7.6 Example of deactivating an object

```
import org.omg.PortableServer.*;

public class AccountManagerActivator extends ServantActivatorPOA {
    public Servant incarnate (byte[] oid, POA adapter) throws ForwardRequest {
        Servant servant;
        String accountType = new String(oid);
        System.out.println("\nAccountManagerActivator.incarnate called with ID = "
            + accountType + "\n");
        // Create Savings or Checking Servant based on AccountType
        if ( accountType.equalsIgnoreCase("SavingsAccountManager"))
            servant = (Servant )new SavingsAccountManagerImpl();
        else
            servant =(Servant)new CheckingAccountManagerImpl();
        new DeactivateThread(oid, adapter).start();
        return servant;
    }
    public void etherealize (byte[] oid,
        POA adapter,
        Servant serv,
        boolean cleanup_in_progress,
        boolean remaining_activations) {
        System.out.println("\nAccountManagerActivator.etherealize called with ID = "
            + new String(oid) + "\n");
        System.gc();
    }
}

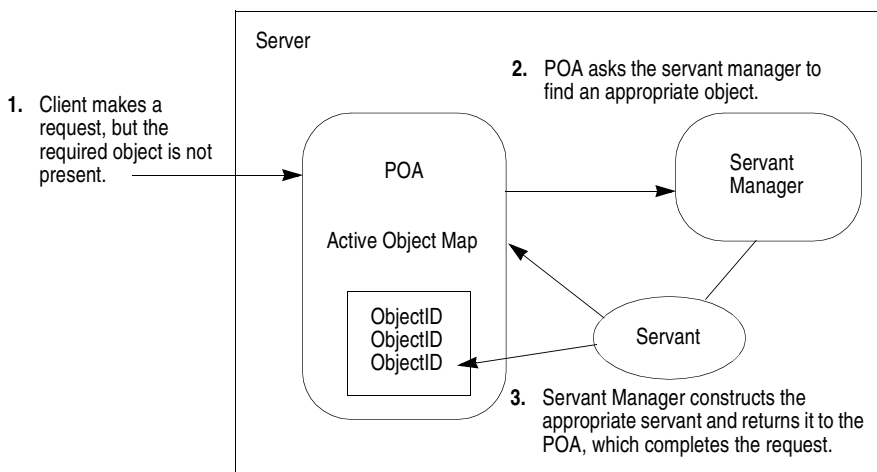
class DeactivateThread extends Thread {
    byte[] _oid;
    POA _adapter;
    public DeactivateThread(byte[] oid, POA adapter) {
        _oid = oid;
        _adapter = adapter;
    }
    public void run() {
        try {
            Thread.currentThread().sleep(15000);
            System.out.println("\nDeactivating the object with ID = " +
                new String(_oid) + "\n");
            _adapter.deactivate_object(_oid);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Using servants and servant managers

Servant managers perform two types of operations: find and return a servant, and deactivate a servant. They allow the POA to activate objects when a request for an inactive object is received. Servant managers are optional. For example, servant managers are not needed when your server loads all objects at startup. Servant managers may also inform clients to forward requests to another object using `ForwardRequest`.

A servant is an active instance of an implementation. The POA maintains a map of the active servants and the object IDs of the servants. When a client request is received, the POA first checks this map to see if the object ID (embedded in the client request) has been recorded. If it exists, then the POA forwards the request to the servant. If the object ID is not found in the map, the servant manager is asked to locate and activate the appropriate servant. This is only an example scenario; the exact scenario depends on what POA policies you have in place.

Figure 7.2 Example servant manager function



There are two types of servant managers: *ServantActivator* and *ServantLocator*. The type of policy already in place determines which callback is used. For more information on POA policy, see “POA policies” on page 7-3. Typically, a *ServantActivator* activates persistent objects and a *ServantLocator* activates transient objects.

To use servant managers, `RequestProcessingPolicy.USE_SERVANT_MANAGER` must be set as well as the policy which defines the type of servant manager (`ServantRetentionPolicy.RETAIN` for *ServantActivator* or `ServantRetentionPolicy.NON_RETAIN` for *ServantLocator*.)

ServantActivators

ServantActivators are used when `ServantRetentionPolicy.RETAIN` and `RequestProcessingPolicy.USE_SERVANT_MANAGER` are set. Servants activated by this type of servant manager are tracked in the Active Object Map.

The following events occur while processing requests using servant activators:

- 1 A client request is received (client request contains POA name, the object ID, and a few others.)
- 2 The POA first checks the active object map. If the object ID is found there, the operation is passed to the servant, and the response is returned to the client.
- 3 If the object ID is not found in the active object map, the POA invokes `incarnate` on a servant manager. `incarnate` passes the object ID and the POA in which the object is being activated.
- 4 The servant manager locates the appropriate servant.
- 5 The servant ID is entered into the active object map, and the response is returned to the client.

Note The `etherealize` and `incarnate` method implementations are user-supplied code.

At a later date, the servant can be deactivated. This may occur from several sources, including the `deactivate_object` operation, deactivation of the POA manager associated with that POA, and so forth. More information on deactivating objects is described in “Deactivating objects” on page 7-11.

Code sample 7.7 Example server code illustrating servant activator-type servant manager

```
import org.omg.PortableServer.*;
public class Server {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.FORB.init(args,null);
            // get a reference to the rootPOA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Create policies for our POA. We need persistence life span and
            // use servant manager request processing policies
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
                rootPOA.create_request_processing_policy(RequestProcessingPolicyValue.
                    USE_SERVANT_MANAGER)
            };

            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA( "bank_servant_activator_poa",
                rootPOA.the_POAManager(),
                policies );

            // Create the servant activator servant and get its reference
            ServantActivator sa = new AccountManagerActivator()._this(orb);
            // Set the servant activator on our POA
            myPOA.set_servant_manager(sa);
            org.omg.CORBA.Object ref;
```

```

// Activate the POA manager
rootPOA.the_POAManager().activate();
// Generate the reference and write it out. One for each Checking and Savings
// account types .Note that we are not creating any
// servants here and just manufacturing a reference which is not
// yet backed by a servant.
try {
    ref = myPOA.create_reference_with_id("CheckingAccountManager".getBytes(),
        "IDL:Bank/AccountManager:1.0");
    // Write out checking object ID
    java.io.PrintWriter pw =
        new java.io.PrintWriter( new java.io.FileWriter("cref.dat") );
    pw.println(orb.object_to_string(ref));
    pw.close();
    ref = myPOA.create_reference_with_id("SavingsAccountManager".getBytes(),
        "IDL:Bank/AccountManager:1.0");

    // Write out savings object ID
    pw = new java.io.PrintWriter( new java.io.FileWriter("sref.dat") );
    System.gc();
    pw.println(orb.object_to_string(ref));
    pw.close();
} catch ( java.io.IOException e ) {
    System.out.println("Error writing the IOR to file ");
    return;
}
System.out.println("Bank Manager is ready.");
// Wait for incoming requests
orb.run();
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

The servant manager for this example follows.

Code sample 7.8 Servant manager for servant activator example

```

import org.omg.PortableServer.*;
public class AccountManagerActivator extends ServantActivatorPOA {
    public Servant incarnate (byte[] oid, POA adapter) throws ForwardRequest {
        Servant servant;
        String accountType = new String(oid);
        System.out.println("\nAccountManagerActivator.incarnate called with ID = " +
            accountType + "\n");
        // Create Savings or Checking Servant based on AccountType
        if ( accountType.equalsIgnoreCase("SavingsAccountManager"))
            servant = (Servant) new SavingsAccountManagerImpl();
        else
            servant = (Servant) new CheckingAccountManagerImpl();
        new DeactivateThread(oid, adapter).start();
        return servant;
    }
}

```

```

    public void etherealize (byte[] oid,
        POA adapter,
        Servant serv,
        boolean cleanup_in_progress,
        boolean remaining_activations) {
        System.out.println("\nAccountManagerActivator.etherealize called with ID = " +
            new String(oid) + "\n");
        System.gc();
    }
}

class DeactivateThread extends Thread {
    byte[] _oid;
    POA _adapter;
    public DeactivateThread(byte[] oid, POA adapter) {
        _oid = oid;
        _adapter = adapter;
    }

    public void run() {
        try {
            Thread.currentThread().sleep(15000);
            System.out.println("\nDeactivating the object with ID = " +
                new String(_oid) + "\n");
            _adapter.deactivate_object(_oid);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

ServantLocators

In many situations, the POA's Active Object Map could become quite large and consume memory. To reduce memory consumption, a POA can be created with `RequestProcessingPolicy.USE_SERVANT_MANAGER` and `ServantRetentionPolicy.NON_RETAIN`, meaning that the servant-to-object association is not stored in the active object map. Since no association is stored, `ServantLocator` servant managers are invoked for each request.

The following events occur while processing requests using servant locators:

- 1 A client request, which contains the POA name and the object id, is received.
- 2 Since `ServantRetentionPolicy.NON_RETAIN` is used, the POA does not search the active object map for the object ID.
- 3 The POA invokes `preinvoke` on a servant manager. `preinvoke` passes the object ID, the POA in which the object is being activated, and a few other parameters.
- 4 The servant locator locates the appropriate servant.

- 5 The operation is performed on the servant and the response is returned to the client.
- 6 The POA invokes `postinvoke` on the servant manager.

Note The `preinvoke` and `postinvoke` methods are user-supplied code.

Code sample 7.9 Example server code illustrating servant locator-type servant managers

```
import org.omg.PortableServer.*;
public class Server {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // get a reference to the rootPOA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Create policies for our POA. We need persistence life span,
            // use servant manager request processing policies and non retain
            // retention policy. This non retain policy will let us use the
            // servant locator instead of servant activator
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
                rootPOA.create_servant_retention_policy(ServantRetentionPolicyValue.
                    NON_RETAIN),
                rootPOA.create_request_processing_policy(RequestProcessingPolicyValue.
                    USE_SERVANT_MANAGER)
            };
            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA( "bank_servant_locator_poa",
                rootPOA.the_POAManager(),
                policies );
            // Create the servant locator servant and get its reference
            ServantLocator sl = new AccountManagerLocator()._this(orb);
            // Set the servant locator on our POA
            myPOA.set_servant_manager(sl);
            org.omg.CORBA.Object ref ;
            // Activate the POA manager
            rootPOA.the_POAManager().activate();
            // Generate the reference and write it out. One for each Checking and Savings
            // account types .Note that we are not creating any
            // servants here and just manufacturing a reference which is not
            // yet backed by a servant.
            try {
                ref = myPOA.create_reference_with_id("CheckingAccountManager".getBytes(),
                    "IDL:Bank/AccountManager:1.0");
                // Write out checking object ID
                java.io.PrintWriter pw =
                    new java.io.PrintWriter( new java.io.FileWriter("cref.dat") );
                pw.println(orb.object_to_string(ref));
                pw.close();
                ref = myPOA.create_reference_with_id("SavingsAccountManager".getBytes(),
                    "IDL:Bank/AccountManager:1.0");
```

```

        // Write out savings object ID
        pw = new java.io.PrintWriter( new java.io.FileWriter("sref.dat") );
        System.gc();
        pw.println(orb.object_to_string(ref));
        pw.close();
    } catch ( java.io.IOException e ) {
        System.out.println("Error writing the IOR to file ");
        return;
    }
    System.out.println("BankManager is ready.");
    // Wait for incoming requests
    orb.run();
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

The servant manager for this example follows.

Code sample 7.10 Servant manager for servant locator example

```

import org.omg.PortableServer.*;
import org.omg.PortableServer.ServantLocatorPackage.CookieHolder;
public class AccountManagerLocator extends ServantLocatorPOA {
    public Servant preinvoke (byte[] oid, POA adapter,
        java.lang.String operation,
        CookieHolder the_cookie) throws ForwardRequest {
        String accountType = new String(oid);
        System.out.println("\nAccountManagerLocator.preinvoke called with ID = " +
            accountType + "\n");
        if ( accountType.equalsIgnoreCase("SavingsAccountManager"))
            return new SavingsAccountManagerImpl();
        return new CheckingAccountManagerImpl();
    }
    public void postinvoke (byte[] oid,
        POA adapter,
        java.lang.String operation,
        java.lang.Object the_cookie,
        Servant the_servant) {
        System.out.println("\nAccountManagerLocator.postinvoke called with ID = " +
            new String(oid) + "\n");
    }
}

```

Managing POAs with the POA manager

A POA manager controls the state of the POA (whether requests are queued or discarded), and can deactivate the POA. Each POA is associated with a POA manager object. A POA manager can control one or several POAs.

A POA manager is associated with a POA when the POA is created. You can specify the POA manager to use, or specify `null` to have a new POA Manager created.

Code sample 7.11 Naming the POA and its POA Manager

```
POA myPOA = rootPOA.create_POA( "bank_agent_poa",
    rootPOA.the_POAManager(),
    policies );
POA myPOA = rootPOA.create_POA( "bank_agent_poa",
    null,
    policies );
```

A POA manager is “destroyed” when all its associated POAs are destroyed.

A POA manager can have the following four states:

- Holding
- Active
- Discarding
- Inactive

These states in turn determine the state of the POA. They are each described in detail in the following sections.

Getting the current state

To get the current state of the POA manager, use

```
enum State{HOLDING, ACTIVE, DISCARDING, INACTIVE};
State get_state();
```

Holding state

By default, when a POA manager is created, it is in the holding state. When the POA manager is in the holding state, the POA queues all incoming requests.

Requests that require an adapter activator are also queued when the POA manager is in the holding state.

To change the state of a POA manager to holding, use

```
void hold_requests(wait_for_completion)
    raises (AdapterInactive);
```

`wait_for_completion` is Boolean. If `FALSE`, this operation returns immediately after changing the state to holding. If `TRUE`, this operation returns only when all requests started prior to the state change have completed or when the POA manager is changed to a state other than holding. `AdapterInactive` is the exception raised if the POA manager was in the inactive state prior to calling this operation.

Note POA managers in the inactive state cannot change to the holding state.

Any requests that have been queued but not yet started will continue to be queued during the holding state.

Active state

When the POA manager is in the active state, its associated POAs process requests.

To change the POA manager to the active state, use

```
void activate()
    raises (AdapterInactive);
```

`AdapterInactive` is the exception raised if the POA manager was in the inactive state prior to calling this operation.

Note POA managers currently in the inactive state can not change to the active state.

Discarding state

When the POA manager is in the discarding state, its associated POAs discard all requests that have not yet started. In addition, the adapter activators registered with the associated POAs are not called. This state is useful when the POA is receiving too many requests. You need to notify the client that their request has been discarded and to resend their request. There is no inherent behavior for determining if and when the POA is receiving too many requests. It is up to you to set-up thread monitoring if so desired.

To change the POA manager to the discarding state, use

```
void discard_requests(wait_for_completion)
    raises (AdapterInactive);
```

The `wait_for_completion` option is Boolean. If `FALSE`, this operation returns immediately after changing the state to holding. If `TRUE`, this operation returns only when all requests started prior to the state change have completed or when the POA manager is changed to a state other than discarding. `AdapterInactive` is the exception raised if the POA manager was in the inactive state prior to calling this operation.

Note POA managers currently in the inactive state can not change to the discarding state.

Inactive state

When the POA manager is in the inactive state, its associated POAs reject incoming requests. This state is used when the associated POAs are to be shut down.

Note POA managers in the inactive state can not change to any other state.

To change the POA manager to the inactive state, use

```
void deactivate(etherealize_objects, wait_for_completion)
    raises (AdapterInactive);
```

After the state changes, if `etherealize_objects` is `TRUE`, then all associated POAs that have `Servant RetentionPolicy.RETAIN` and `RequestProcessingPolicy.USE_SERVANT_MANAGER` set call `etherealize` on the servant manager for all active objects. If `etherealize_objects` is `FALSE`, then `etherealize` is not called. The `wait_for_completion` option is Boolean. If `FALSE`, this operation returns immediately after changing the state to inactive. If `TRUE`,

this operation returns only when all requests started prior to the state change have completed or `etherealize` has been called on all associated POAs (that have `Servant RetentionPolicy.RETAIN` and `RequestProcessingPolicy.USE_SERVANT_MANAGER`). `AdapterInactive` is the exception raised if the POA manager was in the inactive state prior to calling this operation.

Setting the listening and dispatching properties

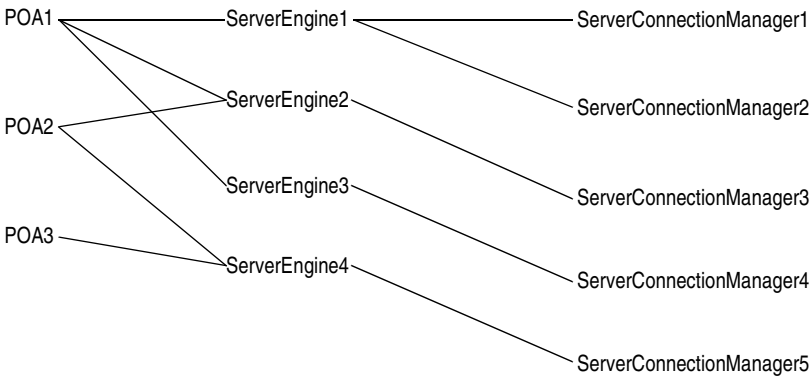
Policies that cover listener and dispatcher features previously supported by the BOA are not supported by POAs. In order to provide these features, a VisiBroker-specific policy (`ServerEnginePolicy`) can be used.

A server engine consists of:

- Host name
- Proxy host name
- Server connection manager or list of server connection managers

The following illustration shows how these fit together.

Figure 7.3 Server engine overview



The simplest case is where POAs have their own unique single server engine. Here, requests for different POAs arrive on different ports. A POA can also have multiple server engines. In this scenario, a single POA supports requests coming from multiple input ports.

Notice that POAs can share server engines. When server engines are shared, the POAs listen to the same port. Even though the requests for (multiple) POAs arrive at the same port, they are dispatched correctly because of the POA name embedded in the request. This scenario occurs, for example, when you use a default server engine and create multiple POAs (without specifying a new server engine during the POA creation).

Setting the server engine properties

The following properties determine which server engine(s) are used by default:

```
vbroker.se.<server_engine_name>.host
vbroker.se.<server_engine_name>.proxyHost
vbroker.se.<server_engine_name>.scms
```

If you don't specify a server engine policy, the POA assumes a server engine name of `iiop` and uses the following default values:

```
vbroker.se.iiop_tp.host=null
vbroker.se.iiop_tp.proxyHost=null
vbroker.se.iiop_tp.scms=iiop
```

To change the default server engine policy, enter its name using the `vbroker.se.default` property and define the values for all the components of the new server engine. For example,

```
vbroker.se.default=abc,def
vbroker.se.abc.host=cob
vbroker.se.abc.proxyHost=null
vbroker.se.abc.scms=cobscm1, cobscm2
vbroker.se.def.host=gob
vbroker.se.def.proxyHost=null
vbroker.se.def.scms=gobscm1
```

For more information on how to set properties, see Chapter 14, "Setting properties."

Setting the server connection manager properties

The server connection manager consists of three property groups: manager, listener, and dispatcher.

Manager properties

You can set the following manager properties:

- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.type`
Identifies the connection manager type. Currently, `Socket` is the only type supported.
- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.connectionMax`
Defines the maximum number of concurrent, incoming connections allowed. The default value is 0, meaning an unlimited number of connections.
- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.connectionMaxIdle`
Defines the maximum number of idle seconds before the connection is shutdown. The default value is 0, meaning there is no timeout.

Listener properties

You can set the following listener properties:

- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.listener.type`
Identifies the listener type. Currently, IIOP is the only type supported.
- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.listener.port`
Defines the listening port that the POA associated with this server which the connection manager uses. The default port is 0 (zero), meaning the system will pick a random port number.
- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.listener.proxyPort`
Specifies the proxy port number used with the proxy host name property. The default value, 0 (zero), means the system will pick a random port number.

Dispatcher properties

You can set the following dispatcher properties:

- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.dispatcher.type`
Identifies the dispatcher type. Currently, `ThreadPool` and `ThreadSession` are the only types supported.
- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.dispatcher.threadMax`
Used only when type is set to `ThreadPool`.
- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.dispatcher.threadMaxIdle`
Used only when type is set to `ThreadPool`.
- `vbroker.se.<server_engine>.scm.<server_connection_mgr>.dispatcher.threadMin`
Used only when type is set to `ThreadPool`.

When to use these properties

There are many times where you need to change some of the server engine properties. The method for changing these properties depends on what you need. For example, suppose you want to change the port number. You could accomplish this by:

- Changing the default `listener.port` property
- Creating a new server engine

Changing the default `listener.port` property is the simplest method, but this affects all POAs that use the default server engine. This may or may not be what you want.

If you want to change the port number on a specific POA, then you'll have to create a new server engine, define the properties for this new server engine, and then reference the new server engine when creating the POA. The previous sections show how to update the server engine properties. The following code snippet shows how to define properties of a server engine and create a POA with a user-defined server engine policy.

Code sample 7.12 Creating a POA with a specific server engine

```
// Server.java
import org.omg.PortableServer.*;
public class Server {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // Get property manager
            com.inprise.vbroker.properties.PropertyManager pm =
                ((com.inprise.vbroker.ORB)orb).getPropertyManager();
            pm.addProperty("vbroker.se.mySe.host", "");
            pm.addProperty("vbroker.se.mySe.proxyHost", "");
            pm.addProperty("vbroker.se.mySe.scms", "scmlist");
            pm.addProperty("vbroker.se.mySe.scm.scmlist.manager.type", "Socket");
            pm.addProperty("vbroker.se.mySe.scm.scmlist.manager.connectionMax", 100);
            pm.addProperty("vbroker.se.mySe.scm.scmlist.manager.connectionMaxIdle", 300);
            pm.addProperty("vbroker.se.mySe.scm.scmlist.listener.type", "IIOP");
            pm.addProperty("vbroker.se.mySe.scm.scmlist.listener.port", 55000);
            pm.addProperty("vbroker.se.mySe.scm.scmlist.listener.proxyPort", 0);
            pm.addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.type", "ThreadPool");
            pm.addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMax", 100);
            pm.addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMin", 5);
            pm.addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMaxIdle", 300);
            // get a reference to the root POA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Create our server engine policy
            org.omg.CORBA.Any seAny = orb.create_any();
            org.omg.CORBA.StringSequenceHelper.insert(seAny, new String[]{"mySe"});
            org.omg.CORBA.Policy sePolicy =
                orb.create_policy(
                    com.inprise.vbroker.PortableServerExt.SERVER_ENGINE_POLICY_TYPE.value, seAny);
            // Create policies for our persistent POA
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT), sePolicy
            };
            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA("bank_se_policy_poa",
                rootPOA.the_POAManager(),
                policies );
            // Create the servant
            AccountManagerImpl managerServant = new AccountManagerImpl();
            // Activate the servant
            myPOA.activate_object_with_id("BankManager".getBytes(), managerServant);
            // Obtaining the reference
            org.omg.CORBA.Object ref = myPOA.servant_to_reference(managerServant);
```

```
// Now write out the IOR
try {
    java.io.PrintWriter pw =
        new java.io.PrintWriter( new java.io.FileWriter("ior.dat") );
    pw.println(orb.object_to_string(ref));
    pw.close();
} catch ( java.io.IOException e ) {
    System.out.println("Error writing the IOR to file ior.dat");
    return;
}
// Activate the POA manager
rootPOA.the_POAManager().activate();
System.out.println(ref + " is ready.");
// Wait for incoming requests
orb.run();
} catch (Exception e) {
    e.printStackTrace();
}
}
```

Adapter activators

Adapter activators are associated with POAs and provide the ability to create child POAs on-demand. This can be done during the `find_POA` operation, or when a request is received that names a specific child POA.

An adapter activator supplies a POA with the ability to create child POAs on demand, as a side-effect of receiving a request that names the child POA (or one of its children), or when `find_POA` is called with an `activate` parameter value of `TRUE`. An application server that creates all its needed POAs at the beginning of execution does not need to use or provide an adapter activator; it is necessary only for the case in which POAs need to be created during request processing.

While a request from the POA to an adapter activator is in progress, all requests to objects managed by the new POA (or any descendant POAs) will be queued. This serialization allows the adapter activator to complete any initialization of the new POA before requests are delivered to that POA.

For an example on using adapter activators, see the POA `adaptor_activator` example included with the product.

Processing requests

Requests contain the Object ID of the target object and the POA that created the target object reference. When a client sends a request, the ORB first locates the appropriate server, or starts the server if needed. It then locates the appropriate POA within that server.

Once the ORB has located the appropriate POA, it delivers the request to that POA. How the request is processed at that point depends on the policies of the POA and the object's activation state. For information about object activation states, see "Activating objects" on page 7-8.

- If the POA has `ServantRetentionPolicy.RETAIN`, the POA looks at the Active Object Map to locate a servant associated with the Object ID from the request. If a servant exists, the POA invokes the appropriate method on the servant.
- If the POA has `ServantRetentionPolicy.NON_RETAIN` or has `ServantRetentionPolicy.RETAIN` but did not find the appropriate servant, the following may take place:
 - If the POA has `RequestProcessingPolicy.USE_DEFAULT_SERVANT`, the POA invokes the appropriate method on the default servant.
 - If the POA has `RequestProcessingPolicy.USE_SERVANT_MANAGER`, the POA invokes `incarnate` or `preinvoke` on the servant manager.
 - If the POA has `RequestProcessingPolicy.USE_OBJECT_MAP_ONLY`, an exception is raised.

If a servant manager has been invoked but can not incarnate the object, the servant manager can raise a `ForwardRequest` exception.

Managing threads and connections

This chapter discusses the use of multiple threads in client programs and object implementations, and will help you understand the thread and connection model that VisiBroker uses.

Using threads with VisiBroker

A *thread*, or a single sequential flow of control within a process, is also called a lightweight process that reduces overhead by sharing fundamental parts with other threads. Threads are lightweight so that there can be many of them present within a process.

Using multiple threads provides concurrency within an application and improves performance. Applications can be structured efficiently with threads servicing several independent computations simultaneously. For example, a database system may have many user interactions in progress while at the same time performing several file and network operations. Although it is possible to write the software as one thread of control moving asynchronously from request to request, the code may be simplified by writing each request as a separate sequence, and letting the underlying system handle the synchronous interleaving of the different operations.

Multiple threads are useful when

- There are groups of lengthy operations that do not necessarily depend on other processing (like painting a window, printing a document, responding to a mouse-click, calculating a spreadsheet column, signal handling).
- There will be few locks on data (the amount of shared data is identifiable and small).
- The task can be broken into various responsibilities. For example, one thread can handle the signals and another thread can handle the user interface.

What thread policies does VisiBroker provide?

VisiBroker provides two thread policies: *thread pooling* or *thread-per-session*. The thread pooling and thread-per-session models differ in these fundamental ways

- Situation in which they are created
- How simultaneous requests from the same client are handled
- When and how threads are released

The default thread policy is thread pooling. For information about setting thread-per-session or changing properties in thread pooling, see “Setting dispatch policies and properties” on page 8-9.

Thread pooling policy

When your server uses the thread pooling policy, it defines the maximum number of threads that can be allocated to handle client requests. A worker thread is assigned for each client request, but only for the duration of that particular request. When a request is completed, the worker thread that was assigned to that request is placed into a pool of available threads so that it may be reassigned to process future requests from any of the clients.

Using this model, threads are allocated based on the amount of request traffic to the server object. This means that a highly active client that makes many requests to the server at the same time will be serviced by multiple threads—ensuring that the requests are quickly executed—while less active clients can share a single thread, and still have their requests immediately serviced. Additionally, the overhead associated with the creation and destruction of worker threads is reduced, because threads are reused rather than destroyed, and can be assigned to multiple connections.

VisiBroker conserves system resources by dynamically allocating the number of threads in the thread pool based on the number of concurrent client requests. If the client becomes very active, threads are allocated to meet its needs. If the threads remain inactive, VisiBroker releases them, only keeping enough threads to meet current client demand. This enables the optimal number of threads to be active in the server at all times.

The size of the thread pool grows based upon server activity and is fully configurable—either before or during execution—to meet the needs of specific distributed systems. With thread pooling, you can configure the following:

- Maximum and minimum number of threads
- Maximum idle time

Each time a client request is received, an attempt is made to assign a thread from the thread pool to process the request. If this is the first client request and the pool is empty, a thread will be created. Likewise, if all threads are busy, a new thread will be created to service the request.

A server can define a maximum number of threads that can be allocated to handle client requests. If there are no threads available in the pool and the maximum number of threads have already been created, the request will block until a thread currently in use has been released back into the pool.

Thread pooling is the default thread policy. You do not have to set up anything to define this environment. If you want to set properties for thread pooling, see “Setting dispatch policies and properties” on page 8-9.

Figure 8.1 Pool of threads is available

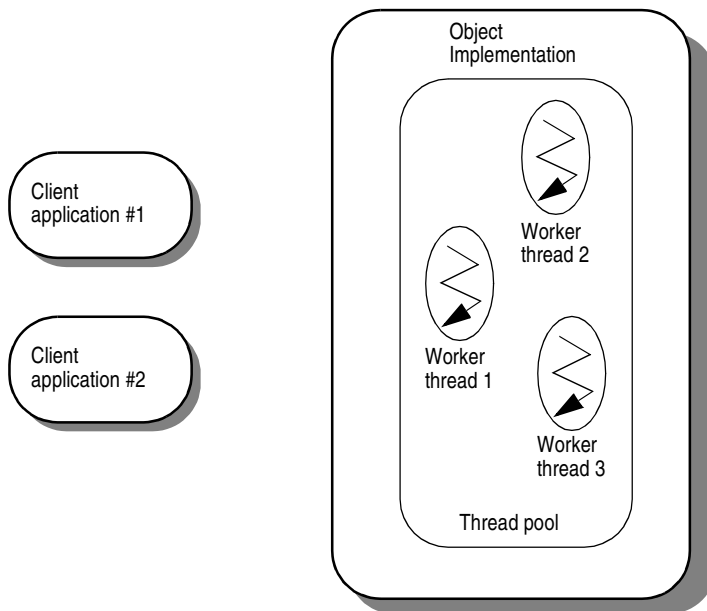
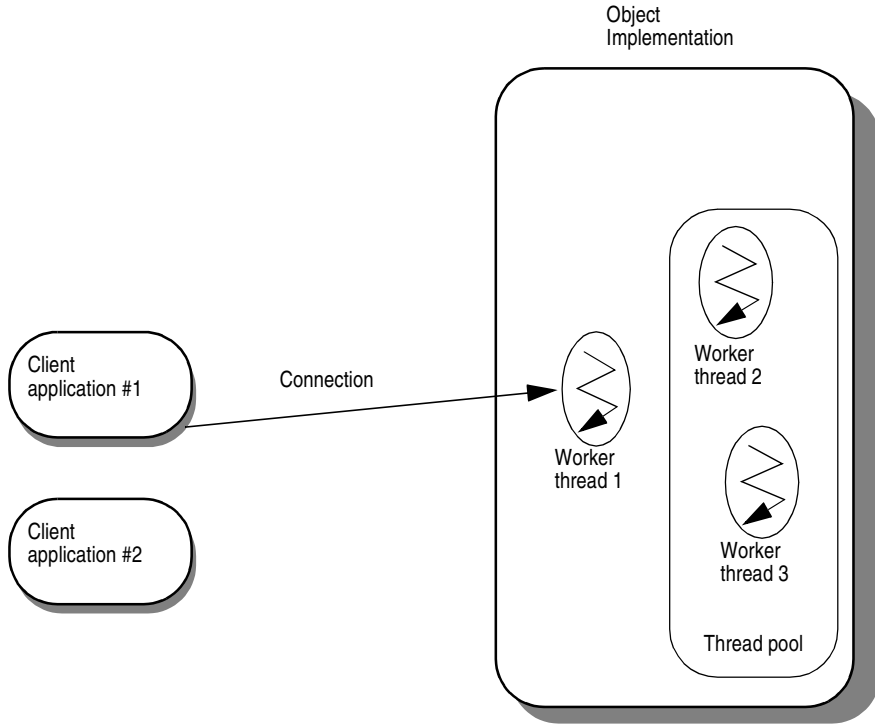
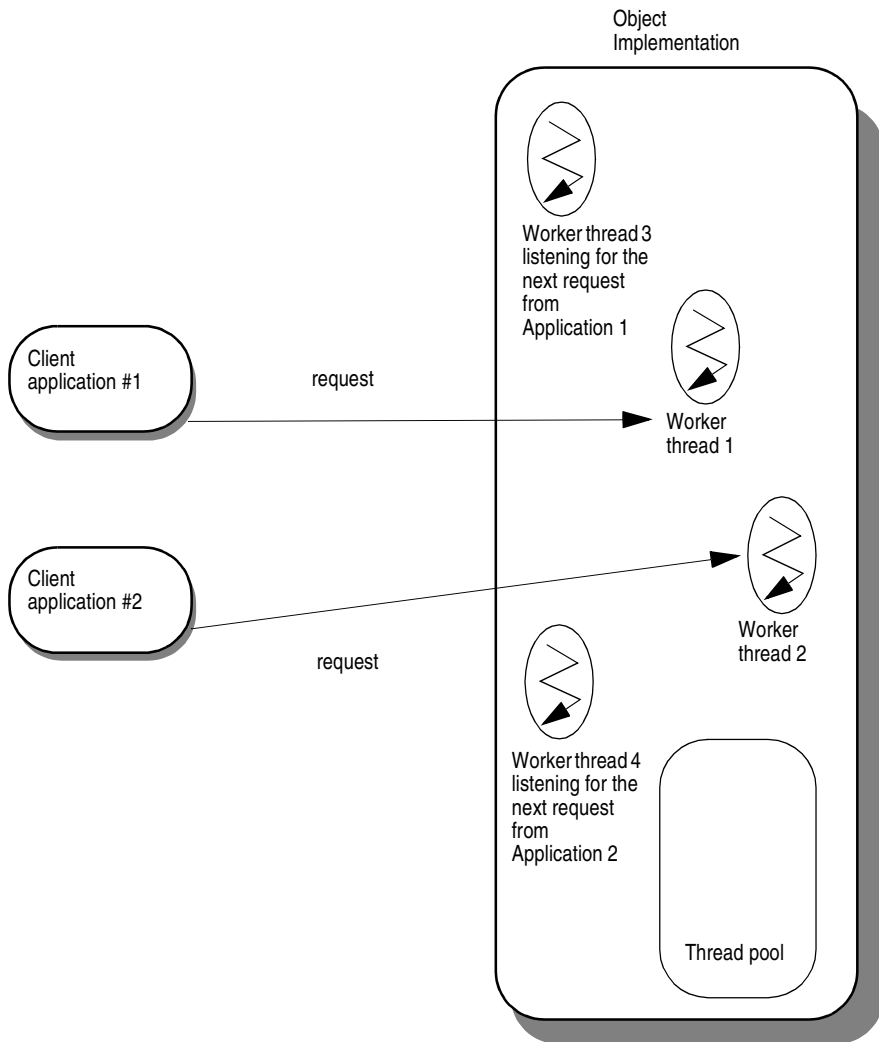


Figure 8.1 shows the object implementation using the thread pooling policy. As the name implies, in this policy there is an available pool of worker threads.

Figure 8.2 Client application #1 sends a request

In Figure 8.2, Client application #1 establishes a connection to the Object Implementation and a thread is created to handle requests. In thread pooling, there is one connection per client and one thread per connection. When a request comes in, a worker thread receives the request; that worker thread is no longer in the pool.

A worker thread is removed from the thread pool and is always listening for requests. When a request comes in, that worker thread reads in the request and dispatches the request to the appropriate object implementation. Prior to dispatching the request, the worker thread wakes up one other worker thread which then listens for the next request.

Figure 8.3 Client application #2 sends a request

As Figure 8.3 shows, when Client application #2 establishes its own connection and sends a request, a second worker thread is created. Worker thread #3 is now listening for incoming requests.

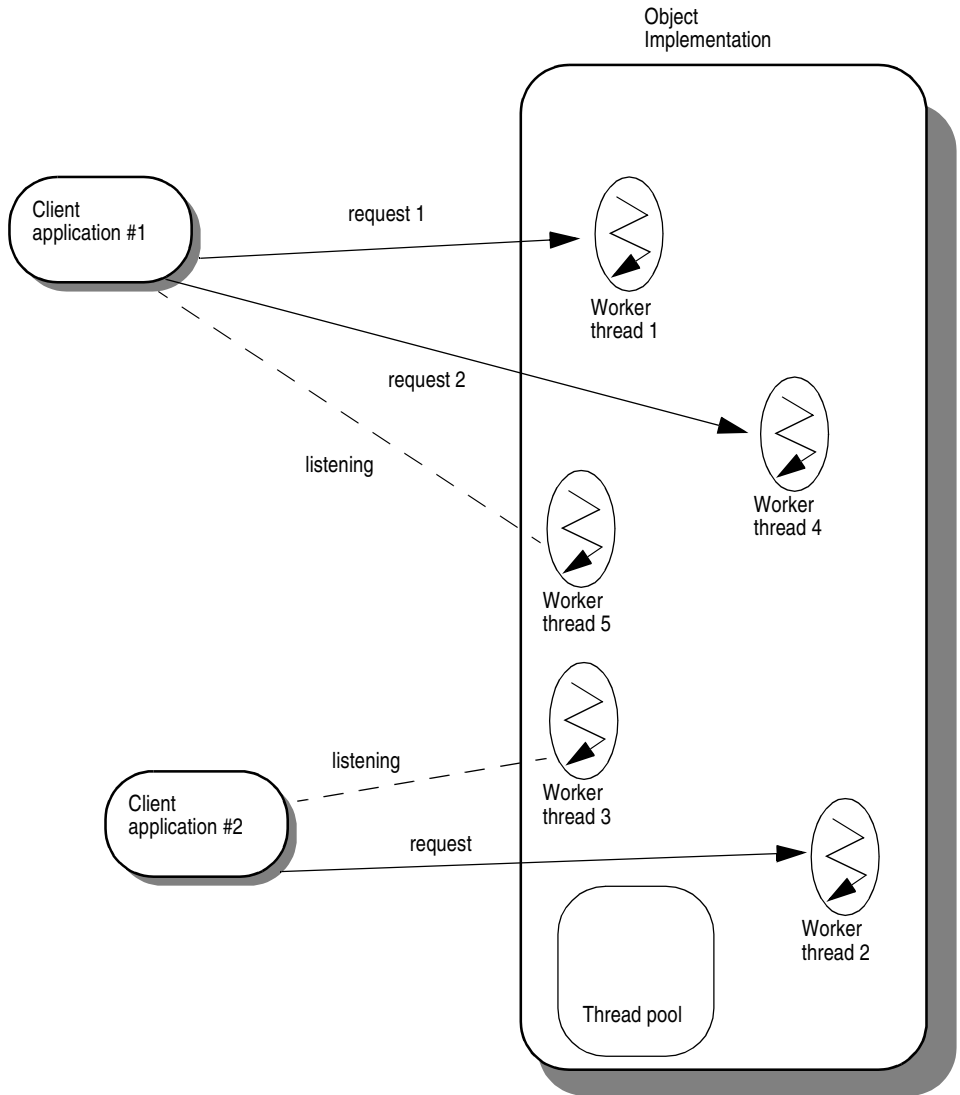
Figure 8.4 Client application #1 sends a second request

Figure 8.4 shows that when a second request comes in from Client application #1, it uses worker thread #4. Worker thread #5 is spawned to listen for new requests. If more requests came in from Client application #1, more threads would be assigned to handle them—each spawned after the listening thread receives a request. As worker threads complete their tasks, they are returned to the pool and become available to handle requests from any client.

Thread-per-session policy

With the thread-per-session policy, threading is driven by connections between the client and server processes. When your server selects the thread-per-session policy, a new thread is allocated each time a new client connects to a server. A thread is assigned to handle all the requests received from a particular client. Because of this, thread-per-session is also referred to as thread-per-connection. When the client disconnects from the server, the thread is destroyed. You may limit the maximum number of threads that can be allocated for client connections by setting the `vbroker.se.iiopt_ts.scm.iiopt_ts.manager.connectionMax` property.

Figure 8.5 Object implementation using the thread-per-session policy

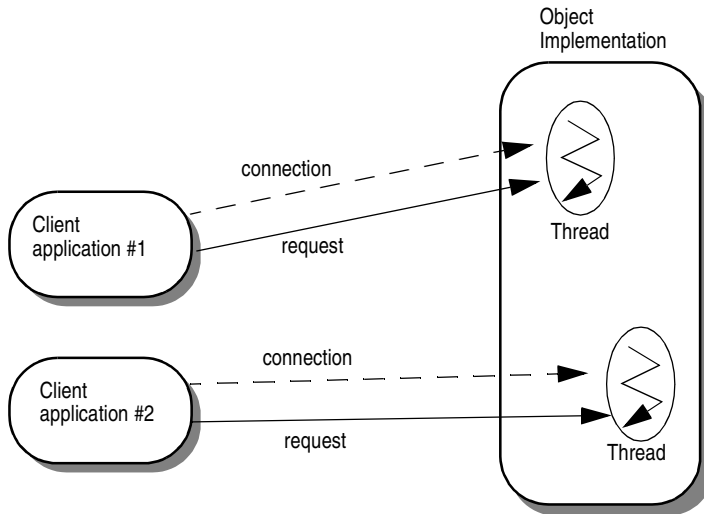
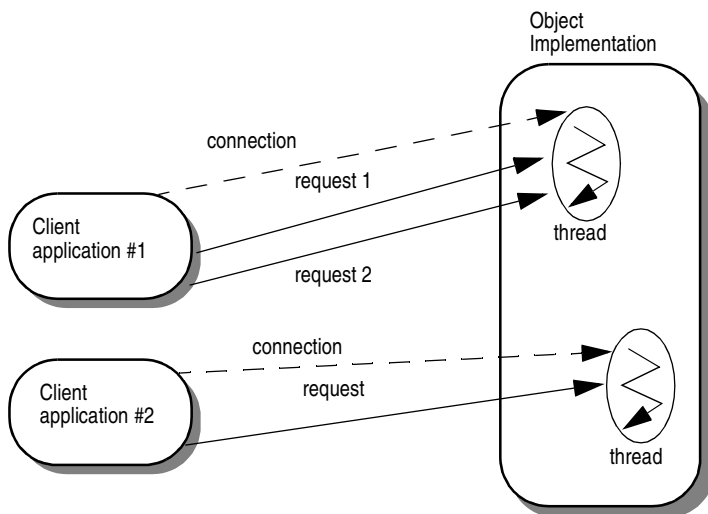


Figure 8.5 shows the use of the thread-per-session policy. The Client application #1 establishes a connection with the object implementation. A separate connection exists between Client application #2 and the object implementation. When a request comes in to the object implementation from Client application #1, a worker thread handles the request. When a request from Client application #2 comes in, a different worker thread is assigned to handle this request.

Figure 8.6 Second request comes in from the same client



In Figure 8.6, a second request has come in to the object implementation from Client application #1. The same thread that handles request 1 will handle request 2. The thread blocks request #2 until it completes request 1. (With thread-per-session, requests from the same Client are not handled in parallel.) When request #1 has completed, the thread can handle request 2 from Client application #1. Multiple requests may come in from Client application #1—they are handled in the order that they come in, no additional threads are assigned to Client application #1.

What connection management does VisiBroker provide?

Overall, VisiBroker's connection management minimizes the number of client connections to the server. In other words there is only one connection per server process which is shared. All client requests are multiplexed over the same connection, even if they originate from different threads. Additionally, released client connections are recycled for subsequent reconnects to the same server, eliminating the need for clients to incur the overhead of new connections to the server.

In the following scenario, a client application is bound to two objects in the server process. Each `bind()` shares a common connection to the server process, even though the `bind()` is for a different object in the server process.

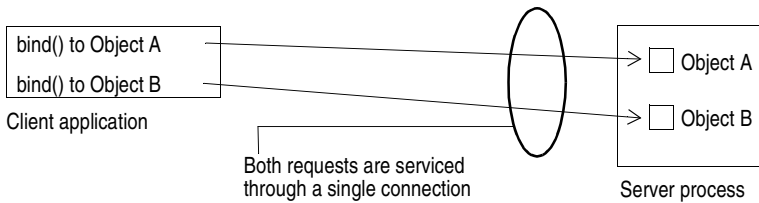
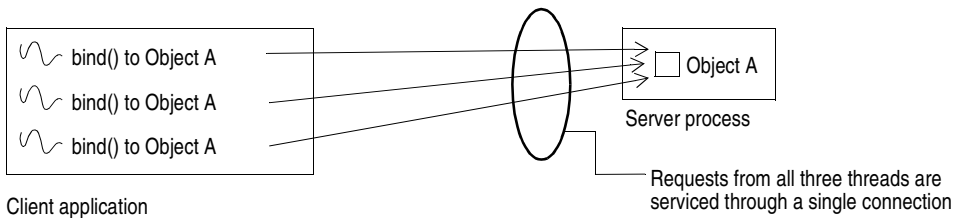
Figure 8.7 Binding to two objects in the same server process

Figure 8.8 shows the connections for a client using multiple threads that has several threads bound to an object on the server.

Figure 8.8 Binding to an object in a server process

As Figure 8.8 shows, all invocations from all threads are serviced by the same connection. For the scenario shown in Figure 8.8, the most efficient multithreading model to use is the thread pooling model (which is the default). If the thread-per-session model is used with this scenario, only one thread on the server will be allocated to service all requests from all threads in the client application—which could easily result in poor performance.

The maximum number of connections to a server or from a client can be configured. Inactive connections will be recycled when the maximum is reached, ensuring resource conservation.

Setting dispatch policies and properties

Each POA in a multi-threaded object server can choose between two dispatch models: *thread-per-session* or *thread pooling*. You choose a dispatch policy by setting the `dispatcher.type` property of the `ServerEngine`.

```
vbroker.se.<srvr_eng_name>.scm.<srvr_connection_mgr_name>.dispatcher.type="ThreadPool"
vbroker.se.<srvr_eng_name>.scm.<srvr_connection_mgr_name>.dispatcher.type="ThreadSession"
```

For more information about these properties, see the *VisiBroker for Java Reference* or “Setting the listening and dispatching properties” on page 7-20.

Thread pooling

`ThreadPool` (thread pooling) is the default dispatch policy when you create a POA without specifying the `ServerEnginePolicy`.

For `ThreadPool`, you can set the following properties:

- `vbroker.se.default.dispatcher.tp.threadMax`
- `vbroker.se.default.dispatcher.tp.threadMin`
- `vbroker.se.default.dispatcher.tp.threadMaxIdle`

Threads-per-session

When using the `ThreadSession` as the dispatcher type, you must set the `se.default` property to `iiop_ts`.

```
vbroker.se.default=iiop_ts
```

Coding considerations

All code within a server that implements an ORB object must be thread-safe. You must take special care when accessing a system-wide resource within an object implementation. For example, many database access methods are not thread-safe. Before your object implementation attempts to access such a resource, it must first lock access to the resource using a synchronized block.

If serialized access to an object is required, you need to create the POA on which this object is activated with the `SINGLE_THREAD_MODEL` value for the `ThreadPolicy`.

Using the tie mechanism

This chapter describes how the tie mechanism may be used to integrate existing Java code into a distributed object system. This chapter will enable you to create a delegation implementation or to provide implementation inheritance.

How does the tie mechanism work?

Object implementation classes normally inherit from a servant class generated by the `idl2java` compiler. The servant class, in turn, inherits from `org.omg.PortableServer.Servant`. When it is not convenient or possible to change existing classes to inherit from the `VisiBroker` servant class, the *tie* mechanism offers an attractive alternative.

The tie mechanism provides object servers with a *delegator implementation* class that inherits from `org.omg.PortableServer.Servant`. The delegator implementation does not provide any semantics of its own. It simply delegates every request it receives to the real implementation class, which can be implemented separately. The real implementation class is not required to inherit from `org.omg.PortableServer.Servant`.

With using the tie mechanism, two additional files are generated from the IDL compiler.

- `<InterfaceName>POATie` defers implementation of all IDL defined methods to a delegate. The delegate implements the interface `<InterfaceName>Operations`. Legacy implementations can be trivially extended to implement the operations interface and in turn delegate to the real implementation.
- `<InterfaceName>Operations` defines all of the methods that must be implemented by the object implementation. This interface acts as the delegate object for the associated `<InterfaceName>POATie` class when the tie mechanism is used.

Example program

Location of an example program using the tie mechanism

A version of the Bank example using the tie mechanism can be found in the VisiBroker for Java distribution under `examples/basic/bank_tie`.

Changes to the server class

The following code sample shows the modifications to the `Server` class. Note the extra step of creating an instance of `AccountManagerManagerPOATie`.

Code sample 9.1 `Server.java` file from the `bank_tie` directory

```
import org.omg.PortableServer.*;

public class Server {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // get a reference to the root POA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Create policies for our persistent POA
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
            };
            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA("bank_agent_poa",
                rootPOA.the_POAManager(), policies);
            // Create the tie which delegates to an instance of AccountManagerImpl
            Bank.AccountManagerPOATie tie =
                new Bank.AccountManagerPOATie(new AccountManagerImpl(rootPOA));
            // Decide on the ID for the servant
            byte[] managerId = "BankManager".getBytes();
            // Activate the servant with the ID on myPOA
            myPOA.activate_object_with_id(managerId, tie);
            // Activate the POA manager
            rootPOA.the_POAManager().activate();
            System.out.println("Server is ready.");
            // Wait for incoming requests
            orb.run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Changes to the AccountManager

The changes made to the `AccountManager` class (in comparison to the `Bank_agent` example) include:

- `AccountManagerImpl` **no longer extends** `Bank.AccountManagerPOA`.
- When a new `Account` is to be created, an `AccountPOATie` is also created and initialized.

Code sample 9.2 `AccountManagerImpl` class

```
import org.omg.PortableServer.*;
import java.util.*;

public class AccountManagerImpl implements Bank.AccountManagerOperations {
    public AccountManagerImpl(POA poa) {
        _accountPOA = poa;
    }
    public synchronized Bank.Account open(String name) {
        // Lookup the account in the account dictionary.
        Bank.Account account = (Bank.Account) _accounts.get(name);
        // If there was no account in the dictionary, create one.
        if (account == null) {
            // Make up the account's balance, between 0 and 1000 dollars.
            float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
            // Create an account tie which delegate to an instance of AccountImpl
            Bank.AccountPOATie tie =
                new Bank.AccountPOATie(new AccountImpl(balance));
            try {
                // Activate it on the default POA which is root POA for this servant
                account = Bank.AccountHelper.narrow(_accountPOA.servant_to_reference(tie));
            } catch (Exception e) {
                e.printStackTrace();
            }
            // Print out the new account.
            System.out.println("Created " + name + "'s account: " + account);
            // Save the account in the account dictionary.
            _accounts.put(name, account);
        }
        // Return the account.
        return account;
    }
    private Dictionary _accounts = new Hashtable();
    private Random _random = new Random();
    private POA _accountPOA = null;
}
```

Changes to the Account class

The changes made to the `Account` class (in comparison to the `Bank` example) are that it no longer extends `Bank.AccountPOA`.

Code sample 9.3 `AccountImpl` class

```
// Server.java
public class AccountImpl implements Bank.AccountOperations {
    public AccountImpl(float balance) {
        _balance = balance;
    }
    public float balance() {
        return _balance;
    }
    private float _balance;
}
```

Building the tie example

The instructions described in Chapter 4, “Developing an example application with `VisiBroker`,” are also valid for building the tie example.

Client concepts

This part of the VisiBroker for Java *Programmer's Guide* includes the following chapter.

Chapter 10 “Client basics”

Client basics

This chapter describes how client programs access and use distributed objects.

Initializing the ORB

The Object Request Broker (ORB) provides a communication link between the client and the server. When a client makes a request, the ORB locates the object implementation, activates the object if necessary, delivers the request to the object, and returns the response to the client. The client is unaware whether the object is on the same machine or across a network.

Note: The ORB is an intensive user of system resources. It is therefore advised that you create only one single instance of the ORB per process.

Though much of the work done by the ORB is transparent to you, your client program must explicitly initialize the ORB. ORB options, described in *VisiBroker for Java Reference*, can be specified as command-line arguments. Therefore, you must pass `args` to `ORB.init` to ensure that these options take effect.

Code sample 10.1 Initializing the ORB

```
public class Client {  
    public static void main (String[] args) {  
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);  
        . . .  
    }  
}
```

Binding to objects

A client program uses a remote object by obtaining a reference to the object. Object references are usually obtained using the `<interface>Helper's bind()` method. The ORB hides most of the details involved with obtaining the object reference, such as

locating the server that implements the object and establishing a connection to that server.

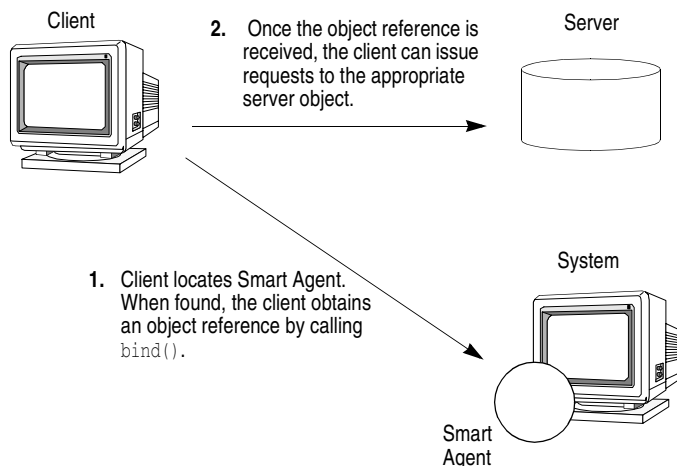
Action performed during the bind process

When the server process starts, it performs an `ORB.init()` and announces itself to Smart Agents on the network.

When your client program invokes the `bind()` method, the ORB performs several functions on behalf of your program.

- ORB contacts the Smart Agent to locate an object implementation that offers the requested interface. If an object name was specified when `bind()` was invoked, that name is used to further qualify the directory service search. The Object Activation Daemon (OAD), described in Chapter 20, “Using the Object Activation Daemon,” may be involved in this process if the server object has been registered with the OAD.
- When an object implementation is located, the ORB attempts to establish a connection between the object implementation that was located and your client program.
- Once the connection is successfully established, the ORB will create a proxy object and return a reference to that object. The client will invoke methods on the proxy object which will, in turn, interact with the server object.

Figure 10.1 Client interaction with the Smart Agent



Note Your client program will never invoke a constructor for the server class. Instead, an object reference is obtained by invoking the static `bind()` method.

Code sample 10.2 Example of bind call

```

. . .
Bank.AccountManager manager =

```

```
Bank.AccountManagerHelper.bind(orb, "/bank_agent_poa", "BankManager".getBytes());
. . .
```

Invoking operations on an object

Your client program uses an object reference to invoke an operation on an object or to reference data contained by the object. “Manipulating object references” describes the variety of ways that object references can be manipulated.

Code sample 10.3 Invoking an operation using an object reference

```
. . .
// Invoke the balance operation.
System.out.println("The balance in Account1: $" + account1.balance());
. . .
```

Manipulating object references

The object reference returned to your client program by the `bind()` method a CORBA object. Your client program can use the object reference to invoke operations on the object that have been defined in the object’s IDL interface specification. In addition, there are methods that all ORB objects inherit from the class `org.omg.CORBA.Object` that you can use to manipulate the object.

Converting a reference to a string

VisiBroker provides an ORB class with methods that allow you to convert an object reference to a string or convert a string back into an object reference. The CORBA specification refers to this process as stringification.

Table 10.1 Methods stringification and de-stringification

Method	Description
<code>object_to_string</code>	Converts an object reference to a string.
<code>string_to_object</code>	Converts a string to an object reference.

A client program can use the `object_to_string` method convert an object reference to a string and pass it to another client program. The second client may then de-stringify the object reference, using the `string_to_object` method, use the object reference without having to explicitly bind to the object.

Note Locally-scoped object references like the ORB or the POA cannot be stringified. If an attempt is made to do so, a `MARSHAL` exception is raised with the minor code 4.

Obtaining object and interface names

Table 10.2 shows the methods by the `Object` class that you can use to obtain the interface and object names as well as the repository id associated with an object reference. The interface repository is discussed in Chapter 21, “Using interface repositories.”

Note If you did not specify an object name when you invoked the `bind()` method, invoking the `_object_name()` method with the resulting object reference will return `null`.

Table 10.2 Methods obtaining interface and object names

Method	Description
<code>_object_name</code>	Returns this object’s name.
<code>_repository_id</code>	Returns the repository’s type identifier.

Determining the type of an object reference

You can check whether an object reference is of a particular type by using the `_is_a()` method. You must first obtain the repository id of the type you wish to check using the `_repository_id()` method. This method returns `true` if the object is either an instance of the type represented by `repository_id()` or if it is a sub-type. The method returns `false` if the object is not of the type specified. Note that this may require remote invocation to determine the type.

Note You cannot use the `instanceof` keyword to determine the runtime type.

You can use the `_is_equivalent()` method to check if two object references refer to the same object implementation. This method returns `true` if the object references are equivalent. This method returns `false` if the object references are distinct, but does not necessarily indicate that the object references are two distinct objects. This is a lightweight method and does not involve actual communication with the server object.

Table 10.3 Methods determining the type of an object reference

Method	Description
<code>_is_a</code>	Determines if an object implements a specified interface.
<code>_is_equivalent</code>	Returns <code>true</code> if two objects refer to the same interface implementation.

Determining the location and state of bound objects

Given a valid object reference, your client program can use the `_is_bound()` method to determine if the object is bound. The method returns `true` if the object is bound and `false` if the object is not bound.

The `_is_local()` method returns `true` if the client program and the object implementation reside within the same process or address space.

The `_is_remote()` method returns `true` if the client program and the object implementation reside in different processes, which may or may not be located on the same host.

Table 10.4 Methods for determining location and state of object reference

Method	Description
<code>_is_bound</code>	Returns <code>true</code> if a connection is currently active for this object.
<code>_is_local</code>	Returns <code>true</code> if this object is implemented in the local address space.
<code>_is_remote</code>	Returns <code>true</code> if this object's implementation does not reside in the local address space.

Note If the object is in the same process where the method is invoked, `_is_local()` returns `true`.

Narrowing object references

The process of converting an object reference's type from a general super-type to a more specific sub-type is called *narrowing*.

Note You cannot use the Java casting facilities for narrowing.

VisiBroker maintains a typograph for each object interface so that narrowing can be accomplished by using the object's `narrow()` method. The IDL exception `CORBA::BAD_PARAM` is thrown if the narrow fails, because the object reference does not support the requested type.

Code sample 10.4 Narrow method generated for the AccountManager

```
public abstract class AccountManagerHelper {
    . . .
    public static Bank.AccountManager narrow(org.omg.CORBA.Object object) {
        . . .
    }
    . . .
}
```

Widening object references

Converting an object reference's type to a super-type is called *widening*.

Code sample 10.5 shows an example of widening an `Account` pointer to an `Object` pointer. The pointer `acct` can be cast as an `Object` pointer because the `Account` class inherits from the `Object` class.

Code sample 10.5 Widening an object reference

```
. . .
Account account;
org.omg.CORBA.Object obj;
account = AccountHelper.bind();
obj = (org.omg.CORBA.Object) account;
. . .
```

Using quality of service

Quality of Service (QoS) utilizes policies to define and manage the connection between your client applications and the servers to which they connect.

Understanding Quality of Service

Quality of Service policy management is performed through operations accessible in the following contexts:

- ORB level policies are handled by a locality constrained `PolicyManager`, through which you can set Policies and view the current `Policy` overrides. Policies set at the ORB level override system defaults.
- Thread level policies are set through `PolicyCurrent`, which contains operations for viewing and setting `Policy` overrides at the thread level. Policies set at the thread level override system defaults and values set at the ORB level.
- Object level policies can be applied by accessing the base Object interface's quality of service operations. Policies applied at the Object level override system defaults and values set in at the ORB or thread level.

Policy overrides and effective policies

The effective policy is the policy that would be applied to a request after all applicable policy overrides have been applied. The effective policy is determined by comparing the `Policy` as specified by the IOR with the effective override. The effective `Policy` is the intersection of the values allowed by the effective override and the IOR-specified `Policy`. If the intersection is empty a `org.omg.CORBA.INV_POLICY` exception is raised.

QoS interfaces

The following interfaces are used to get and set QoS policies.

org.omg.CORBA.Object

`org.omg.CORBA.Object` contains the following methods used to get the effective policy and get or set the policy override.

- `_get_policy` returns the effective policy for an object reference.
- `_set_policy_override` returns a new object reference with the requested list of `Policy` overrides at the object level.

com.inprise.vbroker.CORBA.Object

In order to use this interface, you must cast `org.omg.CORBA.Object` to `com.inprise.vbroker.CORBA.Object`. Because this interface is derived from `org.omg.CORBA.Object`, the following methods are available in addition to the ones defined in `org.omg.CORBA.Object`.

- `_get_client_policy` returns the effective `Policy` for the object reference without doing the intersection with the server-side policies. The effective override is obtained by checking the specified overrides in first the object level, then at the thread level, and finally at the ORB level. If no overrides are specified for the requested `PolicyType` the system default value for `PolicyType` is used.
- `_get_policy_overrides` returns a list of `Policy` overrides of the specified policy types set at the object level. If the specified sequence is empty, all overrides at the object level will be returned. If no `PolicyTypes` are overridden at the object level, an empty sequence is returned.
- `_validate_connection` returns a boolean value based on whether the current effective policies for the object will allow an invocation to be made. If the object reference is not bound, a binding will occur. If the object reference is already bound, but current policy overrides have changed, or the binding is no longer valid, a rebind will be attempted, regardless of the setting of the `RebindPolicy` overrides. A false return value occurs if the current effective policies would raise an `INV_POLICY` exception. If the current effective policies are incompatible, a sequence of type `PolicyList` is returned listing the incompatible policies.

org.omg.CORBA.PolicyManager

The `PolicyManager` is an interface that provides methods for getting and setting `Policy` overrides for the ORB level.

- `get_policy_overrides` returns a `PolicyList` sequence of all the overridden policies for the requested `PolicyTypes`. If the specified sequence is empty, all `Policy` overrides at the current context level will be returned. If none of the requested `PolicyTypes` are overridden at the target `PolicyManager`, an empty sequence is returned.
- `set_policy_overrides` modifies the current set of overrides with the requested list of `Policy` overrides. The first input parameter, `policies`, is a sequence of references to `Policy` objects. The second parameter, `set_add`, of type `org.omg.CORBA.SetOverrideType` indicates whether these policies should be added onto any other overrides that already exist in the `PolicyManager` using `ADD_OVERRIDE`, or they should be added to a `PolicyManager` that doesn't contain any overrides using `SET_OVERRIDES`. Calling `set_policy_overrides` with an empty sequence of policies and a `SET_OVERRIDES` mode removes all overrides from a `PolicyManager`. Should you attempt to override policies that do not apply to your client, an `org.omg.CORBA.NO_PERMISSION` exception will be raised. If the request would cause the specified `PolicyManager` to be in an inconsistent state, no policies are changed or added, and an `InvalidPolicies` exception is raised.

org.omg.CORBA.PolicyCurrent

The `PolicyCurrent` interface derives from `PolicyManager` without adding new methods. It provides access to the policies overridden at the thread level. A reference to a thread's `PolicyCurrent` is obtained by invoking `org.omg.CORBA.ORB.resolve_initial_references` and specifying an identifier of "PolicyCurrent."

org.omg.Messaging.RebindPolicy

`RebindPolicy` accepts values of type `org.omg.Messaging.RebindMode` to define the behavior of the client when rebinding. `RebindPolicy`s are set only on the client side. It can have one of six values that determines the behavior in the case of a disconnection, an object forwarding request, or an object failure. The currently supported values are:

- `org.omg.Messaging.TRANSPARENT` allows the ORB to silently handle object-forwarding and necessary reconnections during the course of making a remote request.
- `org.omg.Messaging.NO_REBIND` allows the ORB to silently handle reopening of closed connections while making a remote request, but prevents any transparent object-forwarding that would cause a change in client-visible effective QoS policies. When `RebindMode` is set to `NO_REBIND`, only explicit rebind is allowed.
- `org.omg.Messaging.NO_RECONNECT` prevents the ORB from silently handling object-forwards or the reopening of closed connections. You must explicitly rebind and reconnect when `RebindMode` is set to `NO_RECONNECT`.
- `com.inprise.vbroker.QoSExt.VB_TRANSPARENT` is the default policy. It extends the functionality of `TRANSPARENT` by allowing transparent rebinding with both implicit and explicit binding. `VB_TRANSPARENT` is designed to be compatible with the object failover implementation in VisiBroker 3.x.
- `com.inprise.vbroker.QoSExt.VB_NOTIFY_REBIND` throws an exception if a rebind is necessary. The client catches this exception, and binds on the second invocation.
- `com.inprise.vbroker.QoSExt.VB_NO_REBIND` does not enable failover. It only allows the client ORB to reopen a closed connection to the same server; it does not allow object forwarding of any kind.

Note Be aware that if the effective policy for your client is `VB_TRANSPARENT` and your client is working with servers that hold state data, `VB_TRANSPARENT` could connect the client to a new server without the client being aware of the change of server, any state data held by the original server will be lost.

For more information on QoS policies and types, see the VisiBroker for Java *Reference* and the Messaging chapter of the CORBA 2.4 specification.

com.inprise.vbroker.QoSExt.RelativeConnectionTimeoutPolicy

The `RelativeConnectionTimeoutPolicy` indicates a timeout after which attempts to connect to an object using one of the available endpoints is aborted. The timeout situation is likely to happen with objects protected by firewalls, where HTTP tunneling is the only way to connect to the object.

com.inprise.vbroker.QoSExt.DeferBindPolicy

The `DeferBindPolicy` determines if the ORB will attempt to contact the remote object when it is first created, or to delay this contact until the first invocation is made. The values of `DeferBindPolicy` are `true` and `false`. If `DeferBindPolicy` is set to `true` all binds will be deferred until the first invocation of a binding instance. The default value is `false`.

If you create a client object, and `DeferBindPolicy` is set to `true`, you may delay the server startup until the first invocation. This option existed before as an option to the `Bind` method on the generated helper classes.

com.inprise.vbroker.QoSExt.ExclusiveConnectionPolicy

The `ExclusiveConnectionPolicy` is a `Visibroker`-specific policy that gives you the ability to establish an exclusive (non-shared) connection to the specified server object. You assign this policy a boolean value of `true` or `false`. If the policy is `true`, connections to the server object are exclusive. If the policy is `false`, existing connections are reused if possible and a new connection is opened only if reuse is not possible. The default value is `false`.

This policy provides the same capabilities as were provided by `Object._clone()` in `Visibroker 3.x`.

An example of how to establish exclusive and non-exclusive connections is provided in the `CloneClient.java` example.

com.inprise.vbroker.QoSExt.SyncScopePolicy

The `SyncScopePolicy` defines the level of synchronization for a request with respect to the target. Values of type `SyncScope` are used in conjunction with a `SyncScopePolicy` to control the behavior of oneway operations.

The default `SyncScopePolicy` is `SYNC_WITH_TRANSPORT`.

Note: Applications must explicitly set an ORB-level `SyncScopePolicy` to ensure portability across ORB implementations. When instances of `SyncScopePolicy` are created, a value of type `Messaging::SyncScope` is passed to `CORBA::ORB::create_policy`. This policy is only applicable as a client-side override.

QoS exceptions

- `org.omg.CORBA.INV_POLICY` is raised when there is an incompatibility between `Policy` overrides.
- `org.omg.CORBA.REBIND` is raised when the `RebindPolicy` has a value of `NO_REBIND`, `NO_RECONNECT`, or `VB_NOTIFY_REBIND` and an invocation on a bound object references results in an object-forward or location-forward message.
- `org.omg.CORBA.PolicyError` is raised when the requested `Policy` is not supported.
- `org.omg.CORBA.InvalidPolicies` can be raised when an operation is passed a `PolicyList` sequence. The exception body contains the policies from the sequence that are not valid, either because the policies are already overridden within the current scope, or are not valid in conjunction with other requested policies.

QoS example

The following example creates a `RebindPolicy` of type `TRANSPARENT` and sets the policy on the ORB, thread, and object levels.

```
Any policyValue= orb.create_any();
RebindModeHelper.insert(policyValue, org.omg.Messaging.TRANSPARENT.value);
Policy myRebindPolicy = orb.create_policy(REBIND_POLICY_TYPE.value, policyValue);

//get a reference to the ORB policy manager
org.omg.CORBA.PolicyManager manager;

try {
    manager =
        PolicyManagerHelper.narrow(orb.resolve_initial_references("ORBPolicyManager"));
}
catch(org.omg.CORBA.ORBPackage.InvalidName e) {}

//get a reference to the per-thread manager
org.omg.CORBA.PolicyManager current;
try {
    current=PolicyManagerHelper.narrow(orb.resolve_initial_references("PolicyCurrent"));
}
catch(org.omg.CORBA.ORBPackage.InvalidName e) {}

//set the policy on the orb level
try{
    manager.set_policy_overrides(myRebindPolicy,
        SetOverrideType.SET_OVERRIDE);
}
catch (InvalidPolicies e){}

// set the policy on the Thread level
try{
    current.set_policy_overrides(myRebindPolicy,
        SetOverrideType.SET_OVERRIDE);
}
catch (InvalidPolicies e){}

//set the policy on the object level:
org.omg.CORBA.Object oldObjectReference=bind(...);
org.omg.CORBA.Object
    newObjectReference=oldObjectReference._set_policy_override(myRebindPolicy,
        SetOverrideType.SET_OVERRIDE);
```

Configuration and management

This part of the VisiBroker for Java *Programmer's Guide* includes these chapters.

- Chapter 11 “Using the VisiBroker Console”
- Chapter 12 “Using the ORB Services browsers”
- Chapter 13 “Using the Server Manager”
- Chapter 14 “Setting properties”

Using the VisiBroker Console

This chapter describes the Inprise VisiBroker Console and provides brief descriptions of the VisiBroker ORB that can be accessed from the Console. Instructions on the interface's basic navigational features and those specific to the VisiBroker Console are covered in this chapter. Instructions on the navigational features specific to the individual ORB Services are covered in those chapters.

What is the VisiBroker Console?

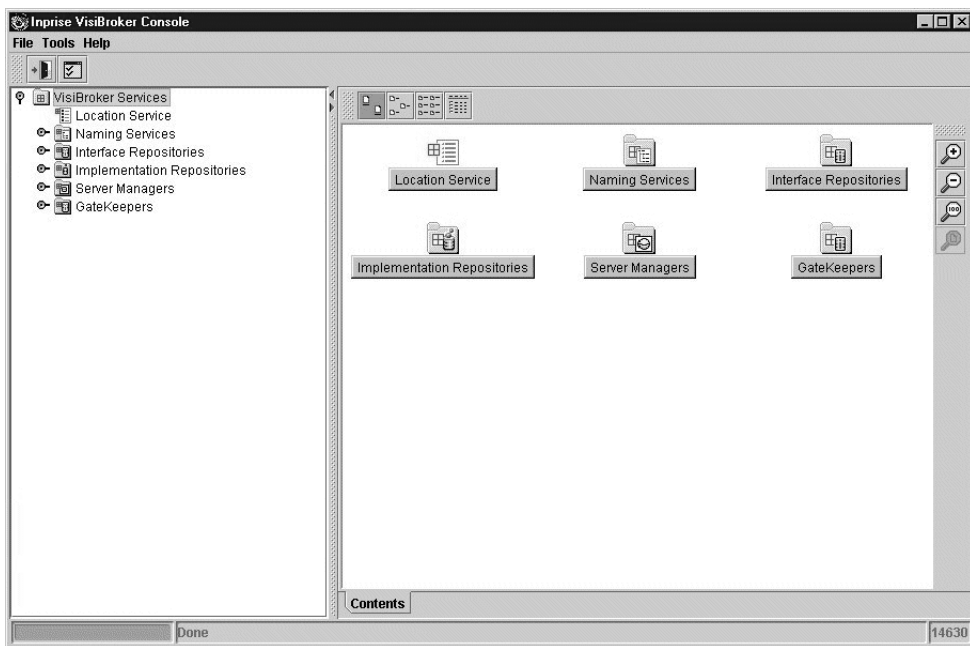
The Inprise VisiBroker Console is a tool that allows you to view and manage the VisiBroker ORB Services in a graphical interface. In particular, you can use the ORB Services browsers to manage object servers, control the configuration of gatekeepers, browse the interface repository, edit naming contexts, look up object instances, and view the OADs on your network.

The design of the VisiBroker Console is similar to the graphical interfaces of the Consoles in the Inprise Application Server and Inprise AppCenter products.

The VisiBroker Console provides browser support and is divided as follows into the following areas, which correspond to the ORB Services that it supports:

- Location Service
- Naming Services
- Interface Repositories
- Implementation Repositories
- Server Managers
- Gatekeepers

Figure 11.1 VisiBroker Console



Starting the VisiBroker Console

Use one of the following methods to start the Console:

- Windows**
- Choose the Start button on the taskbar, select Programs | VisiBroker | VisiBroker Console.
- or
- Run the `console.bat` file from the `vbroker/console/bin` directory
- UNIX**
- Run `console.sh` for the Bourne shell

Once the Console starts, the preferences that were configured during installation take effect. If you have problems, check the path and classpath settings.

Configuring the Console

You can set preferences for the Console at any time during a Console session. However, for some changes made during a Console session, you will be prompted to end that session and restart the Console for the new settings to take effect.

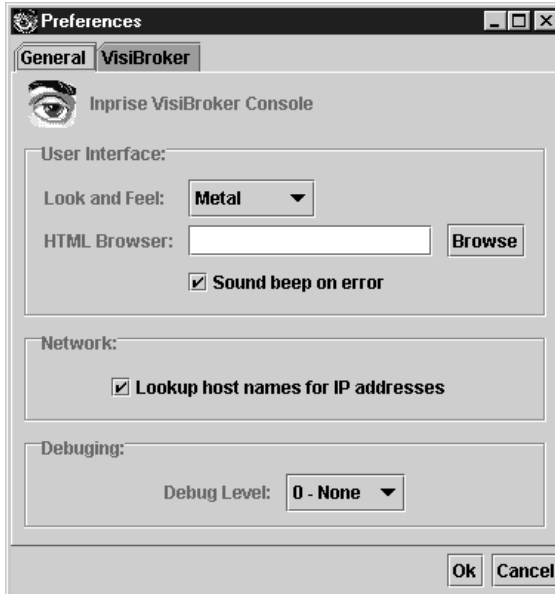
Setting preferences

You can use the Preferences dialog to change the configured settings, including the Smart Agent Port number setting, during your session. Make sure that the configuration is accurate and that the options are set correctly.

To set the general preferences for a Console session

- 1 During login, open the Preferences dialog box by choosing File | Preferences.

Figure 11.2 General tab of the Preferences dialog box



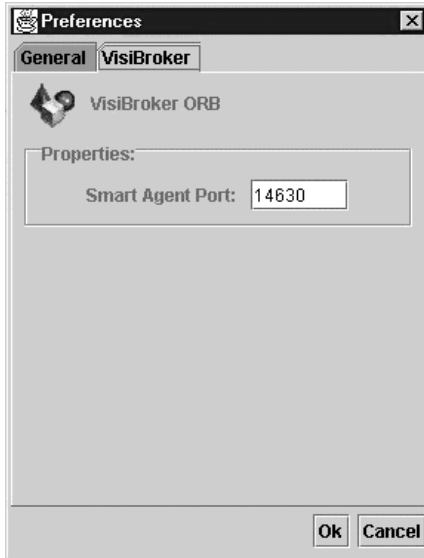
- 2 From the General tab, choose the options that you want to set as defaults for the Console.
 - In the User Interface section, the Look and Feel field allows you to configure the appearance of your display for a specific user interface style. Your choices include Metal, CDE/Motif, and Windows.
 - The HTML Browser field allows you to select the HTML browser in which you want to view the VisiBroker documentation.
 - In the Network section, check the Lookup Host Names for IP Addresses checkbox if you want to resolve the IP address name to host names.

If this box is checked, when you open the Services in the Console, you will see host names as well as IP addresses.

Note that if you select this option, the console will resolve IP addresses to domain names by looking them up in DNS. This can adversely effect performance, and also fail, if DNS is not configured correctly. Uncheck this box if you do not have access to DNS.

- In the Debugging section, the Debug Level field allows you to set the level of debug information that you see about your Console session. The higher the level, the more detailed the information you see. However, setting a high debug level may slow down the Console's response time.
- 3 Select the VisiBroker tab.

Figure 11.3 VisiBroker tab of the Preferences dialog box



- 4 From the VisiBroker tab, change the Smart Agent Port number of the ORB domain. You can specify any unused number from 1024 to 32767.
This value was specified when VisiBroker was installed
- 5 Choose OK to accept the changes.

Viewing system information

You can view information about your system by choosing File | System Information. This displays a System Information dialog which has the following three tabs:

- **General:** This tab displays platform and application information.
- **Java properties:** This tab displays the properties that you set for Java.
- **Plug-ins:** This tab displays information about any plug-ins you may be using including a description of the product and the path.

Navigating the VisiBroker Console

The VisiBroker Console has a typical Explorer-style user interface with elements such as menus, tools, and status bars; a navigation pane on the left side of the viewing area; and a content pane on the right side. You choose options from pull-down or context (right-click) menus to perform common functions; select specific ORB Services from the navigation pane; or perform tasks in the content pane (work area) related to the ORB Service that you select.

The Console's main window consists of the following elements that help you complete the tasks related to the specific ORB Service:

- Menu bar
- Toolbar
- Status bar
- Pull down or context menus
- Navigation pane
- Content pane

Menu bar

The menu bar is located at the top of the Console's main window. The menu bar provides you with some of the common navigational and management options in the Console.

Toolbar

The toolbar is located at the top of the Console main window, just under the menu bar. The toolbar lets you perform some of the Console functions with a single click of the mouse. Toolbar functions are dimmed when their functions are not available in a specific context.

Status bar

The status bar is located at the bottom of the main window of the Console. The status bar displays information about the status of your actions and also displays any warning messages for the current session.

Pull down or context menus

The pull down menus are located in the menu bar area, at the top of the Console's main window. The context menus display when you right-click an item on the Console. You can perform many common functions by either using pull-down or context (right-click) menus. In some cases, you have the option to use either menu to perform the same function.

Navigation pane

The Console's viewing area is divided into two major parts: the Navigation pane on the left side and the Content pane on the right side.

The Navigation pane shows you a hierarchical tree structure in which you can expand items to navigate to the next level. The hierarchical tree contains folders that represent the ORB Services.

Clicking these folders selects the Service and displays a browser to the right of the tree. Right-clicking provides a menu of possible actions on the folder. Once you click an item, the right side of the panel—the Content pane—shows you information about the item you just selected.

Content pane

The Content pane contains the content of the item you select in the Navigation pane. Depending on which item you select, different sets of tabs appear at the bottom of the Content pane. Selecting one of these tabs changes the information that appears in the Content pane.

Supported ORB Services

From the VisiBroker Console, you can view, configure, and manage the ORB Services by accessing a browser for each Service. To get to the ORB Services browsers, you click the VisiBroker Services folder in the navigation pane of the VisiBroker Console. This displays a folder for each ORB Service. From here, you select the specific service that you want to browse. A browser for that service appears in the content pane of the Console.

The VisiBroker Console supports the following ORB Services.

Location Service

The Location Service is the interface to the Smart Agent. This browser provides general purpose facilities for locating object instances and displays all instances of an object to which a client can bind. Also, it provides a list of all smart agents running on the current port.

For more information about the Location Service, see Chapter 17, "Using the Location Service."

Naming Services

The Naming Services browser displays, in a hierarchical format, the contents of the naming services running on your VisiBroker domain. From this browser, you can select, navigate, and edit naming contexts and name bindings.

For more information about the Naming Service, see Chapter 18, “Using the Naming Service.”

Interface Repositories

The Interface Repositories browser displays, in a hierarchical format, the contents of the interface repository on your VisiBroker domain. An interface repository is like a database of CORBA object interface information. The information in an interface repository is equivalent to the information in an IDL file.

For more information about the Interface Repositories, see Chapter 21, “Using interface repositories.”

Implementation Repositories

The Implementation Repositories browser shows a list of all object implementations registered with each OAD.

For more information about the Implementation Repositories, see Chapter 20, “Using the Object Activation Daemon.”

Server Manager

From within the Server Manger browser, an object server can publish its own properties. These properties appear in the Console. The ORB properties are published by default, but each server can hide or rearrange the containers, methods, or properties if it chooses to. This browser allows you to monitor and manage running servers, view the POA hierarchy, and set properties. The Server Manager interface replaces the VisiBroker 3.x ORB Manager interface. You can access the Server Manager interface from the VisiBroker Services area in the navigational pane of the VisiBroker and then selecting the Server Manager folder.

For more information about the Server Manager, see Chapter 13, “Using the Server Manager.”

Gatekeeper

The Gatekeeper browser displays a list of active Gatekeeper instances from which you select, to browse and configure their properties. The selected Gatekeeper instance displays in the content pane of the Console. You can access the Gatekeeper from the VisiBroker Services area in the navigational pane of the VisiBroker Console from the Gatekeepers folder.

For more information on the Gatekeeper and instructions on how to use the Gatekeeper browser within the VisiBroker Console, see the VisiBroker for Java *VisiBroker Gatekeeper Guide*.

Using the ORB Services browsers

Introduction

VisiBroker provides you with graphical interfaces for the ORB services from the Inprise VisiBroker Console. The VisiBroker Console is a tool that allows you to view and manage the ORB Services in a graphical interface. Using these interfaces, you can perform several related ORB functions that previously were performed at the command line. VisiBroker provides browsers for the following ORB services:

- Location Service
- Naming Services
- Implementation Repositories
- Interface Repositories

See Chapter 11, “Using the VisiBroker Console,” for more information about the VisiBroker Console. For additional information on using the services that relate to each browser, see the following chapters:

Service	See chapter
Location Service	Chapter 17, “Using the Location Service”
Naming Service	Chapter 18, “Using the Naming Service”
Implementation Repository	Chapter 20, “Using the Object Activation Daemon”
Interface Repository	Chapter 21, “Using interface repositories”

Location Service

The Location Service, the interface to the Smart Agent, is an extension to the CORBA specification that provides general-purpose facilities for locating object instances. The Location Service communicates directly with one Smart Agent which maintains a list of the instances it knows, along with information it knows about the instances.

When queried by the Location Service, a Smart Agent forwards the query to the other Smart Agents, and aggregates their replies in the result it returns to the Location Service. In this way, the Location Service sees all the object instances to which a client can bind. Smart Agents only know about instances with persistent object references that are accessible. An accessible instance is either an *active* object—an object whose server is running and has activated the object on its POA—or an *activable* object—an object that has been registered with an OAD.

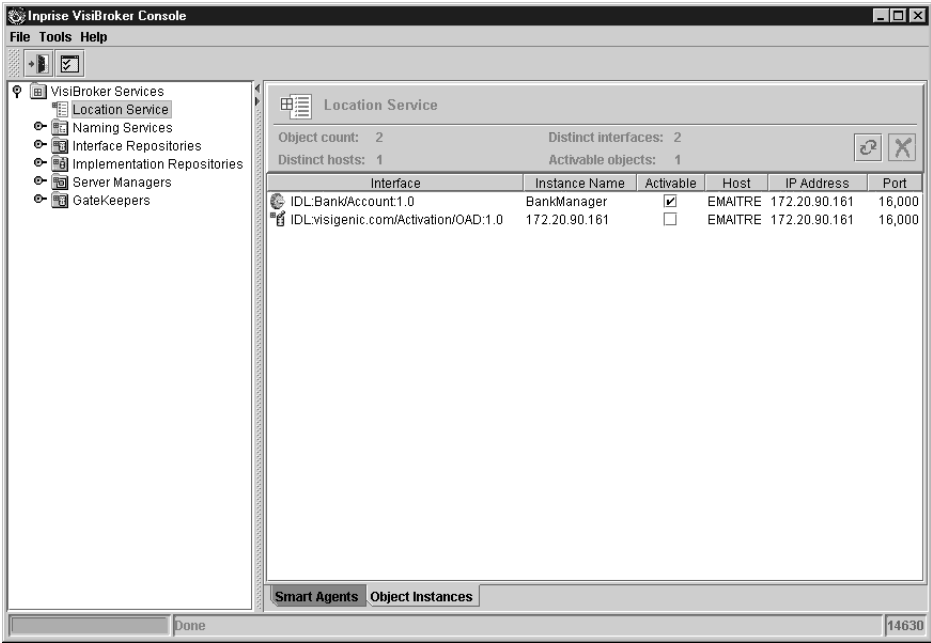
Accessing the Location Service browser

To use the Location Service browser

- 1 Expand the VisiBroker Services folder.
- 2 Select the Location Service icon.

The Location Service browser appears in the Content pane.

Figure 12.1 Console's Location Service browser



- 3 Choose the Smart Agent tab to view the Smart Agents running on your subnet. This list only displays the Smart Agents that were configured to run on the same port as your Console's Smart Agent port. The Smart Agent port number appears in the lower right-hand corner of the Console.
- 4 Choose the Object Instances tab to view the object instances running on your subnet.

The Location Service browser lists each active object instance and the following object attributes:

Name	Description
Interface	Interface exposed by the object instance.
Instance Name	The name of the object instance.
Activatable	Whether the object instance has been registered (true) with an Object Activation Daemon (OAD), or has been started by some other means (false).
Host	The name of the host on which the object instance is running or the domain name, if the Look Up Host Names for IP Addresses checkbox was selected in the Preferences dialog box.
IP Address	IP address of the host machine on which the object instance is running.
Port	Port number, on the host, that is associated with the object instance.

Refreshing the active object list

The Location Service browser searches for all active objects registered with Smart Agents in your CORBA network. You can refresh the list any time you want to see objects that have been activated since the last time you refreshed the display.

To refresh the list of active objects, click the Refresh button. The time required to complete the search depends on the size of your CORBA network.

Naming Services

The Naming Services browser is a graphical tool that displays, in hierarchical form, the contents of the naming service running on your CORBA network. It provides an intuitive, visual interface that allows you to select, navigate, and edit naming contexts and name bindings.

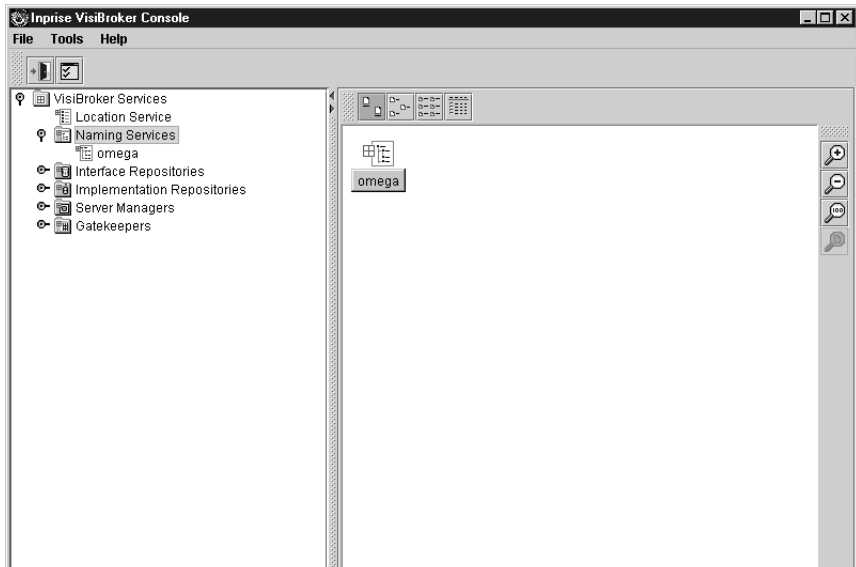
The VisiBroker Naming Services for Java and C++ are full implementations of the OMG's CORBA Naming Service specification. They enable developers to register object names at runtime. Client applications can then use the Naming Service to discover the names of objects they want to use simply by traversing the naming context graph.

Accessing the Naming Services

To use the Naming Services browser

- 1 Expand the VisiBroker Services area.
- 2 Expand the Naming Services folder.
- 3 Choose the Naming Service you want to browse.
- 4 The Naming Services browser appears in the Content pane.

Figure 12.2 Console's Naming Services browser with the Naming Service selected

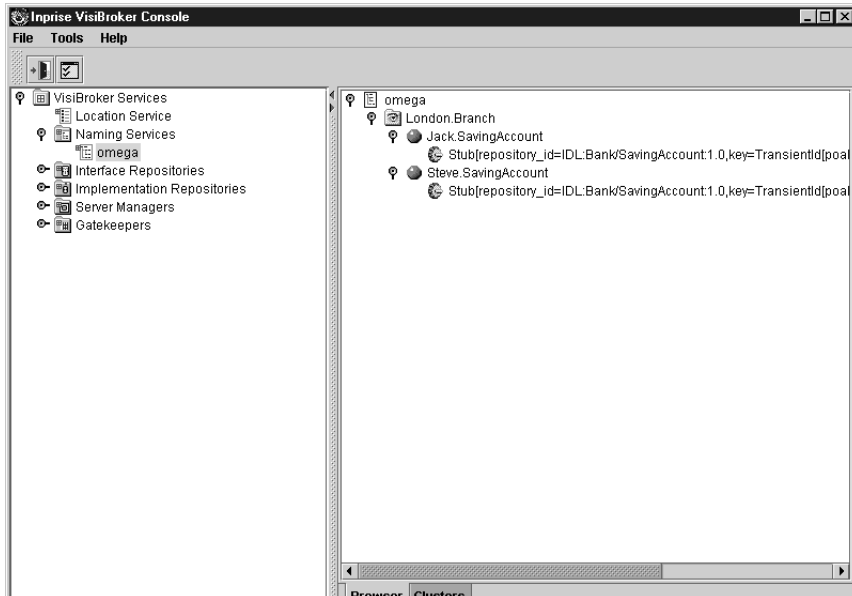


Browsing the Naming Service

To browse the Naming Service

- 1 Expand or collapse the context of the tree as desired.

Figure 12.3 Console's Naming Service browser with the context selected



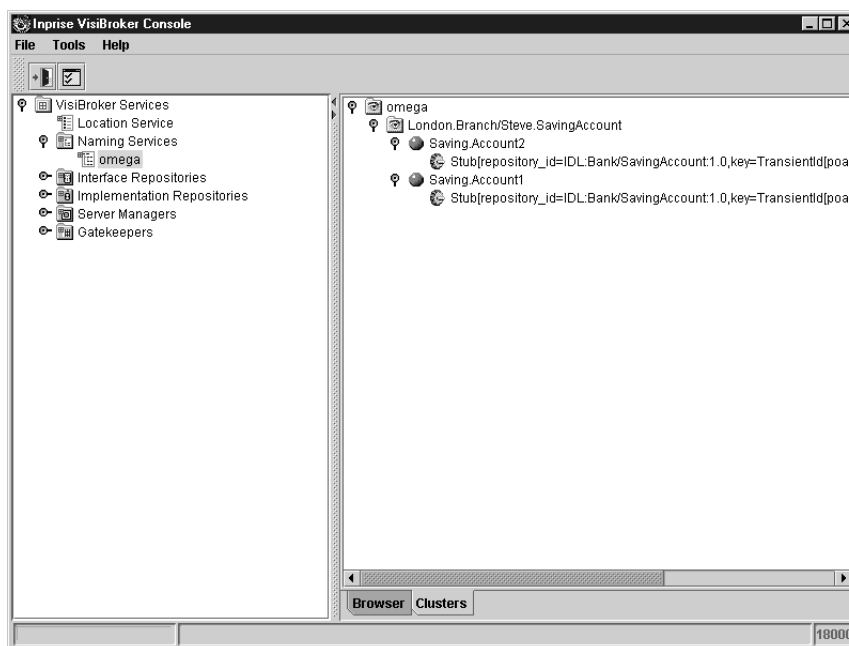
This example shows the bindings for the London.Branch context and the objects that are associated with each binding.

Browsing the VisiBroker Naming Service clusters

The clustering feature of the VisiBroker Naming Service allows you to associate a number of object bindings with a single name. See “Clusters” on page 18-20 for more information on clustering.

To browse the Naming Service clusters and see their associated object bindings

- 1 Select the cluster tab in the Naming Service browser.
- 2 Expand and collapse the clusters for the associated object bindings.

Figure 12.4 Console's Naming Service browser showing a cluster with two objects

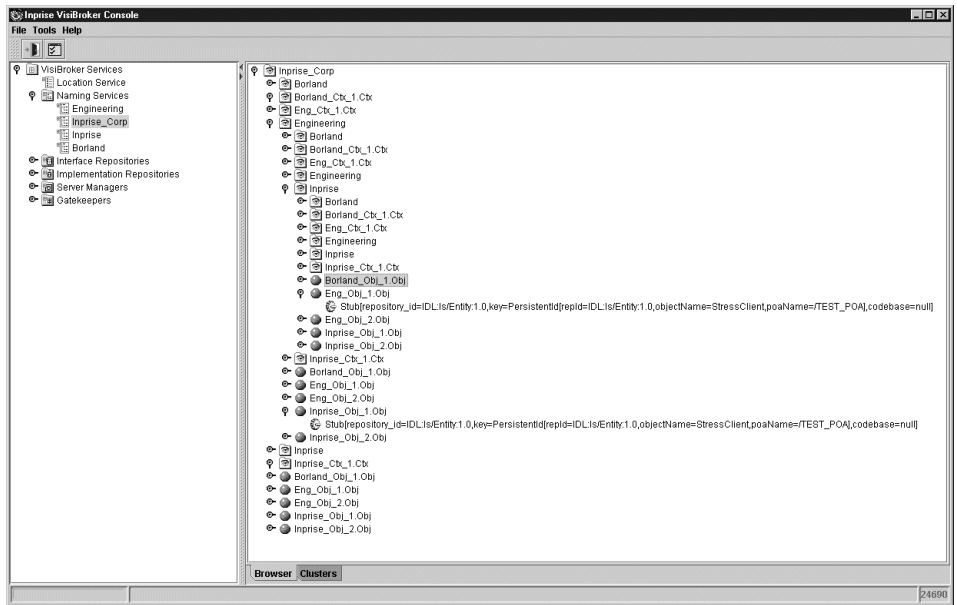
Browsing the VisiBroker Naming Service federations

The federation feature of the VisiBroker Naming Service allows you to link together information in naming services, thereby allowing name servers to make references to one another. This means that one name server is linked in such a way that a name can be resolved across the different name servers.

The Smart Agent must be configured to allow the objects to be resolved across different domains. See “Configuring the Naming Service” on page 18-6, and the VisiBroker Naming Service FAQ for more information on setting up the Naming Service to use federations at runtime.

To browse the federations in the Naming Services

- 1 Expand the Naming Services folder to view the list of Naming Services running on your domain.
- 2 Select the federated Naming Service.
- 3 Expand the Naming Service in the content pane to view the federations.
- 4 Using the right click menu, you can add a new context and delete or rename existing ones and delete or rename objects.

Figure 12.5 Console's Naming Services browser showing a list of federations

Implementation Repositories

The Implementation Repositories browser provides a visual interface to implementation repositories managed by Object Activation Daemons (OADs) on your CORBA network. An implementation repository contains the object implementations that are registered with an OAD. You can use the Implementation Repositories browser to view these object implementations, their state of activation, and activation policies.

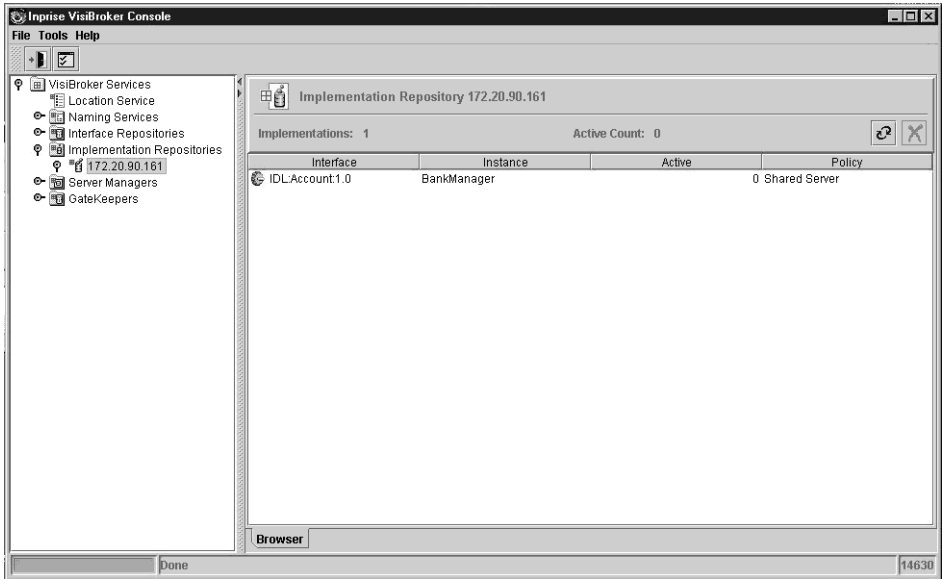
Accessing the Implementation Repositories

To use the Implementation Repositories browser

- 1 Expand the VisiBroker Services area.
- 2 Expand the Implementation Repositories folder to view the OADs running on your subnet.
- 3 Select the Implementation Repository that you want to browse.

The selected repository appears in the Content pane.

Figure 12.6 Console's Implementation Repositories browser



Interface Repositories

An interface repository (IR) contains descriptions of CORBA interfaces. The data in an IR is the same as in IDL files—descriptions of modules, interfaces, operations, and parameters—but it's organized for runtime access by clients. A client can browse an interface repository (perhaps serving as an online reference tool for developers) or can look up the interface of any object for which it has a reference (perhaps in preparation for invoking the object with the dynamic invocation interface).

An interface repository (IR) is like a database of CORBA object interface information. In contrast to the Location Services, which holds data describing object instances, an IR's data describes interfaces (types). There may or may not be instances satisfying the interfaces in an IR. The information in an IR is equivalent to the information in an IDL file (or files), but it is represented in a way that is easier for clients to use. Summarizing, an IR enables clients to learn about or update interface descriptions at runtime.

Clients that use interface repositories may also use the VisiBroker Dynamic Invocation Interface (DII). Such clients use an interface repository to learn about an unknown object's interface, and they use the DII to invoke methods on the object. However, there is no necessary connection between an IR and the DII.

Viewing an Interface Repository

The Interface Repository browser is a graphical tool that displays, in hierarchical form, the contents of one or more interface repositories stored on your CORBA network.

Interface repositories can contain any of the following types of definitions:

Object Type	Description
Repository	Top-level module.
ModuleDef	Logical grouping of interfaces.
InterfaceDef	Interface definition.
AttributeDef	Attribute definition associated with an interface.
OperationDef	Operation definition associated with an interface.
TypedefDef	Type definition for a named type (other than primitives or interfaces), such as enumerated types.
ConstantDef	Named constant definition.
ExceptionDef	Exception definition for an exception triggered by an operation.
StructDef	An IDL structure definition.
UnionDef	An IDL union definition.
EnumDef	An IDL enumeration definition.
AliasDef	An IDL typedef that aliases another definition.
StringDef	An IDL bounded string type.
SequenceDef	An IDL sequence type
ArrayDef	An IDL array type.
PrimitiveDef	One of the IDL primitive types
WstringDef	An IDL wide string.

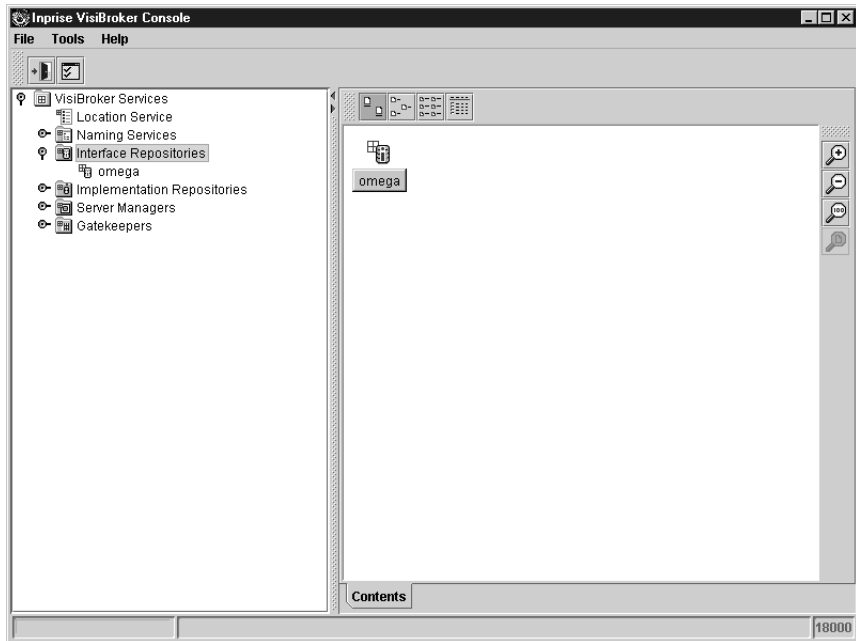
For more detailed information about interface repositories, see Chapter 21, “Using interface repositories,” of this documentation or Chapter 6, “The Interface Repository,” of the *The CORBA 2.0 Specification* (97-02-05).

Accessing the Interface Repositories

To use the Interface Repositories browser

- 1 Expand the VisiBroker Services area.
- 2 Expand the Interface Repositories folder.
- 3 Choose the repository you want to browse.
- 4 The Interface Repositories browser appears in the Content pane.

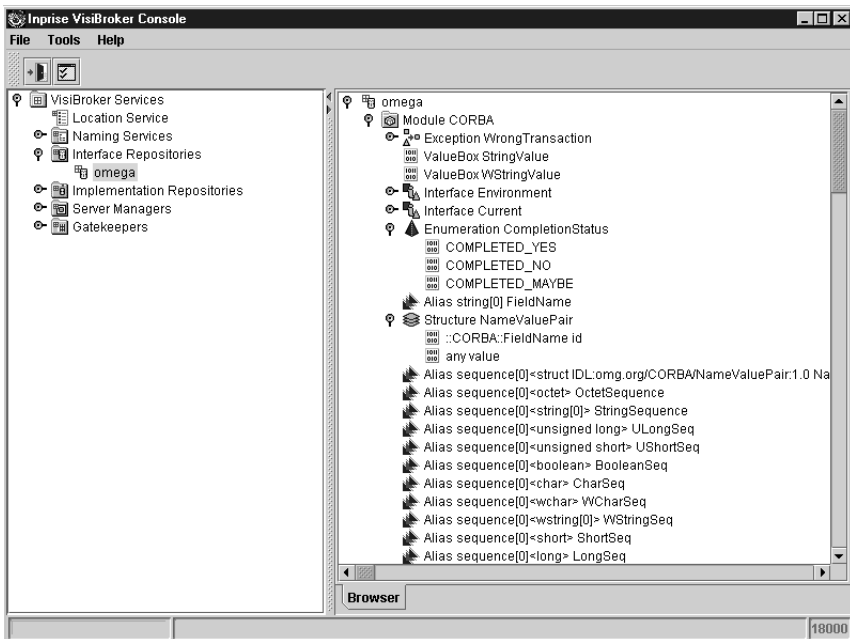
Figure 12.7 Console's Interface Repositories browser with the repository selected



Browsing the Interface Repositories

To browse the Interface Repository, expand or collapse the module as desired.

Figure 12.8 Console's Interface Repositories browser with the repository expanded



Using the Server Manager

This chapter describes the VisiBroker Server Manager that is used to manage object servers and explains how to use the Server Manager browser to easily perform these related tasks.

What is the Server Manager

The VisiBroker Server Manager allows your client applications to monitor and manage object servers, view and set properties at runtime for those servers, and view and invoke methods on server manager objects. The Server Manager has an element known as a container which represents each major ORB component. A container can contain properties, operations, and other containers.

The Server Manager replaces the VisiBroker 3.x ORB Manager and this interface is where you set the attributes of a server application.

For example, the top level container corresponds to the ORB, which in turn contains ORB properties, a shutdown method, and other containers such as the RootPOA, Agent, and the OAD. For more information on the top level container, see “Viewing the top-level container” on page 13-2.

Each container provides a number of methods you can invoke, as well as properties that you can get and set.

The Server Manager can support the properties added by other ORB Services such as the Gatekeeper.

Viewing the top-level container

The top level container or root container, which represents the server object in the Console, does not support any properties and operations. It just contains the ORB container. When you choose the Server Manager browser, the top level container appears in the tree. You cannot replace the top level container.

You can expand this container to view its contents in the Server Manager browser.

Server Manager browser

The Server Manager browser allows you to easily monitor and manage object servers using the browser and to see a graphical representation of the information from within the VisiBroker Console. For more information on the VisiBroker Console, see Chapter 11, “Using the VisiBroker Console.”

With the Server Manager browser, you can traverse through the hierarchy of Portable Object Adaptors (POAs). From within the hierarchy, you can get properties for a particular POA, manage the POAs, see the various properties of the server you selected, set properties for the current session that you are running or for future sessions, or invoke methods for selected containers.

Using the VisiBroker 4.x example server

To manage a server in the Server Manager browser, you need a VisiBroker 4.x server. An example is provided with the VisiBroker distribution. The example server is called Bank Agent. It provides you with a sample server.

The example is found in the following location:

```
<install_directory>/examples/basic/bank_agent
```

To use the example with the Server Manager browser you must:

Windows

- 1 Run the command `vbmake` to compile the file.

Solaris

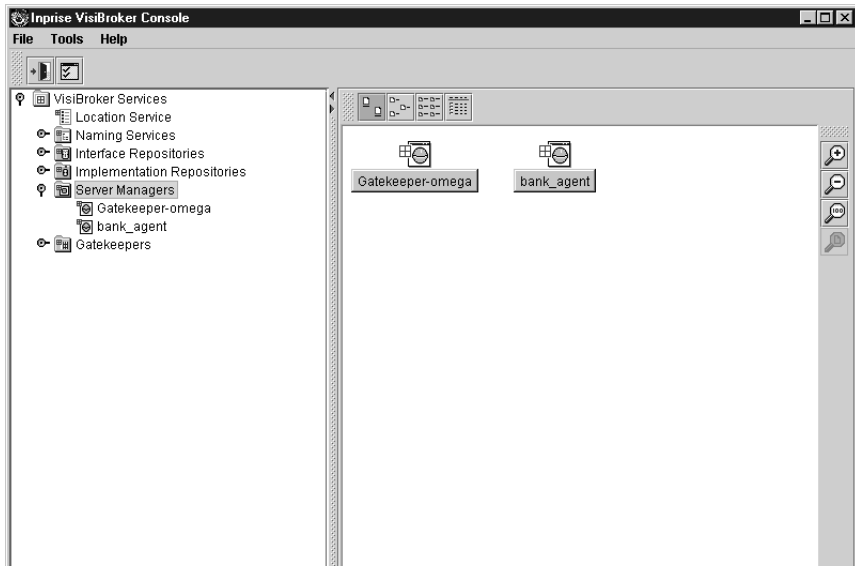
Run `make` to compile the file.

This produces a `Server.class` file.

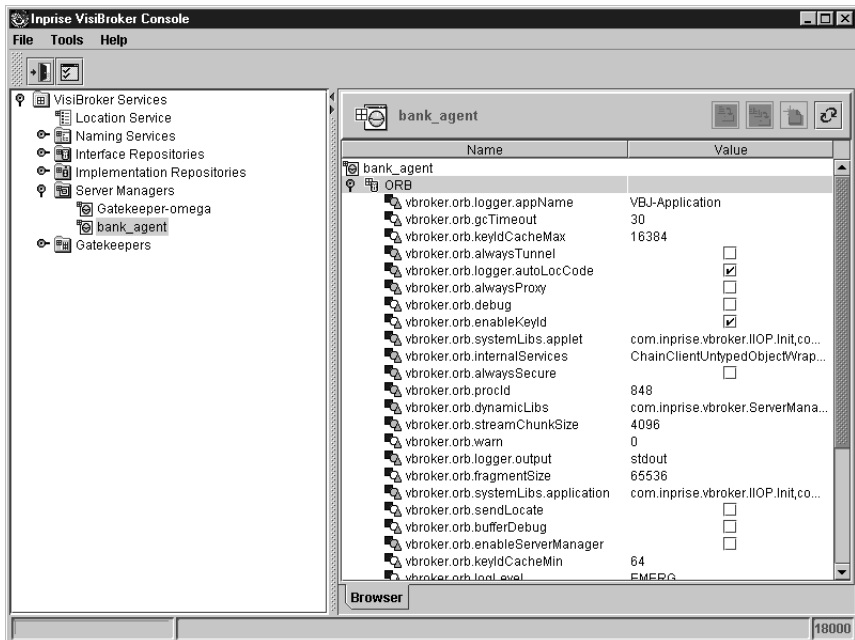
- 2 Enter the following at the command line to run the server:

```
vbj -Dvbroker.orb.enableServerManager=true -Dvbroker.serverManager.name=<name> <server_name>
```

This exposes the server objects in the Server Manager browser.

Figure 13.1 Server Manager browser

- 3 Choose the Bank Agent to view its contents in the right pane.

Figure 13.2 Server Manager with Bank Agent Server object selected

Setting security for the Server Manager

There are two levels of security that the Server Manager supports. These are:

- Enabling the Server Manager on the server side for access by the client.
- Setting the following flags to control what can be done to a server through the client:

- `vbroker.ServerManager.enableOperations=true`

If this flag is set to false, the client will not be able to invoke `do_operations` on any container.

- `vbroker.ServerManager.enablesSetProperty=true`

If this flag is set to false, the client will not be able to set a property on any container. However, it can still get the property.

Also, you can install a security interceptor that can compare the client credentials with the required credentials using an authentication server.

Using the Server Manager browser

The Server Manager browser displays in the Inprise VisiBroker Console. When you choose the Server Managers folder, a top-level container for your server object appears in the tree. You can view the contents of the server by clicking this container. Although the contents of the container will vary depending on the particular server object that you select, most likely you will see sub-containers, root POA, Smart Agent, and OAD.

The appearance of the browser or context area changes depending on which sub-container you select. Each choice displays some information relating to the selection. For example, the top level container for a server object provides the methods which you can invoke from within the Server Manager browser.

Viewing the contents of a server

To view the contents of the server manager from within the Inprise VisiBroker Console:

- 1 Expand the VisiBroker Services folder to view the list of available VisiBroker services.

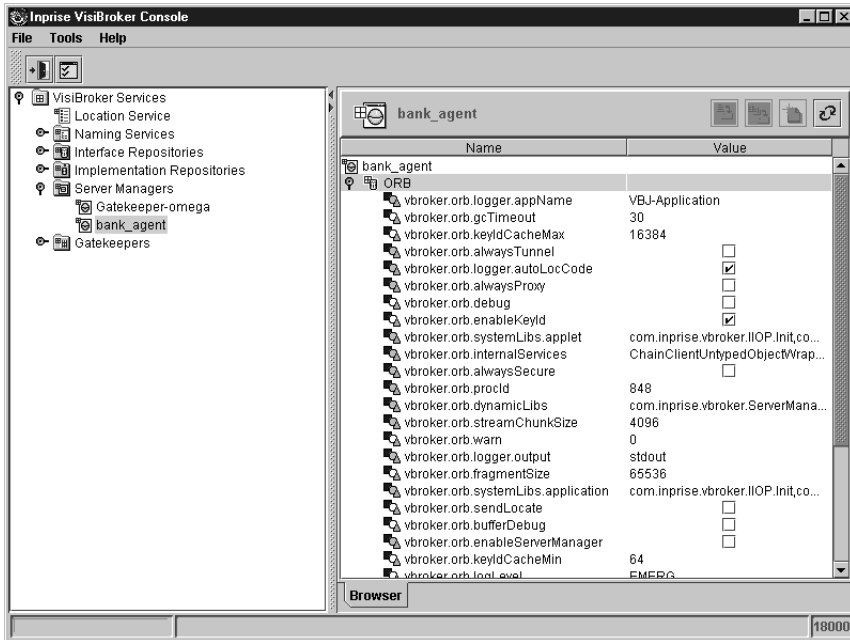
2 Expand the Server Managers folder to view the list of enabled servers.

You must enable the server manager before you can view it in the Server Manager browser. For instructions on enabling a server manager, see “Enabling the server” on page 13-5.

3 Choose the server object that you want to manage from the list of enabled servers.

The expanded container for the server looks something like the following sample.

Figure 13.3 Server Manager browser with selected server object



Enabling the server

Before you can view a server in the Server Manager browser, you must enable the server manager using the following command-line prompt:

```
vbj -Dvbroker.orb enableServerManager=true -Dvbroker.serverManager.name=<server_manager_name> <server_name>
```

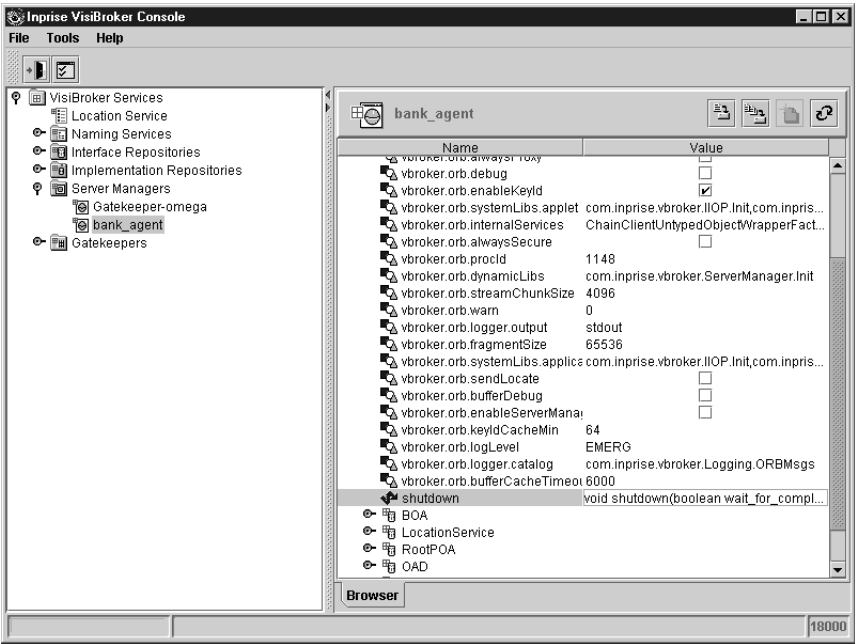
By default, the server is not enabled for management until you enable it in this way.

Invoking methods

In the Server Manager browser, each node in the tree represents either a container, property, or method/operation.

- 1 To invoke a method, double click the Value field associated with the method.

Figure 13.4 Server Manager browser with the server method's associated value field selected



- 2 In the Invoke Server Method dialog box, enter the parameter values and click the Invoke button to invoke the method or exit the dialog box.

Figure 13.5 Invoke Server Method dialog box appears

Setting properties

From within the Server Manager browser, you can set properties for the server objects on a per session or future session basis. The per session changes take effect in the current session that you are running and the future session changes are stored in a file for later use. These settings take effect the next time the server is run.

The ORB container can only support persistent changes in future sessions if you specify a storage file name on the command line using the ORBpropStorage option and if the file that you specified already exists. The ORB container saves any changes that you make during a session to that file. If the file does not exist, the ORB container throws a `StorageException`.

Property types

The Server Manager supports the following types of properties:

- **READ ONLY:** These properties cannot be changed in session, nor stored for future sessions. These types deal with statistics like the current number of connections, or bytes of incoming data for a session.
- **READ ONLY IN SESSION:** These properties can be changed for future sessions, but cannot be changed for a current session. These types typically include properties that are used in initialization and cannot be changed at runtime. For example, the Agent port number used by the server.
- **READ AND WRITE:** These properties can be changed for a current session and also saved for future sessions.

When you view the properties in the Server Manager browser, only the READ ONLY IN SESSION and READ AND WRITE properties have editable fields. You can change

these properties and then determine which property setting type you want to assign to the change. See “Changing property settings” on page 13-8 for more information.

Specifying the property storage file

The property storage file must be specified before you enable the server, as the server reads the property settings in this file, when it is initialized. If the file is not specified, the server will not be aware of these settings during the session. Also, this file must be specified each time you enable the server if you want to use those property settings or save changes to the settings. For example, if you are running the server and it crashes, when you restarted the server, you also must specify the storage file name again.

If you make changes to the property settings during a Server Manager session, and save these changes to the property storage file, that property storage file name must be specified the next time you start the server or those changes will not take effect.

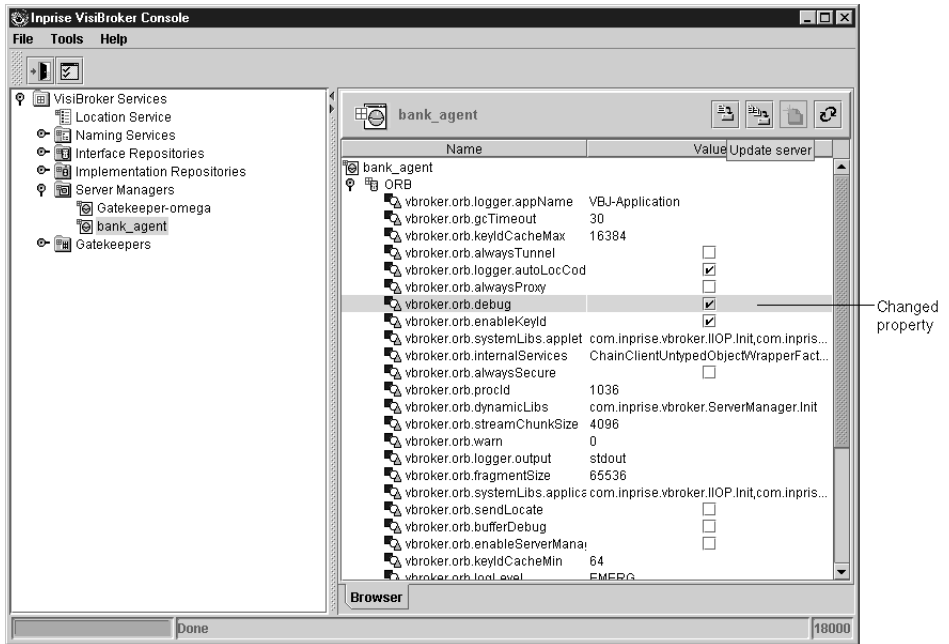
To enable a server, specify a name for the Server Manager, and specify the storage file, where you can store properties for future sessions, enter the following command line option.

```
vbj -Dvbroker.orb.enableSeverManager=true -Dvbroker.serverManager.name=BigBoss -  
-ORBpropStorage=<property_file-name> Server
```

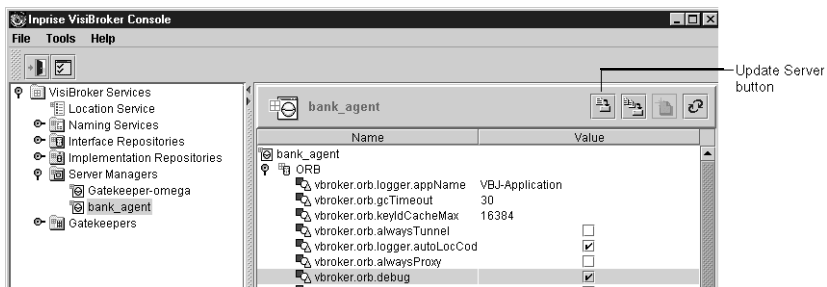
Changing property settings

To change the property settings in the Server Manager browser:

- 1 Double click a property in the Server Manager browser pane.

Figure 13.6 Clicking on a property in the Server Manager browser

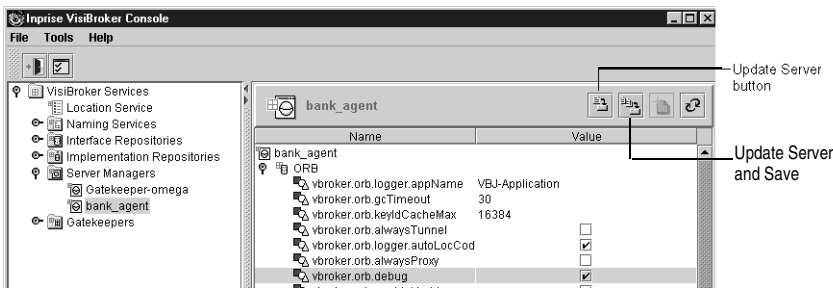
- 2 In the Server Manager browser pane, make your change on the same line as the property (in place editing) if the property is Read Only in Session or Read and Write. A check appears on the same line as the changed property.
- 3 Choose the following options from the toolbar buttons:
 - Update Server

Figure 13.7 Server Manager toolbar—Update Server button

This button updates the server properties with your changed properties but does not save the changes to a file. This button will only be enabled if you change a property. Use this feature to fine tune the server properties before saving the changes to the file. The server will not recognize any Read Only in Session Properties properties until you persist the changes and restart the server.

- Update Server and Save Properties to File

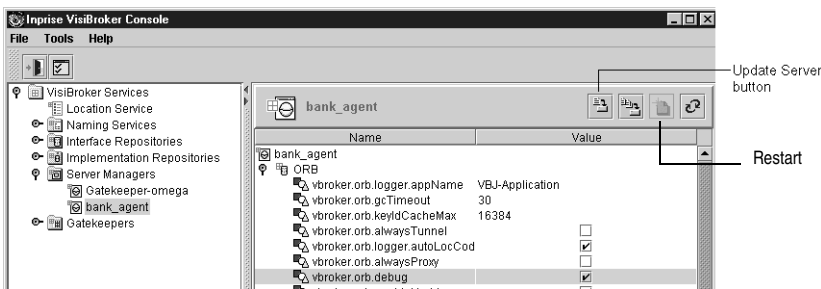
Figure 13.8 Server Manager toolbar—Update Server and Properties to File button



This button updates the server and instructs the server to save the properties to the property storage file. See “Specifying the property storage file” on page 13-8 for instructions on specifying a storage file. If the property storage file has not been set up, this button will be disabled.

- Restart Server

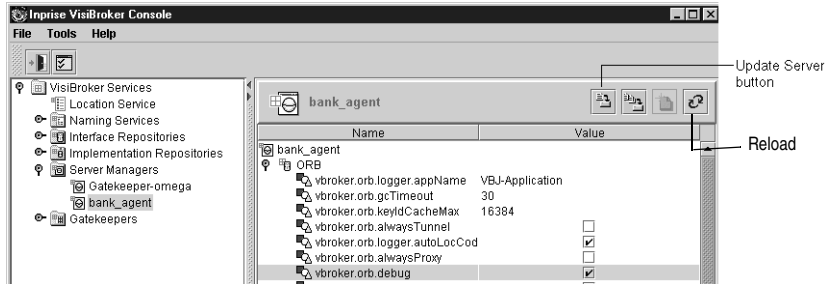
Figure 13.9 Server Manager toolbar—Restart Server button



This button instructs the server to restart. It is only enabled if the server publishes a restart method in its root container.

- Reload Properties from Server

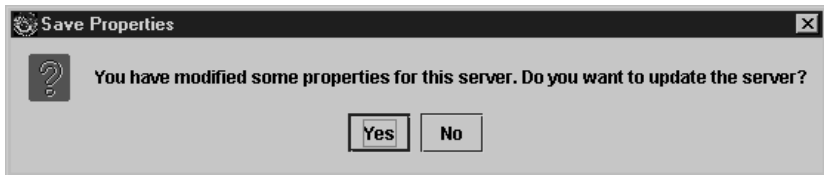
Figure 13.10 Server Manager toolbar—Reload Properties from Server button



This button reloads the properties from the server and refreshes the view.

- 4 When you exit the Server Manager browser without saving your changes, it prompts you to save them by displaying the Save Properties dialog box.
- 5 This will update the server and persist the properties to a property file, if the storage file has been set up.
- 6 Click OK to save changes.

Figure 13.11 Server Manager's Save Properties dialog box



Setting properties

This chapter describes how to set ORB properties through a property text file or as command-line arguments.

Overview

Each ORB has properties that define its characteristics. For example, `vbroker.agent.debug` directs the ORB to turn on output of debugging information for all communication with the Smart Agent. Each property has a predetermined data type, such as string, unsigned long and boolean; and a value that describes that property for the ORB. For example, `vbroker.agent.enableLocator=false` disables lookups to the smart agent.

When the ORB starts its initialization process, many of these properties are read.

You can specify ORB properties in a property text file or in command-line arguments when starting applications. A statement in the properties file may look like the following:

Code sample 14.1 Excerpt from a properties file

```
vbroker.agent.enableLocator=false
```

Specifying a property in a command-line argument could look like:

Code sample 14.2 Example of setting properties through command-line arguments

```
vbj -Dvbroker.agent.port=5024 Server
```

The order in which these properties take precedence is listed in “Property precedence under NT and Unix” on page 14-5 and “Property precedence for applets” on page 14-5.

Properties are read when `ORB.init()` is called. Once the properties are stored in memory within the Property Manager, the files or command-line arguments are no longer referenced.

Setting Visibroker properties

Properties may be set in the following ways:

- Shell/console environment
- Windows registry (NT only)
- Command line arguments (the first parameter of ORB.init)
- Applet parameters (the first parameter of ORB.init)
- System properties
- Programmatically via `ORB.init` (the second parameter)
- Property file via `ORBpropStorage` option
- Default property file `ORB.properties` inside `vbjorb.jar`

When properties are set using the first two options, they are converted to system properties.

Shell/console environment variables

Several properties can be set as environment variables. They are automatically picked up when the program starts. VisiBroker for Java converts them to system properties as follows:

Environmental variable	Property
OSAGENT_ADDR	vbroker.agent.addr
OSAGENT_ADDR_FILE	vbroker.agent.addrFile
OSAGENT_PORT	vbroker.agent.port
VBROKER_ADM	vbroker.orb.admDir

Following are examples of how to set environment variables:

Code sample 14.3 Setting environment variables

UNIX

```
setenv OSAGENT_PORT 10000
setenv VBROKER_ADM /usr/local/vbroker/adm
```

Windows

```
set OSAGENT_PORT=10000
set VBROKER_ADM=c:\c:\Inprise\VBroker\adm
```

Note For more information about setting VisiBroker environment variables, see Chapter 3, “Setting up your environment.”

Windows registry

You can also put environmental variables into the Windows registry. VisiBroker provides a tool called `vregedit` to make it easier to modify these entries.

Environmental variables in the Windows registry are converted into system properties in the same way as environment variables. However, the environment settings take higher precedence than registry entries.

Command line arguments

Any property listed in the properties file can be set through command-line arguments.

Code sample 14.4 Setting properties from the command-line

```
vbj -Dvbroker.agent.port=1024 Server
```

Note The `-D` is required and indicates that the string following it is a property.

A property set at the command line overrides that property's setting in Properties.

Command line arguments passed to an application are passed as parameters to the application class. For example, the command:

```
vbj Server -vbroker.agent.port 15000
```

passes property `vbroker.agent.port` with value `15000` to an application called `Server`. You can have more than one property on the parameter list.

Only properties starting with `ORB`, `OA`, and `vbroker.` can be specified in this way. In addition, `ORB` and `OA` properties are migrated to appropriate `vbroker.` properties except for `ORBpropStorage`.

These settings are passed as the first parameter of `ORB.init`.

Applet parameters

Parameters to an applet are specified using the html `<param>` tag:

```
<applet ...>
  <param name="vbroker.agent.port" value="15000">
</applet>
```

Each param defines one property. No migration is performed here. These settings are passed as the first parameter of `ORB.init`.

Code sample 14.5 Setting applet properties from the HTML

```
. . .
<applet class = "MyCORBAApplet.class"
  codebase = ". ."
  archive="my Lib.jar"
  <param name = "vbroker.orb.alwaysTunnel" value = "true">
  <param name = "vbroker.orb.gatekeeper.ior" value = "">
</applet>
. . .
```

System properties

Properties defined using `-D` are set as system properties by the Java virtual machine:

```
vbj -Dvbroker.agent.port=15000 Server
```

Two alternatives to `-D` are:

```
-J-D<name>=<value>
-VBJprop <name>=<value>
```

These differ from command line arguments because they are parsed by the Java virtual machine and not by VisiBroker itself. Settings from the environment and registry are converted into system properties and passed to JVM.

Programmatically via ORB.init

ORB.init accepts a parameter of type `java.util.Properties`. You can use this parameter to pass in a set of properties.

Property file via ORBpropStorage option

The name of a property file can be supplied in a command line argument, as a system property, or an applet parameter by using the property name `ORBpropStorage`. Inside the file, properties are listed line by line. Each line takes the form:

```
<property name>=<property value>
```

Empty lines and comments (lines starting with `#`) are ignored.

Properties file

The properties file is a text file with the following format:

```
property_name=value
```

The ORB has a predefined set of property names for you to use. These names are case-sensitive, so make sure you type the names exactly as they are listed. The default properties file, `ORB.properties`, is free-form, meaning that you can enter properties in any order as long as you follow the correct format. However, the file can be read more easily if you break the properties into logical groups. You can label each group with a comment line. Comment lines are any lines that start with a pound sign (`#`).

Code sample 14.6 Example of grouping properties

```
# OSAgent properties
vbroker.agent.debug=false
vbroker.agent.addr=null
vbroker.agent.port=14000
vbroker.agent.addrFile=null
vbroker.agent.enableLocator=true
```

There are only three property data types.

- String

- Unsigned long
- Boolean

If the string value is null, you can enter `null` as the property value:

Code sample 14.7 Setting a null value

```
vbroker.repository.name=null
```

If the value is boolean, enter `true` or `false`.

Code sample 14.8 Setting a boolean value

```
vbroker.agent.enableLocator=true
```

To use your properties, place them in a file and reference them through the following command-line argument:

```
-DORBpropStorage=filename
```

filename can be a relative or absolute path.

Code sample 14.9 Specifying a property file

```
vbj -DORBpropStorage=myprops Server
```

Note The `-D` is required and indicates that the string following it is a property.

Default properties file

The file `ORB.properties`, located in `vbjorb.jar`, shows the default properties.

Property precedence under NT and Unix

Properties are set in the following order (from highest priority to lowest priority):

- 1 Command line arguments
- 2 System properties (NT environment settings come before NT registry settings)
- 3 Property file (`ORBpropStorage`)
- 4 Properties programmatically passed to `ORB.init`
- 5 Default property file `ORB.properties`

Property precedence for applets

Properties are set in the following order (from highest priority to lowest priority):

- 1 Applet parameters
- 2 Property file (`ORBpropStorage`); may from an URL
- 3 Programmatically passed to `ORB.init`
- 4 Default property file `ORB.properties`

VisiBroker for Java properties

For a list of the properties available in VisiBroker for Java, see Appendix B, “Using VisiBroker properties,” in the VisiBroker for Java *Reference*.

For information about BOA properties, See Chapter 31, “Using the BOA with VisiBroker 4.x.”

For information about the server engine properties, see Chapter 7, “Using POAs.”



Tools and services

This part of the *VisiBroker for Java Programmer's Guide* includes these chapters.

- | | |
|------------|--------------------------------------|
| Chapter 15 | "Using IDL" |
| Chapter 16 | "Using the Smart Agent" |
| Chapter 17 | "Using the Location Service" |
| Chapter 18 | "Using the Naming Service" |
| Chapter 19 | "Using the Event Service" |
| Chapter 20 | "Using the Object Activation Daemon" |
| Chapter 21 | "Using interface repositories" |

Using IDL

This chapter describes how to use the CORBA interface description language (IDL).

Introduction to IDL

The Interface Definition Language (IDL) is a *descriptive language* (not a programming language) to describe the interfaces being implemented by the remote objects. Within IDL, you define the name of the interface, the names of each of the attributes and methods, and so forth. Once you've created the IDL file, you can use an IDL compiler to generate the client stub file and the server skeleton file in the Java programming language. For more information on the VisiBroker `idl2java` compiler see Chapter 3, "IDL to Java mapping," in the VisiBroker for Java *Reference*.

The OMG has defined specifications for such language mapping. Information about the language mapping is not covered in this manual since VisiBroker adheres to the specification set forth by OMG. If you need more information about language mapping, see the OMG web site at <http://www.omg.org/>.

Note The CORBA formal specification can be found at <http://www.omg.org/corba/corba11op.html>. See Chapter 24 for mapping of OMG IDL to Java.

Discussions on the IDL can be quite extensive. Because VisiBroker adheres to the specification defined by OMG, you can visit the OMG site for more information about IDL.

How the IDL compiler generates code

You use the Interface Definition Language (IDL) to define the object interfaces that client programs may use. The `idl2java` compiler uses your interface definition to generate code.

For details on usage syntax for the `idl2java` compiler, see the *VisiBroker for Java Reference*.

Example IDL specification

Your interface definition defines the name of the object as well as all of the methods the object offers. Each method specifies the parameters that will be passed to the method, their type, and whether they are for input or output or both.

IDL sample 15.1 shows an IDL specification for an object named `example`. The `example` object has only one method, `op1`.

IDL sample 15.1 Example IDL specification

```
// IDL specification for the example object
interface example {
    long op1(in char x, out short y);
};
```

Looking at the generated code

The IDL compiler generates several files from the above example IDL.

- `_exampleStub.java` is the stub code for the `example` object on the client side.
- `example.java` is the `example` interface declaration.
- `exampleHelper.java` declares the `exampleHelper` class, which defines helpful utility functions and support functions for the `example` interface.
- `exampleHolder.java` declares the `exampleHolder` class, which provides a holder for passing out and inout parameters.
- `exampleOperations.java` defines the methods in the `example` interface and is used both on the client and the server side. It also works together with the tie classes to provide the tie mechanism.
- `examplePOA.java` contains the skeleton code (implementation base code) for the `example` object on the server side.
- `examplePOATie.java` contains the class used to implement the `example` object on the server side using the tie mechanism.

`<interface name>Stub.java`

For each user-defined type, a stub class is created by the `idl2java` compiler. This is the class which is instantiated on the client side which implements the `<interface name>` interface.

Code sample 15.1 Example of the stub class code

```
public class exampleStub extends com.inprise.vbroker.CORBA.portable.ObjectImpl
    implements example {
    final public static java.lang.Class _opsClass = exampleOperations.class;
    public java.lang.String[] ids () {
        . . .
    }
    public int op1 (char x, org.omg.CORBA.ShortHolder y) {
        . . .
    }
}
```

`<interface name>.java`

The `<interface name>.java` file is the Java interface generated for each IDL interface. This is the direct mapping of the IDL interface definition to the appropriate Java interface. This interface is then implemented by both the client and server skeleton.

Code sample 15.2 Example of the interface declaration code

```
public interface example extends com.inprise.vbroker.CORBA.Object,
    exampleOperations,
    org.omg.CORBA.portable.IDLEntity {
}
```

`<interface name>Helper.java`

For each user-defined type, a helper class is created by `idl2java`. The helper class is an abstract class with various static methods for the generated Java interface.

Code sample 15.3 Example of the helper class code

```
public final class exampleHelper {
    public static example narrow (final org.omg.CORBA.Object obj) {
        . . .
    }
    public static example unchecked_narrow (org.omg.CORBA.Object obj) {
        . . .
    }
    public static example bind (org.omg.CORBA.ORB orb) {
        . . .
    }
    public static example bind (org.omg.CORBA.ORB orb, java.lang.String name) {
        . . .
    }
}
```

Looking at the generated code

```
public static example bind (org.omg.CORBA.ORB orb, java.lang.String name,
    java.lang.String host,
    com.inprise.vbroker.CORBA.BindOptions _options) {
    . . .
}
public static example bind (org.omg.CORBA.ORB orb, java.lang.String fullPoaName,
    byte[] oid) {
    . . .
}
public static example bind (org.omg.CORBA.ORB orb,
    java.lang.String fullPoaName, byte[] oid,
    java.lang.String host,
    com.inprise.vbroker.CORBA.BindOptions _options) {
    . . .
}
public java.lang.Object read_Object (final org.omg.CORBA.portable.InputStream istream) {
    . . .
}
public void write_Object (final org.omg.CORBA.portable.OutputStream ostream,
    final java.lang.Object obj) {
    . . .
}
public java.lang.String get_id () {
    . . .
}
public org.omg.CORBA.TypeCode get_type () {
    . . .
}
public static example read (final org.omg.CORBA.portable.InputStream _input) {
    . . .
}
public static void write (final org.omg.CORBA.portable.OutputStream _output,
    final example value) {
    . . .
}
public static void insert (final org.omg.CORBA.Any any, final example value) {
    . . .
}
public static example extract (final org.omg.CORBA.Any any) {
    . . .
}
public static org.omg.CORBA.TypeCode type () {
    . . .
}
public static java.lang.String id () {
    . . .
}
}
```


<interface name>Holder.java

For each user-defined type, a holder class is created by the `idl2java` compiler. It provides a class for an object which wraps objects which support the *<interface name>* interface when passed as `out` and `inout` parameters.

Code sample 15.4 Example of the Holder class

```
public final class exampleHolder implements org.omg.CORBA.portable.Streamable {
    public foo.example value;
    public exampleHolder () {
    }
    public exampleHolder (final foo.example _vis_value) {
        . . .
    }
    public void _read (final org.omg.CORBA.portable.InputStream input) {
        . . .
    }
    public void _write (final org.omg.CORBA.portable.OutputStream output) {
        . . .
    }
    public org.omg.CORBA.TypeCode _type () {
        . . .
    }
}
```

<interface name>Operations.java

For each user-defined type, an operations class is created by the `idl2java` compiler which contains all the methods defined in the IDL declaration.

Code sample 15.5 Example of the operation code

```
public interface exampleOperations {
    public int op1 (char x, org.omg.CORBA.ShortHolder y);
}
```

<interface name>POA.java

The *<interface name>*POA.java file is the server-side skeleton for the interface. It unmarshals in parameters and passes them in an upcall to the object implementation and marshals back the return value and any `out` parameters.

Code sample 15.6 ExamplePOA.java file

```
public abstract class examplePOA extends org.omg.PortableServer.Servant implements
    org.omg.CORBA.portable.InvokeHandler, exampleOperations {
    public example _this () {
        . . .
    }
    public example _this (org.omg.CORBA.ORB orb) {
        . . .
    }
}
```

```

    public java.lang.String[] _all_interfaces (final org.omg.PortableServer.POA poa,
        . . .
    }
    public org.omg.CORBA.portable.OutputStream _invoke (java.lang.String opName,
        org.omg.CORBA.portable.InputStream _input,
        org.omg.CORBA.portable.ResponseHandler handler) {
        . . .
    }
    public static org.omg.CORBA.portable.OutputStream _invoke (exampleOperations _self,
        int _method_id, org.omg.CORBA.portable.InputStream _input,
        org.omg.CORBA.portable.ResponseHandler _handler) {
        . . .
    }
}

```

<interface name>POATie.java

The <interface name>POATie.java file is a delegator implementation for the <interface name> interface. Each instance of the tie class must be initialized with an instance of an implementation class that implements the <interface name>Operations class to which it delegates every operation.

Code sample 15.7 Example POATie file

```

public class examplePOATie extends examplePOA {
    public examplePOATie (final exampleOperations _delegate) {
        . . .
    }
    public examplePOATie (final exampleOperations _delegate,
        final org.omg.PortableServer.POA _poa) {
        . . .
    }
    public exampleOperations _delegate () {
        . . .
    }
    public void _delegate (final exampleOperations delegate) {
        . . .
    }
    public org.omg.PortableServer.POA _default_POA () {
        . . .
    }
    public int op1 (char x, org.omg.CORBA.ShortHolder y) {
        . . .
    }
}

```

Defining interface attributes in the IDL

In addition to operations, an interface specification can also define attributes as part of the interface. By default, all attributes are *read-write* and the IDL compiler will generate two methods—one to set the attribute's value, and one to get the attribute's value. You can also specify *read-only* attributes, for which only the reader method is generated.

IDL sample 15.2 shows an IDL specification that defines two attributes—one read-write and one read-only. Code sample 15.8 shows the operations class generated for the interface declared in the IDL.

IDL sample 15.2 IDL specification with two attributes—one read-write and one read-only

```
interface Test {
    attribute long count;
    readonly attribute string name;
};
```

Code sample 15.8 Code generated for the testOperations interface

```
public interface TestOperations {
    public int count ();
    public void count (int count);
    public java.lang.String name ();
}
```

Specifying oneway methods with no return value

IDL allows you to specify operations that have no return value, called *oneway* methods. These operations may only have input parameters. When a *oneway* method is invoked, a request is sent to the server but there is no confirmation from the object implementation that the request was actually received. VisiBroker uses TCP/IP for connecting clients to servers. This provides reliable delivery of all packets so the client can be sure the request will be delivered to the server, as long as the server remains available. Still, the client has no way of knowing if the request was actually processed by the object implementation itself.

Note Oneway operations cannot raise exceptions or return values.

IDL sample 15.3 Defining a oneway operation

```
interface oneway_example {
    oneway void set_value(in long val);
};
```

Specifying an interface in IDL that inherits from another interface

IDL allows you to specify an interface that inherits from another interface. The classes generated by the IDL compiler will reflect the inheritance relationship. All methods, data type definitions, constants and enumerations declared by the parent interface will be visible to the derived interface.

IDL sample 15.4 Example of inheritance in an interface specification

```
interface parent {
    void operation1();
};
interface child : parent {
    . . .
    long operation2(in short s);
};
```

Code sample 15.9 shows the code that is generated from the interface specification shown in IDL sample 15.4.

Code sample 15.9 Code generated from IDL sample 15.4

```
public interface parentOperations {
    public void operation1 ();
}
public interface childOperations extends parentOperations {
    public int operation2 (short s);
}
public interface parent extends com.inprise.vbroker.CORBA.Object, parentOperations,
    org.omg.CORBA.portable.IDLEntity {
}
public interface child extends childOperations, Baz.parent,
    org.omg.CORBA.portable.IDLEntity {
}
```

Using the Smart Agent

This chapter describes the Smart Agent (`osagent`), which client programs register with in order to find object implementations. This chapter explains how to configure your own ORB domain, connect Smart Agents on different local networks, and migrate objects from one host to another.

What is the Smart Agent?

VisiBroker's Smart Agent (`osagent`) is a dynamic, distributed directory service that provides facilities used by both client programs and object implementations. A Smart Agent must be started on at least one host within your local network. When your client program invokes `bind()` on an object, the Smart Agent is automatically consulted. The Smart Agent locates the specified implementation so that a connection can be established between the client and the implementation. The communication with the Smart Agent is completely transparent to the client program.

If the `PERSISTENT` policy is set on the POA, and `activate_object_with_id` is used, the Smart Agent registers the object or implementation so that it can be used by client programs. When an object or implementation is deactivated, the Smart Agent removes it from the list of available objects. As with client programs, the communication with the Smart Agent is completely transparent to the object implementation.

Locating Smart Agents

VisiBroker locates a Smart Agent for use by a client program or object implementation using a broadcast message. The first Smart Agent to respond is used. After a Smart Agent has been located, a point-to-point UDP connection is used for sending registration and look-up requests to the Smart Agent. The UDP protocol is used because it consumes fewer network resources than a TCP connection. All

registration and locate requests are dynamic, so there are no required configuration files or mappings to maintain.

Note Broadcast messages are used only to locate a Smart Agent. All other communication with the Smart Agent makes use of point-to-point communication. See “Using point-to-point communications” on page 16-8 for information on how to override the use of broadcast messages.

Locating objects through Agent cooperation

When a Smart Agent is started on more than one host in the local network, each Smart Agent will recognize a subset of the objects available and communicate with other Smart Agents to locate objects it cannot find. If one of the Smart Agent processes should terminate unexpectedly, all implementations registered with that Smart Agent discover this event and they will automatically reregister with another available Smart Agent.

Cooperating with the OAD to connect with objects

Object implementations may be registered with the OAD so that they can be started on demand. Such objects are registered with the Smart Agent as if they are actually active and located within the OAD. When a client requests one of these objects, it is directed to the OAD. The OAD then forwards the client request to the *real* (possibly newly) spawned server. The Smart Agent does not know that the real object implementation is not *actually* active within the OAD.

Starting a Smart Agent (osagent)

At least one instance of the Smart Agent should be running on a host in your local network. Local network refers to a subnetwork within which broadcast message can be sent.

To start the Smart Agent under Windows, select its icon from the VisiBroker Program Group or enter the following command at the DOS prompt:

WinNT prompt> osagent [options]

To start the Smart Agent on a UNIX system, enter the following command.

UNIX prompt> osagent &

The `osagent` command accepts the following command line arguments:

Option	Description
-p <i>UDP_port</i>	Overrides the setting of <code>OSAGENT_PORT</code> and the registry setting.
-v	Turns verbose mode on, which provides information and diagnostic messages during execution.
-help or -?	Prints the help message.
-n, -N	Disables system tray icon on Windows.

The following example of the `osagent` command specifies a particular UDP port:

Example `osagent -p 17000`

Verbose output

UNIX On a UNIX platform, the verbose output is sent to stdout.

WinNT On a Windows system, the verbose output is written to a log file stored at `<installation_location>\log\osagent.log` or to the directory specified by the `VBROKER_ADM` environment variable or use `OSAGENT_LOG_DIR` to specify a different directory to put the log.

The Smart Agent can be installed as an NT service under Windows NT (by choosing to do so during installation), allowing you to control it with the Service Manager provided with Windows NT. You may also start the Smart Agent in console mode from the DOS prompt entering the following command:

```
prompt> osagent
```

Disabling the agent

Communication with the Smart Agent can be disabled by passing the ORB the property at runtime:

```
prompt> vbj -Dvbroker.agent.enableLocator=false
```

If you use string-to-object references, a naming service, or pass in a URL reference, the Smart Agent is not required, and can be disabled. If you pass in an object name to the `bind` method, you must use the Smart Agent.

Ensuring Agent availability

Starting a Smart Agent on more than one host within the local network allows clients to continue to bind to objects, even if one of the Smart Agents terminates unexpectedly. If a Smart Agent becomes unavailable, all object implementations registered with that Smart Agent will be automatically re-registered with another Smart Agent. If no Smart Agents are running on the local network, object implementations will continue retrying until a new Smart Agent can be contacted.

If a Smart Agent terminates, any connections between a client and an object implementation that were established before the Smart Agent terminated will continue without interruption. However, any new `bind()` requests issued by a client will cause a new Smart Agent to be contacted.

No special coding techniques are required to take advantage of these fault-tolerant features. You only need to make sure a Smart Agent is started on one or more hosts on the local network.

Checking client existence

A Smart Agent sends an “Are You Alive” message (often called a *heartbeat* message) to its clients every two minutes to verify the client is still connected. If the client does not respond, the Smart Agent assumes the client has terminated the connection.

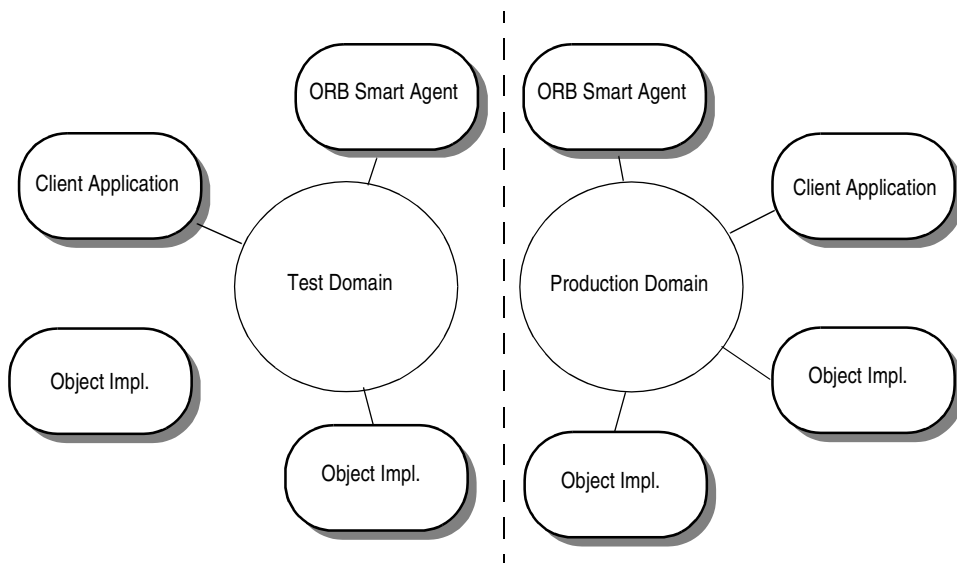
You can not change the interval for polling the client.

Note The use of the term “client” does not necessarily describe the function of the object or process. Any program that connects to the Smart Agent for object references is a client.

Working within ORB domains

It is often desirable to have two or more separate ORB domains running at the same time. One domain might consist of the production versions of client programs and object implementations while another domain might be made up of test versions of the same clients and objects that have not yet been released for general use. If several developers are working on the same local network, each may want to establish their own ORB domain so that their testing efforts do not conflict with one another.

Figure 16.1 Running separate ORB domains simultaneously



VisiBroker allows you to distinguish between multiple ORB domains on the same network by using a unique UDP port number for the Smart Agents for each domain. By default, the `OSAGENT_PORT` variable is set to 14000. If you wish to use a different port number, check with your system administrator to determine what port numbers are available. To override the default setting, the `OSAGENT_PORT` variable must be set accordingly before running a Smart Agent, an OAD, object implementations, or client programs assigned to that ORB domain.

Code sample 16.1 Setting the `OSAGENT_PORT` environment variable for a UNIX system running `csh`

```

prompt> setenv OSAGENT_PORT 5678
prompt> osagent &
prompt> oad &
  
```

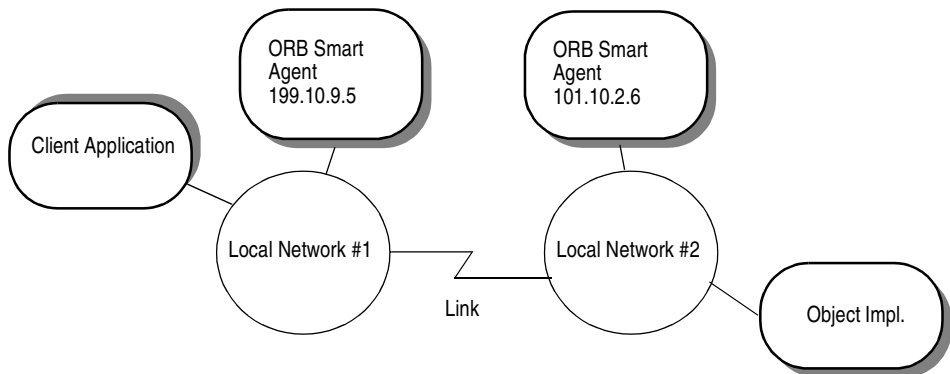

The Smart Agent also uses another port number internally. This port number can be set by using the `OSAGENT_CLIENT_HANDLER_PORT` environment variable. This port number is used for both TCP and UDP protocols and is the same for both.

Windows Setting the `OSAGENT_PORT` environment variable overrides the Windows registry setting or the setting set using the `vregedit.exe` utility program (located in the VisiBroker bin directory).

Connecting Smart Agents on different local networks

If you start multiple Smart Agents on your local network, they will discover each other by using UDP broadcast messages. Your network administrator configures a local network by specifying the scope of broadcast messages using the IP subnet mask. Figure 16.2 shows two local networks, connected by a network link.

Figure 16.2 Two Smart Agents on separate local networks



To allow the Smart Agent on one network to contact a Smart Agent on another local network, you must make the IP address of the remote Smart Agent available in a file named `agentaddr`. This is only necessary if the two Smart Agents can not detect each other through the UDP broadcast. Code sample 16.2 shows what this file would contain to allow the Smart Agent on Local Network #1 to connect to the Smart Agent on the other network. The path to this file is specified by the `VBROKER_ADM` environment variable that is set for the Smart Agent process. You can override this file name by setting the `OSAGENT_ADDR_FILE` environment variable.

Code sample 16.2 Content of the `agentaddr` file for the Smart Agent on network #1

```
101.10.2.6
```

With the appropriate `agentaddr` file, the client program on Network #1 could locate and use object implementations on Network #2. For more information on environment variables, see “Setting up your environment” on page 3-1.

Note If a remote network has multiple Smart Agents running, you should list the IP addresses of all of the Smart Agents on the remote network.

How Smart Agents detect each other

Suppose two agents, Agent 1 and Agent 2, are listening on the same UDP port from two different machines on the same subnet. Agent 1 starts before Agent 2. The following events occur:

- When Agent 2 starts, it UDP broadcasts its existence and sends a request message to locate any other Smart Agents.
- Agent 1 makes note that Agent 2 is available on the network and responds to the request message.
- Agent 2 makes note that another agent, Agent 1, is available on the network.

If Agent 2 is terminated gracefully (such as killing with *Ctrl+C*), Agent 1 is notified that Agent 2 is no longer available.

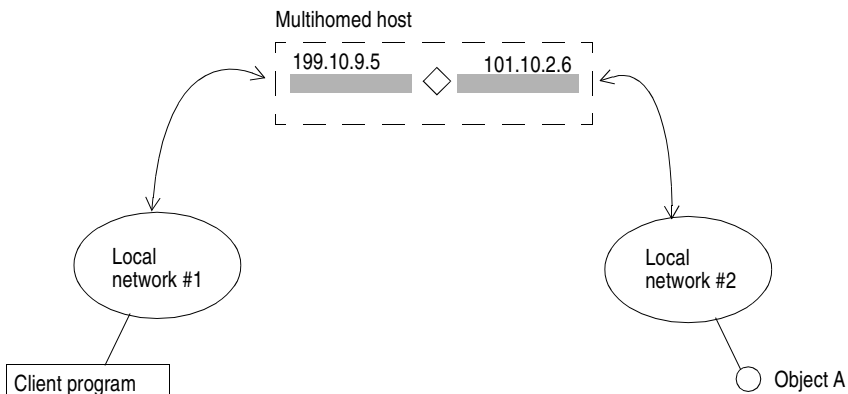
If Agent 2 is terminated abnormally (such as killing Agent 2 using the Task Manager window), Agent 1 is not notified that Agent 2 is no longer available (there is no periodic heartbeat messages between the agents). Agent 1 continues until a client asks for an object reference that does not exist in Agent 1's dictionary. Agent 1 forwards the request to Agent 2. Since Agent 2 is no longer available, Agent 1 is forced to clean up.

Until Agent 1 is forced to clean up, `osfind` still shows two agents listed and catches `ObjLocation::Fail` exception.

Working with multihomed hosts

When you start the Smart Agent on a host that has more than one IP address (known as a multihomed host) it can provide a powerful mechanism for bridging objects located on separate local networks. All local networks to which the host is connected will be able to communicate with a single Smart Agent, effectively bridging the local networks.

Figure 16.3 Smart Agent on a multihomed host



UNIX On a multihomed UNIX host, the Smart Agent dynamically configures itself to listen and broadcast on all of the host's interfaces which support point-to-point connections or broadcast connections. You may explicitly specify interface settings using the `localaddr` file as described in "Specifying interface usage for Smart Agents" below.

Windows On a multihomed Windows host, the Smart Agent is not able to dynamically determine the correct subnet mask and broadcast address values. To overcome this limitation, you must explicitly specify the interface settings you want the Smart Agent to use with the `localaddr` file.

When you start the Smart Agent with the `-v` (verbose) option, each interface that the Smart Agent uses will be listed at the beginning of the messages produced. Code sample 16.3 shows the sample output from a Smart Agent started with the verbose option on a multihomed host.

Code sample 16.3 Verbose output from a Smart Agent started on a multihomed host

```
Bound to the following interfaces:
Address: 199.10.9.5 Subnet: 255.255.255.0 Broadcast:199.10.9.255
Address: 101.10.2.6 Subnet: 255.255.255.0 Broadcast:101.10.2.255
...
```

As shown in Code sample 16.3, the output shows the address, subnet mask, and broadcast address for each interface in the machine. For UNIX, this output should match the results from the UNIX command `ifconfig -a`.

If you wish to override these settings, you can specify this interface information in the `localaddr` file. See "Specifying interface usage for Smart Agents" below for details.

Specifying interface usage for Smart Agents

Note It is not necessary to specify interface information on a single-homed host.

You can specify interface information for each interface you wish the Smart Agent to use on your multihomed host in the `localaddr` file. The `localaddr` file should have a separate line for each interface that contains the host's IP address, subnet mask, and broadcast address. By default, VisiBroker searches for the `localaddr` file in the `VBROKER_ADM` directory. You can override this location by setting the `OSAGENT_LOCAL_FILE` environment variable to point to this file. Lines in this file that begin with a `"#"` character are treated as comments and ignored. Code sample 16.4 shows the contents of the `localaddr` file for the multihomed host listed above.

Code sample 16.4 Contents of an example `localaddr` file

```
#entries of format <address> <subnet_mask> <broadcast address>
199.10.9.5 255.255.255.0 199.10.9.255
101.10.2.6 255.255.255.0 101.10.2.255
```

UNIX Though the Smart Agent can automatically configure itself on a multihomed host running UNIX, you can use the `localaddr` file to explicitly specify the interfaces that your host contains. You can display all the available interface values for your UNIX host by using the following command:

```
prompt> ifconfig -a
```

Output from this command appears similar to the following:

```
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST> mtu 8232
    inet 127.0.0.1 netmask ff000000
le0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 1500
    inet 199.10.9.5 netmask ffffffff broadcast 199.10.9.255
le1: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 1500
    inet 101.10.2.6 netmask ffffffff broadcast 101.10.2.255
```

Windows The use of the `localaddr` file with multihomed hosts is required for hosts running Windows because the Smart Agent is not able to automatically configure itself. You can obtain the appropriate values for this file by accessing the TCP/IP protocol properties from the Network Control Panel. If your host is running Windows NT, the `ipconfig` command will provide the needed values. You run this command as follows:

```
prompt>ipconfig
```

Output from this command appears similar to the following:

```
Ethernet adapter El59x1:
    IP Address. . . . . : 199.10.9.5
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 199.10.9.1

Ethernet adapter Elnk32:
    IP Address. . . . . : 101.10.2.6
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 101.10.2.1
```

Using point-to-point communications

VisiBroker provides you with three different mechanisms for circumventing the use of UDP broadcast messages for locating Smart Agent processes. When a Smart Agent is located with any of these alternate approaches, that Smart Agent will be used for all subsequent interactions. If a Smart Agent cannot be located using any of these alternate approaches, the ORB will revert to using the broadcast message scheme to locate a Smart Agent.

Specifying a host as a runtime parameter

Code sample 16.5 shows how you can specify the IP address where a Smart Agent is running as a runtime parameter for your client program or object implementation. Since specifying an IP address will cause a point-to-point connection to be established, you can even specify an IP address of a host located outside your local network. This mechanism takes precedence over any other host specification.

Code sample 16.5 Specifying a Smart Agent's IP address as a runtime parameter

```
prompt> vbj -Dvbroker.agent.addr=<ip_address> Server
```

You can also specify the IP address through the properties file. Look for the `vbroker.agent.addr` entry.

Code sample 16.6 Specifying the Smart Agent's IP address in the properties file

```
vbroker.agent.addr=<ip_address>
```

By default, `vbroker.agent.addr` in the properties file is set to `NULL`.

You can also list the hostnames where the agent might reside and then point to that file with the `vbroker.agent.addrFile` option in the properties file.

Specifying an IP address with an environment variable

You can specify the IP address of a Smart Agent by setting the `OSAGENT_ADDR` environment variable prior to starting your client program or object implementation. This environment variable takes precedence if a host is not specified as a runtime parameter.

Figure 16.4 Setting the `OSAGENT_ADDR` environment variable using the C shell

UNIX `prompt> setenv OSAGENT_ADDR 199.10.9.5`
 `prompt> client`

Win95 To set the `OSAGENT_ADDR` environment variable on a Windows 95 system, you can add the following line to your `autoexec.bat` file:

```
SET OSAGENT_ADDR=199.10.9.5
```

WinNT To set the `OSAGENT_ADDR` environment variable on a Windows NT system, you can use the System control panel and edit the environment variables:

- 1 Under System Variables, select any current variable.
- 2 Type `OSAGENT_ADDR` in the Variable edit box.
- 3 Type the IP address in the Value edit box. For example, `199.10.9.5`.

Specifying hosts with the `agentaddr` file

Your client program or object implementation can use the `agentaddr` file, described in “Connecting Smart Agents on different local networks” on page 16-5, to circumvent the use of UDP broadcast message to locate a Smart Agent. Simply create a file containing the IP addresses or fully qualified hostname of each host where a Smart Agent is running and then set the `OSAGENT_ADDR_FILE` environment variable to point to the path of the file. When a client program or object implementation has this environment variable set, the ORB will try each address in the file until a Smart Agent is located. This mechanism has the lowest precedence of all the mechanisms for specifying a host. If this file is not specified, the `VBROKER_ADM/agentaddr` file is used.

Ensuring object availability

You can provide fault tolerance for objects by starting instances of those objects on multiple hosts. If an implementation becomes unavailable, the ORB will detect the loss of the connection between the client program and the object implementation and will automatically contact the Smart Agent to establish a connection with another instance of the object implementation, depending on the effective rebind policy established by the client. See “Using quality of service” on page 10-6 for more information on establishing client policies.

Caution The rebind option must be enabled if the ORB is to attempt to reconnect the client with a replica object implementation. This is the default behavior.

Invoking methods on stateless objects

Your client program can invoke a method on an object implementation which does not maintain state without being concerned if a new instance of the object is being used.

Achieving fault-tolerance for objects that maintain state

Fault tolerance can also be achieved with object implementations that maintain state, but it will not be transparent to the client program. In these cases, your client program must either use the Quality of Service (QoS) policy `VB_NOTIFY_REBIND` or register an interceptor for the ORB object. For information on using QoS, see “Using quality of service” on page 10-6.

When the connection to an object implementation fails and the ORB reconnects the client to a replica object implementation, the `bind` method of the bind interceptor will be invoked by the ORB. The client must provide an implementation of this `bind` method to bring the state of the replica up to date. Interceptors are described in Chapter 24, “Using interceptors.”

Replicating objects registered with the OAD

The OAD ensures greater object availability because if the object goes down, the OAD will restart it. If you want fault tolerance for the case where a host becomes unavailable, the OAD must be started on multiple hosts, and the objects must be registered with each OAD instance.

Note The type of object replication provided by VisiBroker does not provide a multicast or mirroring facility. At any given time there is always a one-to-one correspondence between a client program and a particular object implementation.

Migrating objects between hosts

Object migration is the process of terminating an object implementation on one host, and then starting it on another host. Object migration can be used to provide load balancing by moving objects from overloaded hosts to hosts that have more resources or processing power (there is no load balancing between servers registered with different `osagents`.) Object migration can also be used to keep objects available when a host has to be shutdown for hardware or software maintenance.

Note The migration of objects that do not maintain state is transparent to the client program. If a client is connected to an object implementation that has migrated, the Smart Agent will detect the loss of the connection and transparently reconnect the client to the new object on the new host.

Migrating objects that maintain state

The migration of objects that maintain state is also possible, but it will not be transparent to a client program that has connected before the migration process begins. In these cases, the client program must register an interceptor for the object. When the connection to the original object is lost and the ORB reconnects the client to the object, the interceptor's `rebind_succeeded()` method will be invoked by the ORB. The client can implement this method to bring the state of the object up to date. Interceptors are described in Chapter 24, "Using interceptors."

Migrating instantiated objects

If the objects that you wish to migrate were created by a server process instantiating the implementation's class, you need only start it on a new host and terminate the server process. When the original instance is terminated, it will be unregistered with the Smart Agent. When the new instance is started on the new host, it will register with the Smart Agent. From that point on, client invocations will be routed to the object implementation on the new host.

Migrating objects registered with the OAD

If the ORB objects that you wish to migrate are registered with the OAD, you must unregister them with the OAD on the old host. Then, reregister them with the OAD on the new host. Here are the steps:

- 1 Unregister the object implementation from the OAD on the old host.
- 2 Register the object implementation with the OAD on the new host.
- 3 Terminate the object implementation on the old host.

See Chapter 20, "Using the Object Activation Daemon," for detailed information on registering and Unregistering object implementations.

Reporting all objects and services

The Smart Finder (`osfind`) command reports on all VisiBroker related objects and services which are currently available on a given network.

You can use `osfind` to determine the number of Smart Agent processes running on the network and the exact host on which they are executing. The `osfind` command also reports on all VisiBroker objects that are active on the network. You can use `osfind` to monitor the status of the network and locate stray objects during the debugging phase.

The `osfind` command has the following syntax:

Syntax `osfind [options]`

The following options are valid with `osfind`. If no options are specified, `osfind` lists all of the agents, OAD's, and implementations in your domain.

Option	Description
-a	Lists all Smart Agents in your domain.
-o	Lists all Object Activation Daemons in your domain.
-d	Prints hostnames as quad addresses

Windows `osfind` is a console application. If you start `osfind` from the Start menu, it runs until completion and exits before you can view the results.

Binding to Objects

Before your client application can invoke methods on an interface, it must first obtain an object reference using the `bind` method.

When your client application invokes the `bind` method, the ORB performs several functions on behalf of your application.

- The ORB contacts the `osagent` to locate an object server that is offering the requested interface. If an object name and a host name (or IP address) are specified, they will be used to further qualify the directory service search.
- When an object implementation is located, the ORB attempts to establish a connection between the object implementation that was located and your client application.
- If the connection is successfully established, the ORB will create a proxy object, if necessary, and return a reference to that object.

Note The ORB is not a separate process. It is a collection of classes and other resources that allow communication between clients and servers.

Using the Location Service

The VisiBroker Location Service provides enhanced object discovery that enables you to find object instances based on particular attributes. Working with VisiBroker Smart Agents, the Location Service notifies you of what objects are presently accessible on the network, and where they reside. The Location Service is a VisiBroker extension to the CORBA specification and is only useful for finding objects implemented with VisiBroker.

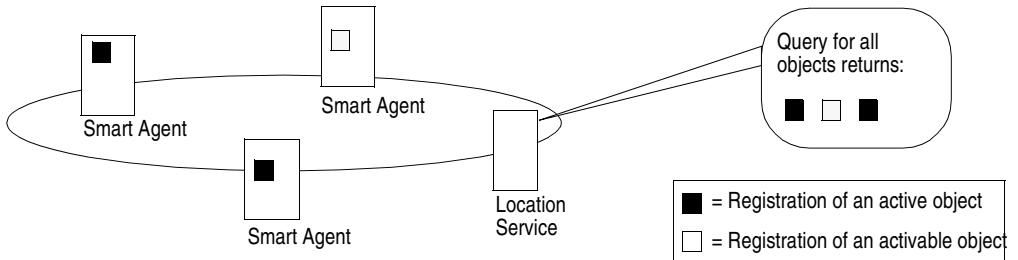
What is the Location Service?

The Location Service is an extension to the CORBA specification that provides general-purpose facilities for locating object instances. The Location Service communicates directly with one Smart Agent which maintains a *catalog*, which contains the list of the instances it knows about. When queried by the Location Service, a Smart Agent forwards the query to the other Smart Agents, and aggregates their replies in the result it returns to the Location Service.

The Location Service knows about all object instances that are registered on a POA with the `BY_INSTANCE` Policy and objects that are registered as persistent on a BOA. The server containing these objects may be started manually or automatically by the OAD.

The following diagram illustrates this concept.

Figure 17.1 Using the Smart Agent to find instances of objects



Note A server specifies an instance's scope when it creates the instance. Only globally-scoped instances are registered with Smart Agents.

The Location Service can make use of the information the Smart Agent keeps about each object instance. For each object instance, the Location Service maintains information encapsulated in the structure `ObjLocation::Desc` shown in IDL sample 17.1.

IDL sample 17.1 IDL for the Desc structure

```
struct Desc {
    Object ref;
    ::IIOP::ProfileBodyValue iiop_locator;
    string repository_id;
    string instance_name;
    boolean activable;
    string agent_hostname;
};
typedef sequence<Desc> DescSeq;
```

The IDL for the Desc structure contains the following information:

- The object reference, `ref`, is a handle for invoking the object.
- The `iiop_locator` interface provides access to the host name and the port of the instance's server. This information is only meaningful if the object is connected with IIOP, which is the only supported protocol. Host names are returned as strings in the instance description.
- The `repository_id`, which is the interface designation for the object instance that can be looked up in the Interface and Implementation Repositories. If an instance satisfies multiple interfaces, the catalog contains an entry for each interface, as if there were an instance for each interface.
- The `instance_name`, which is the name given to the object by its server.
- The `activable` flag, which differentiates between instances that can be activated by an OAD and instances that are started manually.
- The `agent_hostname`, the name of the Smart Agent with which the instance is registered.

The Location Service is useful for purposes such as load balancing and monitoring. Suppose that replicas of an object are located on several hosts. You could deploy a bind interceptor that maintains a cache of the host names that offer a replica and each host's recent load average. The interceptor updates its cache by asking the Location Service for the hosts currently offering instances of the object, and then queries the hosts to obtain their load averages. The interceptor then returns an object reference for the replica on the host with the lightest load. See Chapter 24, "Using interceptors," for more information about writing interceptors.

Location Service components

The Location Service is accessible through the `Agent` interface. Methods for the `Agent` interface can be divided into two groups: those that query a Smart Agent for data describing instances and those that register and unregister *triggers*. Triggers provide a mechanism by which clients of the Location Service can be notified of changes to the availability of instances.

What is the Location Service agent?

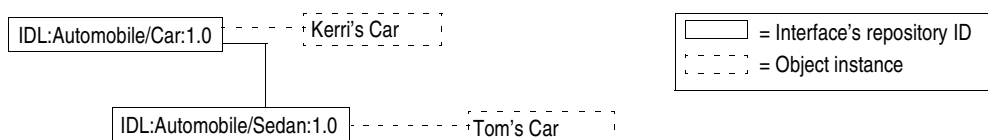
The Location Service Agent is a collection of methods that enable you to discover objects on a network of Smart Agents. You can query based on the interface's repository ID, or based on a combination of the interface's repository ID and the instance name. Results of a query can be returned as either *object references* or more complete *instance descriptions*. An object reference is simply a handle to a specific instance of the object located by a Smart Agent. Instance descriptions contain the object reference, as well as the instance's interface name, instance name, host name and port number, and information about its state (for example, whether it is running or can be activated).

Note The `locserv` executable no longer exists since the service is now part of the core ORB.

Figure 17.2 illustrates the use of interface repository IDs and instance names given the following example IDL:

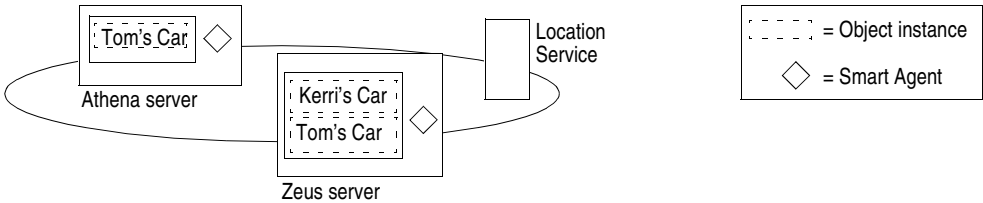
```
module Automobile {
    interface Car{...};
    interface Sedan:Car {...};
}
```

Figure 17.2 Use of interface repository IDs and instance names



Given the example in Figure 17.2, the following diagram visually depicts Smart Agents on a network with references to instances of Car. In this example, there are three instances: one instance of Kerri's Car and two replicas of Tom's Car.

Figure 17.3 Smart Agents on a network with instances of an interface



The following sections explain how the methods provided by the `Agent` class can be used to query VisiBroker Smart Agents for information. Each of the query methods can raise the `Fail` exception, which provides a reason for the failure.

Obtaining names of all hosts running Smart Agents

Using the `String[] all_agent_locations()` method, you can find out which servers are hosting VisiBroker Smart Agents. In the example shown in Figure 17.3, this method would return the names of two servers: Athena and Zeus.

Finding all accessible interfaces

You can query the VisiBroker Smart Agents on a network to find out about all accessible interfaces. To do so, you can use the `String[] all_repository_ids()` method. In the example shown in Figure 17.3, this method would return the repository IDs of two interfaces: Car and Sedan.

Note Earlier versions of the VisiBroker ORB used IDL interface names to identify interfaces, but the Location Service uses the repository id instead. To illustrate the difference, if an interface name is `::module1::module2::interface`, the equivalent repository id is `IDL:module1/module2/interface:1.0`. For the example shown in Figure 17.2, the repository ID for Car would be `IDL:Automobile/Car:1.0`, and the repository ID for Sedan would be `IDL:Automobile/Sedan:1.0`.

Obtaining references to instances of an interface

You can query VisiBroker Smart Agents on a network to find all available instances of a particular interface. When performing the query, you can use either of these methods:

Table 17.1 Obtaining references to objects that implement a given interface

Method	Description
<code>Object[] all_instances(String repository_id)</code>	Use this method to return object references to instances of the interface.
<code>Desc[] all_instance_descs(String repository_id)</code>	Use this method to return an instance description for instances of the interface.

In the example shown in Figure 17.3, a call to either method with the request `IDL:Automobile/Car:1.0` would return three instances of the Car interface: Tom's Car on Athena, Tom's Car on Zeus, and Kerri's Car. The Tom's Car instance is returned twice because there are occurrences of it with two different Smart Agents.

Obtaining references to like-named instances of an interface

Using one of the following methods, you can query VisiBroker Smart Agents on a network to return all occurrences of a particular instance name.

Table 17.2 References to like-named instances of an interface

Method	Description
<code>Object[] all_replica(String repository_id, String instance_name)</code>	Use this method to return object references to like-named instances of the interface.
<code>Desc[] all_replica_descs(String repository_id, String instance_name)</code>	Use this method to return an instance description for like-named instances of the interface.

In the example shown in Figure 17.3 on page 17-4, a call to either method specifying the repository ID `IDL:Automobile/Sedan:1.0` and instance name `Tom's Car` would return two instances because there are occurrences of it with two different Smart Agents.

What is a trigger?

A trigger is essentially a callback mechanism that lets you determine changes to the availability of a specified instance. It is an asynchronous alternative to polling an Agent, and is typically used to recover after the connection to an object has been lost. Whereas queries can be employed in many ways, triggers are special-purpose.

Looking at trigger methods

The trigger methods in the `Agent` class are described in the following table:

Table 17.3 Trigger methods

Methods	Description
<code>void reg_trigger(com.inprise.vbroker.ObjLocation.TriggerDesc desc, com.inprise.vbroker.ObjLocation.TriggerHandler handler)</code>	Use this method to register a trigger handler.
<code>void unreg_trigger(com.inprise.vbroker.ObjLocation.TriggerDesc desc, com.inprise.vbroker.ObjLocation.TriggerHandler handler)</code>	Use this method to unregister a trigger handler.

Both of the `Agent` trigger methods can raise the `Fail` exception, which provides a reason for the failure.

The `TriggerHandler` interface consists of the methods described in the following table:

Table 17.4 `TriggerHandler` interface method

Method	Description
<code>void impl_is_ready(com.inprise.vbroker.ObjLocation.TriggerDesc desc)</code>	This method is called by the Location Service when an instance matching the <code>desc</code> becomes accessible.
<code>void impl_is_down(com.inprise.vbroker.ObjLocation.TriggerDesc desc)</code>	This method is called by the Location Service when an instance becomes unavailable.

Creating triggers

A `TriggerHandler` is a callback object. You implement a `TriggerHandler` by deriving from the `TriggerHandlerPOA` class (or the `TriggerHandlerImpl` class with BOA), and implementing its `impl_is_ready()` and `impl_is_down()` methods. To register a trigger with the Location Service, you use the `reg_trigger()` method in the `Agent` interface. This method requires that you provide a description of the instance you want to monitor, and the `TriggerHandler` object you want invoked when the availability of the instance changes. The instance description (`TriggerDesc`) can contain combinations of the following instance information: repository ID, instance name, and host name. The more instance information you provide, the more particular your specification of the instance.

IDL sample 17.2 IDL for `TriggerDesc`

```
struct TriggerDesc {  
    string repository_id;  
    string instance_name;  
    string host_name;  
};
```

Note If a field in the `TriggerDesc` is set to the empty string (""), it is ignored. The default for each field value is the empty string.

For example, a `TriggerDesc` containing only a repository ID matches any instance of the interface. Looking back to our example in Figure 17.3 on page 17-4, a trigger for any instance of `IDL:Automobile/Car:1.0` would occur when one of the following instances becomes available or unavailable: Tom’s Car on Athena, Tom’s Car on Zeus, or Kerri’s Car. Adding an instance name of “Tom’s Car” to the `TriggerDesc` tightens the specification so that the trigger only occurs when the availability of one of the two “Tom’s Car” instances changes. Finally, adding a host name of Athena refines the trigger further so that it only occurs when the instance Tom’s Car on the Athena server becomes available or unavailable.

Looking at only the first instance found by a trigger

Triggers are “sticky.” A `TriggerHandler` is invoked every time an object satisfying the trigger description becomes accessible. You may only be interested in learning when the first instance becomes accessible. If this is the case, invoke the `Agent’s unreg_trigger()` method to unregister the trigger after the first occurrence is found.

Querying an agent

This section contains two examples of using the Location Service to find instances of an interface. The first example uses the `Account` interface shown in the following IDL excerpt:

IDL sample 17.3 Account example interface definition

```
// Bank.idl
module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open (in string name);
    };
};
```

Finding all instances of an interface

The following code sample uses the `all_instances()` method to locate all instances of the `Account` interface. Notice that the Smart Agents are queried by passing “`LocationService`” to the `ORB.resolve_initial_references()` method, then narrowing the object returned by that method to an `ObjLocation.Agent`. Notice, as well, the format of the `Account` repository id—`IDL:Bank/Account:1.0`.

Code sample 17.1 Finding all instances satisfying the `AccountManager` interface

```
// AccountFinder.java
public class AccountFinder {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            com.inprise.vbroker.ObjLocation.Agent the_agent = null;
            try {
                the_agent =
com.inprise.vbroker.ObjLocation.AgentHelper.narrow(orb.resolve_initial_references("LocationService"));
            } catch (org.omg.CORBA.ORBPackage.InvalidName e) {
                System.out.println("Not able to resolve references " +
                    "for LocationService");
                System.exit(1);
            } catch (Exception e) {
                System.out.println("Unable to locate LocationService!");
                System.out.println("Caught exception: " + e);
                System.exit(1);
            }
        }
        org.omg.CORBA.Object[] accountRefs =
            the_agent.all_instances("IDL:Bank/AccountManager:1.0");
        System.out.println("Agent returned " + accountRefs.length +
            " object references");
        for (int i=0; i < accountRefs.length; i++) {
```

```

        System.out.println("Stringified IOR for account #" + (i+1) + ":");
        System.out.println(orb.object_to_string(accountRefs[i]));
        System.out.println();
    }
} catch (Exception e) {
    System.out.println("Caught exception: " + e);
    System.exit(1);
}
}
}

```

Finding everything known to Smart Agents

The following code sample shows how to find everything known to Smart Agents. It does this by invoking the `all_repository_ids()` method to obtain all known interfaces. Then it invokes the `all_instances_descs()` method for each interface to obtain the instance descriptions.

Code sample 17.2 Finding everything known to an osagent

```

// Find.java
public class Find {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            com.inprise.vbroker.ObjLocation.Agent agent = null;
            try {
                agent =
                    com.inprise.vbroker.ObjLocation.AgentHelper.narrow(orb.resolve_initial_references("LocationService"));
            } catch (org.omg.CORBA.ORBPackage.InvalidName e) {
                System.out.println("Not able to resolve references " + "for LocationService");
                System.exit(1);
            } catch (Exception e) {
                System.out.println("Not able to resolve references " + "for LocationService");
                System.out.println("Caught exception: " + e);
                System.exit(1);
            }
            boolean done=false;
            java.io.BufferedReader in =
                new java.io.BufferedReader(new java.io.InputStreamReader(System.in));
            while (! done) {
                System.out.print("-> ");
                System.out.flush();
                String line = in.readLine();
                if(line.startsWith("agents")) {
                    java.lang.String[] agentList = agent.all_agent_locations();
                    System.out.println("Located " + agentList.length + " agents");
                    for (int i=0; i < agentList.length; i++) {
                        System.out.println("\t" + "Agent #" + (i+1) + ": " + agentList[i]);
                    }
                } else if(line.startsWith("rep")) {
                    java.lang.String[] repIds = agent.all_repository_ids();

```



```

        System.out.println("Located " + repIds.length + " repository Ids");
        for (int i=0; i < repIds.length; i++) {
            System.out.println("\t" + "Repository Id #" + (i+1) + ": " + repIds[i]);
        }
    } else if(line.startsWith("objects ")) {
        String names = line.substring("objects ".length(), line.length());
        PrintObjects(names,agent,orb);
    } else if(line.startsWith("quit")) {
        done = true;
    } else {
        System.out.println("Commands: agents\n" +
            "        repository_ids\n" +
            "        objects      <rep Id>\n" +
            "        objects      <rep Id> <obj name>\n" +
            "        quit\n");
    }
}
} catch (com.inprise.vbroker.ObjLocation.Fail err) {
    System.out.println("Location call failed with reason " + err.reason);
} catch (java.lang.Exception err) {
    System.out.println("Caught error " + err);
    err.printStackTrace();
}
}

public static void PrintObjects(String names,
    com.inprise.vbroker.ObjLocation.Agent agent,
    org.omg.CORBA.ORB orb)
    throws com.inprise.vbroker.ObjLocation.Fail {
    int space_pos = names.indexOf(' ');
    String repository_id;
    String object_name;
    if (space_pos == -1) {
        repository_id = names;
        object_name = null;
    } else {
        repository_id = names.substring(0,names.indexOf(' '));
        object_name = names.substring(names.indexOf(' ')+1);
    }
    org.omg.CORBA.Object[] objects;
    com.inprise.vbroker.ObjLocation.Desc[] descriptors;
    if (object_name == null) {
        objects = agent.all_instances(repository_id);
        descriptors = agent.all_instances_descs(repository_id);
    } else {
        objects = agent.all_replica(repository_id,object_name);
        descriptors = agent.all_replica_descs(repository_id,object_name);
    }
    System.out.println("Returned " + objects.length + " objects");
    for (int i=0; i<objects.length; i++) {
        System.out.println("\n\nObject #" + (i+1) + ":");
        System.out.println("=====");
        System.out.println("\tRep ID: " +
            ((com.inprise.vbroker.CORBA.Object)objects[i])._repository_id());
        System.out.println("\tInstance: " +

```

```
        ((com.inprise.vbroker.CORBA.Object)objects[i])._object_name());
    System.out.println("\tIOR: " + orb.object_to_string(objects[i]));
    System.out.println();
    System.out.println("Descriptor #" + (i+1));
    System.out.println("=====");
    System.out.println("Host:           " + descriptors[i].iiop_locator.host);
    System.out.println("Port:          " + descriptors[i].iiop_locator.port);
    System.out.println("Agent Host:    " + descriptors[i].agent_hostname);
    System.out.println("Repository Id: " + descriptors[i].repository_id);
    System.out.println("Instance:      " + descriptors[i].instance_name);
    System.out.println("Activable:     " + descriptors[i].activable);
    }
}
}
```

Writing and registering a trigger handler

The following section illustrates how a trigger is implemented and registered.

Implementing and registering a trigger handler

The following code sample implements and registers a `TriggerHandler`. The `TriggerHandlerImpl`'s `impl_is_ready()` and `impl_is_down()` methods display the description of the instance that caused the trigger to be invoked, and optionally unregister itself. If it is unregistered, the method calls `System.exit()` to terminate the program.

Notice that the `TriggerHandlerImpl` class keeps a copy of the `desc` and `Agent` parameters with which it was created. The `unreg_trigger()` method requires the `desc` parameter. The `Agent` parameter is duplicated in case the reference from the main program is released.

Code sample 17.3 Implementing a trigger handler

```
// AccountTrigger.java

import java.io.*;
import org.omg.PortableServer.*;

class TriggerHandlerImpl extends
    com.inprise.vbroker.ObjLocation.TriggerHandlerPOA {
    public TriggerHandlerImpl(com.inprise.vbroker.ObjLocation.Agent agent,
        com.inprise.vbroker.ObjLocation.TriggerDesc initial_desc) {
        agent = agent;
        initial_desc = initial_desc;
    }

    public void impl_is_ready(com.inprise.vbroker.ObjLocation.Desc desc) {
        notification(desc, true);
    }

    public void impl_is_down(com.inprise.vbroker.ObjLocation.Desc desc) {
```

```

        notification(desc, false);
    }

private void notification(com.inprise.vbroker.ObjLocation.Desc desc, boolean isReady) {
    if (isReady) {
        System.out.println("Implementation is ready:");
    } else {
        System.out.println("Implementation is down:");
    }
    System.out.println("\tRepository Id = " + desc.repository_id + "\n" +
        "\tInstance Name = " + desc.instance_name + "\n" +
        "\tHost Name      = " + desc.iiop_locator.host + "\n" +
        "\tBOA Port         = " + desc.iiop_locator.port + "\n" +
        "\tActivable        = " + desc.activable + "\n" + "\n");
    System.out.println("Unregister this handler and exit (yes/no)?");
    try {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line = in.readLine();
        if (line.startsWith("y") || line.startsWith("Y")) {
            try {
                agent.unreg_trigger(_initial_desc, _this());
            } catch (com.inprise.vbroker.ObjLocation.Fail e) {
                System.out.println("Failed to unregister trigger with reason=[" +
                    e.reason + "]);
            }
            System.out.println("exiting...");
            System.exit(0);
        }
    } catch (java.io.IOException e) {
        System.out.println("Unexpected exception caught: " + e);
        System.exit(1);
    }
}

private com.inprise.vbroker.ObjLocation.Agent _agent;
private com.inprise.vbroker.ObjLocation.TriggerDesc _initial_desc;
}

public class AccountTrigger {

    public static void main(String args[]) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            POA rootPoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootPoa.the_POAManager().activate();
            com.inprise.vbroker.ObjLocation.Agent the_agent =
                com.inprise.vbroker.ObjLocation.AgentHelper.narrow(
                    orb.resolve_initial_references("LocationService"));
            // Create a trigger description and an appropriate TriggerHandler.
            // The TriggerHandler will be invoked when the osagent becomes
            // aware of any new implementations of the interface "Bank::AccountManger"
            com.inprise.vbroker.ObjLocation.TriggerDesc desc =
                new com.inprise.vbroker.ObjLocation.TriggerDesc(

```

Writing and registering a trigger handler

```
                                "IDL:Bank/AccountManager:1.0", "", "");
    TriggerHandlerImpl trig = new TriggerHandlerImpl(the_agent, desc);
    rootPoa.activate_object(trig);
    the_agent.reg_trigger(desc, trig._this());
    orb.run();
} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
}
```

Using the Naming Service

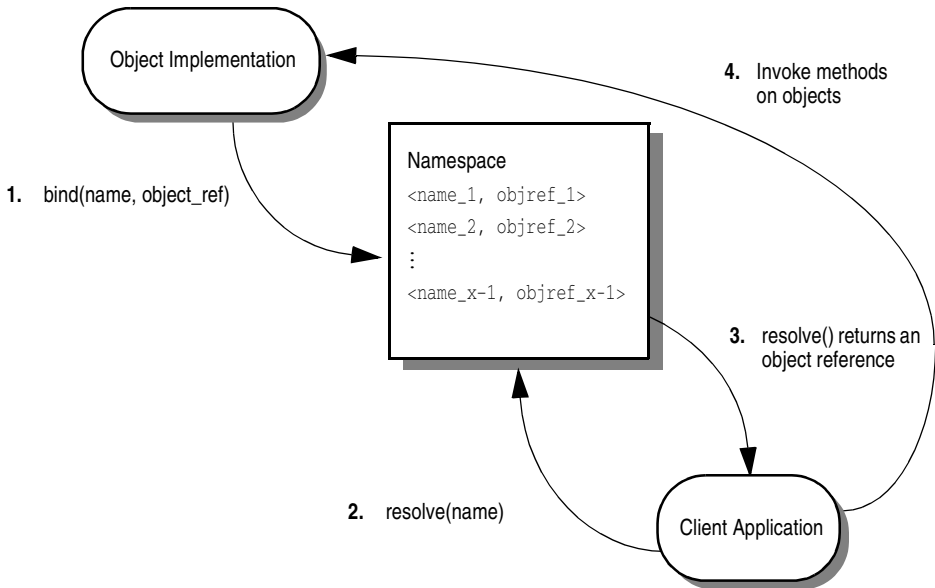
This chapter describes how to use the VisiBroker Naming Service which is a complete implementation of the *Interoperable Naming Specification* document (orbos/98-10-11) from the OMG.

Overview

The Naming Service allows you to associate one or more *logical* names with an object reference and store those names in a *namespace*. It also allows your client applications to use the Naming Service to obtain an object reference by using the logical name assigned to that object.

Figure 18.1 contains a simplified view of the Naming Service that shows how

- 1 An object implementation can *bind* a name to one of its objects within a namespace.
- 2 Client applications can then use the same namespace to *resolve* a name which returns an object reference to a naming context or an object.

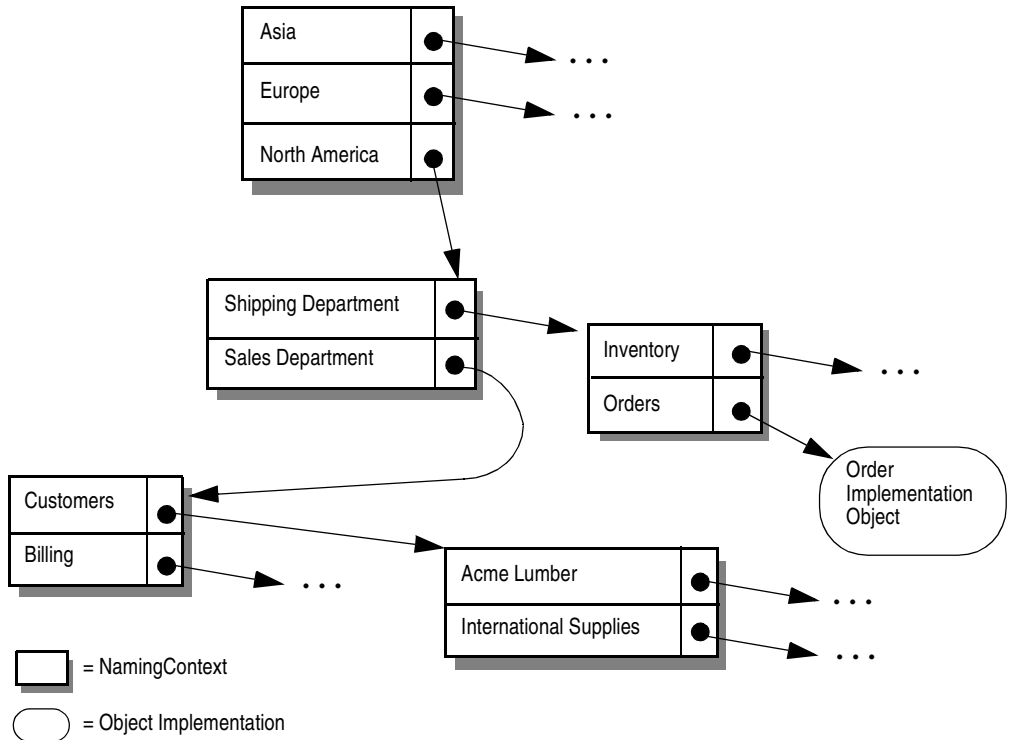
Figure 18.1 Binding, resolving, and using an object name from a naming context within a namespace

There are some important differences to consider between locating an object implementation with the VisiBroker Naming Service as opposed to the Smart Agent.

- Smart Agent uses a flat namespace, while the Naming Service uses a hierarchical one.
- Object's interface name is defined at the time you compile your client and server applications. Changing an interface name requires that you recompile your applications. In contrast, the Naming Service allows object implementations to bind logical names to its objects at runtime.
- Object may implement only one interface name, but the Naming Service allows you to bind more than one logical name to a single object.

Understanding the namespace

Figure 18.2 shows how the Naming Service might be used to name objects that make up an order entry system. This hypothetical order entry system organizes its namespace by geographic region, then by department, and so on. The Naming Service allows you to organize the namespace in a hierarchical structure of `NamingContext` objects that can be traversed to locate a particular name. For example, the logical name `NorthAmerica/ShippingDepartment/Orders` could be used to locate an `Order` object.

Figure 18.2 Naming scheme for an order entry system

Naming contexts

To implement the namespace shown in Figure 18.2 with the VisiBroker Naming Service, each of the shadowed boxes would be implemented by a `NamingContext` object. A `NamingContext` object contains a list of Name structures that have been bound to object implementations or to other `NamingContext` objects. Though a logical name may be bound to a `NamingContext`, it is important to realize that a `NamingContext` does not, by default, have a logical name associated with it nor is such a name required.

Object implementations use a `NamingContext` object to *bind* a name to an object that they offer. Client applications use a `NamingContext` to *resolve* a bound name to an object reference.

A `NamingContextExt` interface is also available which provides methods necessary for using stringified names.

Naming context factories

A naming context factory provides the interface for *bootstrapping* the Naming Service. It has operations for shutting down a Naming Service and creating new contexts when there are none. Factories also have an additional API that returns the root context. The root context provides a very critical role as a reference point. This is the common starting point to store all data that are supposed to be publicly available.

Two classes are provided with the VisiBroker Naming Service that allow you to create a namespace; the default naming context factory and the extended naming context factory. The default naming context factory creates an empty namespace that has no root `NamingContext`. You may find it more convenient to use the extended naming context factory because it creates a namespace with a root `NamingContext`.

You must obtain at least one of these `NamingContext` objects before your object implementations can bind names to their objects and before client applications can resolve a name to an object reference.

Each of the `NamingContext` objects shown in Figure 18.2 on page 18-3 could be implemented within a single *name service* process, or they could be implemented within as many as five distinct name server processes.

Names and NameComponent

A `CosNaming::Name` represents an identifier that can be bound to an object implementation or a `CosNaming::NamingContext`. A `Name` is not simply a string of alphanumeric characters; it is a sequence of one or more `NameComponent` structures.

Each `NameComponent` contains two attribute strings, `id` and `kind`. The naming service does not interpret or manage these strings, except to ensure that each `id` and `kind` is unique within a given `NamingContext`.

The `id` and `kind` attributes are strings which uniquely identify the object to which the name is bound. The `kind` member adds a descriptive quality to the name. For example, the name “Inventory.RDBMS” has an `id` member of “Inventory” and a `kind` member of “RDBMS.”

IDL sample 18.1 IDL Specification for the NameComponent structure

```
module CosNaming
{
    typedef string Istring;
    struct NameComponent {
        Istring id;
        Istring kind;
    };
    typedef sequence<NameComponent> Name;
};
```

The `id` and `kind` attributes of a `NameComponent` must be a character from the ISO 8859-1 (Latin-1) character set, excluding the null character (0x00) and other non-printable characters. Neither of the strings in a `NameComponent` can exceed 255 characters. Furthermore, the Naming Service does not support `NameComponent` which uses wide strings.

Note The `id` attribute of a `Name` cannot be an empty string, but the `kind` attribute can be an empty string.

Name resolution

Your client applications use the `NamingContext` method `resolve` to obtain an object reference, given a logical `Name`. Because a `Name` consists of one or more `NameComponent` objects, the resolution process requires that all of the `NameComponent` structures that make up the `Name` be traversed.

Stringified names

Because the representation of `CosNaming::Name` is not in a form that is readable or convenient for exchange, a stringified name has been defined to resolve this problem. A stringified name is a one-to-one mapping between a string and a `CosNaming::Name`. If two `CosNaming::Name` objects are equal, then their stringified representations are equal and vice versa. In a stringified name, a forward slash (/) serves as a name component separator; a period (.) serves as the `id` and `kind` attributes separator; and a backslash (\) serves as an escape character. By convention a `NameComponent` with an empty `kind` attribute does not use a period (for example, `Order`).

Code sample 18.1 Stringified name example

```
"Inprise.Company/Engineering.Department/Printer.Resource"
```

Note In the following examples, `NameComponent` structures are given in their stringified representations.

Simple and complex names

A *simple name*, such as `Billing`, has only a single `NameComponent` and is always resolved relative to the target naming context. A simple name may be bound to an object implementation or to a `NamingContext`.

A *complex name*, such as `NorthAmerican/ShippingDepartment/Inventory`, consists of a sequence of three `NameComponent` structures. If a complex name consisting of n `NameComponent` objects has been bound to an object implementation, then the first $(n-1)$ `NameComponent` objects in the sequence must each resolve to a `NamingContext`, and the last `NameComponent` object must resolve to an object implementation.

If a `Name` is bound to a `NamingContext`, each `NameComponent` structure in the sequence must refer to a `NamingContext`.

Code sample 18.2 shows a complex name, consisting of three components and bound to a CORBA object. This name corresponds to the stringified name, `NorthAmerica/SalesDepartment/Order`. When resolved within the topmost naming context, the first two components of this complex name resolve to `NamingContext` objects, while the last component resolves to an object implementation with the logical name "Order."

Code sample 18.2 Example of a complex name bound to an ORB object

```

. . .
// Name stringifies to "NorthAmerica/SalesDepartment/Order"
NameComponent[] continentName = { new NameComponent("NorthAmerica", "") };
NamingContext continentContext = rootNamingContext.bind_new_context(continentName);
NameComponent[] departmentName = { new NameComponent("SalesDepartment", "") };
NamingContext departmentContext = continentContext.bind_new_context(departmentName);
NameComponent[] objectName = { new NameComponent("Order", "") };
departmentContext.rebind(objectName, myPOA.servant_to_reference(managerServant));
. . .

```

Running the Naming Service

The Naming Service can be started with the following commands. Once you have started the naming service, you may browse its contents by using the VisiBroker Console. For more details see, “Naming Services” on page 11-6.

Installing the Naming Service

The Naming Service is installed automatically when you install VisiBroker for Java 4.5. It consists of a file `nameserv`, which for Windows NT is a binary executable and for UNIX is a script, and Java class files which are stored in the `vbjorb.jar` file.

Configuring the Naming Service

In previous versions of VisiBroker, the Naming Service maintained persistence by logging any modifying operations to a flat-file. From version 4.0 on, the Naming Service works in conjunction with backing store adaptors. It is important to note that not all backing store adaptors support persistence. The default `InMemory` adaptor is non-persistent while all the other adaptors are. For more details about adaptors, see “Pluggable backing store” on page 18-14.

Note A Naming Server needs to register itself with the Smart Agent when it is starting up. Therefore, you need to run the Smart Agent to bootstrap the Naming Service. This allows clients to retrieve the initial root context by calling the `resolve_initial_references` method. The resolving function works through the Smart Agent for the retrieval of the required references. Similarly, Naming Servers that participate in a federation also uses the same mechanism for setting up a federation.

Starting the Naming Service

You can start the Naming Service by using the `nameserv` launcher program in the `bin` directory. The `nameserv` launcher uses the `com.inprise.vbroker.naming.ExtFactory` factory class by default.

UNIX `nameserv [driver_options] [nameserv_options] root_context_name &`

Windows

```
start nameserv [driver_options] [nameserv_options] root_context_name
```

Option	Description
driver_options	(Must appear before the factory name)
-J<Java option>	Pass the specified option directly to the JVM.
-VBJversion	Print the version number of the VBJ.
-VBJdebug	Print debugging information for the VBJ.
nameserv_options	
-, -h, -help, -usage	Print out the usage information.
-config=<properties_file>	Use <properties_file> as the configuration file when starting up the Naming Service
<ns_name>	The name to use for this Naming Service. This is optional; the default name is <code>NameService</code> .

Starting the Naming Service with vbj

The Naming Service may still be started using `vbj`.

```
prompt>vbj com.inprise.vbroker.naming.ExtFactory <ns_name>
```

Invoking the Naming Service from the Command Line

The Naming Service Utility (`nsutil`) provides the ability to store and retrieve bindings from the command line.

Configuring nsutil

To use `nsutil`, first configure the naming service instance using either:

```
prompt>nameserve <factory_name>
```

or

```
prompt>nsutil -VBJprop <ns_config> <cmd> [args]
```

Option	Description
ns_config	Defines the factory name
SVCnameroot=<factory_name>	Note: Before using <code>SVCnameroot</code> , you must first run <code>OSAgent</code> .
ORBInitRef=NameService=<url>	File name or URL, prefixed by its type, which may be (<code>corbaloc:</code> , <code>corbaname:</code> , <code>file:</code> , <code>ftp:</code> , <code>http:</code> , or <code>ior:</code>). So, for example to assign a file in a local directory, the <code>ns_config</code> string would be: -VBJprop ORBInitRef=NameService=<file:ns.ior>
cmd	Any CosNaming operation, and, in addition, ping and shutdown.

Running nsutil

The Naming Service Utility supports all the CosNaming operations as well as two additional commands. The CosNaming operations supported are:

cmd	Parameter(s)
bind	name objRef
bind_context	name objRef
bind_new_context	name ctxRef
destroy	name
list	name*
new_context	
rebind	name objRef
rebind_context	name ctxRef
resolve	name
unbind	name

The additional nsutil commands are:

cmd	Parameter	Description
ping	name	Resolves the stringified name and contacts the object to see if it is still alive.
shutdown	factory_name	Shuts the Naming Service down gracefully from the command line. The factory_name is the name specified when the Naming Service was started. Note: The initial context need not have been set for this command to be invoked.

To run an operation from the nsutil command, place the operation name and its parameters as the <cmd> parameter. For example:

```
prompt>nsutil -VBJprop ORBInitRef=NameService=file://ns.iior resolve myName
```

Closing nsutil

To close nsutil, use the shutdown command:

```
prompt>nsutil -VBJprop ORBInitRef=NameService=file://ns.iior shutdown
```

Bootstrapping a Naming Service

There are three ways to start a client application so that it can obtain an initial object reference to a specified Naming Service. You can use the following three command-line options when starting a Naming Service:

- ORBInitRef
- ORBDefaultInitRef
- SVCnameroot

Calling `resolve_initial_references`

The new Naming Service provides a simple mechanism by which the `resolve_initial_references` method can be configured to return a common naming context. You use the `resolve_initial_references` method which returns the root context of the Naming Server to which the client program connects. Three simple examples will illustrate how to use these three options. Suppose there are three VisiBroker Naming Services running on the host `TestHost`: `ns1`, `ns2`, and `ns3`. And there are three server applications: `sr1`, `sr2`, `sr3`, each running on a different port (20001, 20002, and 20003) on the host `TestHost`. Server `sr1` binds itself in `ns1`, `sr2` in `ns2`, and `sr3` in `ns3`.

Code sample 18.3 Code snippet showing how to obtain the root naming context

```

. . .
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
org.omg.CORBA.Object rootObj = orb.resolve_initial_references("NameService");
. . .

```

Using `-DSVCnameroot`

You use the `-DSVCnameroot` option to specify which VisiBroker Naming Service instance (especially important if several unrelated naming service instances are running) you want to bootstrap. For instance, if you want to bootstrap into `ns1`, you would start your client program as:

```
vbj -DSVCnameroot=ns1 <client_application>
```

You can then obtain the root context of `ns1` by calling the `resolve_initial_references` method on an ORB reference inside your client application as illustrated in Code sample 18.3.

Using `-DORBInitRef`

You can use either the `corbaloc` or `corbaname` URL naming schemes to specify which VisiBroker Naming Service you want to bootstrap.

Using a corbaloc URL

If you want to bootstrap using Naming Service `ns2`, then you should start your client application as follows:

```
vbj -DORBInitRef NameService=corbaloc::TestHost:20002/NameService <client_application>
```

You can then obtain the root context of `ns2` by calling the `resolve_initial_references` method on an ORB reference inside your client application as illustrated in Code sample 18.3 on page 18-9.

Note This example will work only if there is a server running at port 20002 that is bound to the Naming Service you want to access.

Note The `iiploc` and `iiopname` URL schemes are implemented by `corbaloc` and `corbaname`, respectively. For backwards compatibility, the old schemes are still supported.

Using a corbaname URL

If you want to bootstrap into `ns3` by using `corbaname`, then you should start your client program as:

```
vbj -DORBInitRef NameService=corbaname::TestHost:20003/ <client_application>
```

You can then obtain the root context of `ns3` by calling the `resolve_initial_references` method on an ORB reference inside your client application as illustrated in Code sample 18.3 on page 18-9.

-DORBDefaultInitRef

You can use either a `corbaloc` or `corbaname` URL to specify which VisiBroker Naming Service you want to bootstrap.

Using -DORBDefaultInitRef with a corbaloc URL

If you want to bootstrap into `ns2`, then you should start your client program as:

```
vbj -DORBDefaultInitRef corbaloc::TestHost:20002 <client_application>
```

You can then obtain the root context of `ns2` by calling the `resolve_initial_references` method on an ORB reference inside your client application as illustrated in Code sample 18.3 on page 18-9.

Using -DORBDefaultInitRef with corbaname

The combination of `-DORBDefaultInitRef` and `corbaname` works differently from what is expected. If `-DORBDefaultInitRef` is specified, a slash and the stringified object key is always appended to the `corbaname`. For example, if the URL `corbaname::TestHost:20002`, then by specifying `-DORBDefaultInitRef`, `resolve_initial_references` will result in a new URL: `corbaname::TestHost:20003/NameService`.

NamingContext

This object is used to contain and manipulate a list of names that are bound to ORB objects or to other `NamingContext` objects. Client applications use this interface to resolve or list all of the names within that context. Object implementations use this object to bind names to object implementations or to bind a name to a `NamingContext` object. IDL sample 18.2 shows the IDL specification for the `NamingContext`.

IDL sample 18.2 Specification for the NamingContext interface

```
module CosNaming {
    interface NamingContext {
        void bind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind(in Name n, in Object obj)
            raises(NotFound, CannotProceed, InvalidName);
        void bind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind_context(in Name n, in NamingContext nc)
            raises(NotFound, CannotProceed, InvalidName);
        Object resolve(in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        void unbind(in Name n)
            raises(NotFound, CannotProceed, InvalidName);
        NamingContext new_context();
        NamingContext bind_new_context(in Name n)
            raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void destroy()
            raises(NotEmpty);
        void list(in unsigned long how_many,
            out BindingList bl,
            out BindingIterator bi);
    };
};
```

NamingContextExt

The `NamingContextExt` interface, which extends `NamingContext`, provides the operations required to use stringified names and URLs.

IDL sample 18.3 Specification for the NamingContextExt interface

```
module CosNaming {
    interface NamingContextExt : NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName to_string(in Name n)
            raises(InvalidName);
        Name to_name(in StringName sn)
            raises(InvalidName);

        exception InvalidAddress {};
        URLString to_url(in Address addr, in StringName sn)
            raises(InvalidAddress, InvalidName);
        Object resolve_str(in StringName n)
            raises(NotFound, CannotProceed, InvalidName);
    };
};
```

Default naming contexts

A client application can specify a *default naming context*, which is the naming context that the application will consider to be its *root* context. Note that the default naming context is the *root* only in relation to this client application and, in fact, it may be contained by another context.

Obtaining the default naming context

Java client applications can connect to the Naming Service by using the `resolve_initial_references` method in the ORB interface. To use this feature, the `SVCnameroot` parameters must be specified when the client is started.

For example, to start a Java application named `ClientApplication` that intends to use the naming context `Inventory` as its default naming context, you could enter the following command:

```
prompt> vbj -DSVCnameroot=NorthAmerica/ShippingDepartment/Inventory \
    ClientApplication
```

In the example, `NorthAmerica` is the server name and `ShippingDepartment/Inventory` is the stringified name from the root context.

Note When using the `vbj` command, all `-D` properties must appear before the Java class name.

Naming Service Properties

The naming service properties are:

Table 18.1 Naming service properties

Property	Default	Description
<code>vbroker.naming.adminPwd</code>	<code>inprise</code>	Password required by administrative Visibroker naming service operations.
<code>vbroker.naming.enableClusterFailOver</code>	<code>true</code>	When set to <code>true</code> , specifies that an interceptor be installed that handles fail-over for objects that were retrieved from the Naming Service. In case of an object failure, an attempt is made to transparently reconnect to another object from the same cluster as the original.
<code>vbroker.naming.enableSlave</code>	<code>0</code>	If <code>true</code> , enables master/slave naming services configuration. See “Failover” on page 18-23 for information about configuring master/slave naming services.
<code>vbroker.naming.iorFile</code>	<code>ns.ior</code>	Specifies the full path name for storing the naming service IOR. If you do not set this property, the naming service will try to output its IOR into a file named <code>ns.ior</code> in the current directory. The naming service silently ignores file access permission exceptions when it tries to output its IOR.
<code>vbroker.naming.LogLevel</code>	<code>emerg</code>	Specifies the level of log messages to be output from naming service.
<code>vbroker.naming.propBindOn</code>	<code>0</code>	If <code>true</code> , the implicit clustering feature is turned on.
<code>vbroker.naming.smrr.pruneStaleRef</code>	<code>1</code>	This property is relevant when the name service cluster uses the smart round robin criterion. When this property is set to <code>1</code> , a stale object reference that was previously bound to a cluster with the smart round robin criterion will be removed from the bindings when the name service discovers it. If this property is set to <code>0</code> , stale object reference bindings under the cluster are not eliminated. However, a cluster with smart round robin criterion will always return an active object reference upon a <code>resolve()</code> or <code>select()</code> call if such an object binding exists, regardless of the value of the <code>vbroker.naming.smrr.pruneStaleRef</code> property. By default, the implicit clustering in the 4.5 name service uses the smart round robin criterion with the property value set to <code>1</code> .

Pluggable backing store

The previous version of the Naming Service kept its namespace (that is, the set of naming contexts and object-name bindings) in memory. However, it logged all modifiable operations from its namespace into a logging file. This flat file could then be used when starting up the naming service to recreate the previous namespace.

The current Naming Service maintains its namespace by using a pluggable backing store. Whether or not the namespace is persistent, depends on how you configure the backing store: to use JDBC adaptor, the Java Naming and Directory Interface (JNDI, which is certified for LDAP), or the default, in-memory adaptor.

Types of backing stores

The types of backing store adaptors supported are:

- In-memory adaptor
- JDBC adaptor for relational databases
- DataExpress adaptor
- JNDI (for LDAP only)

Note For an example using pluggable adaptors, see the code in the `examples/ins/pluggable_adaptors` directory.

In-memory adaptor

The in-memory adaptor keeps the namespace information in memory and is not persistent. This is the adaptor used by the Naming Service by default.

JDBC adaptor

Relational databases are supported via JDBC. The following databases have been certified to work with the Naming Service JDBC adaptor:

- JDataStore
- Oracle
- Sybase
- Microsoft SQLServer
- DB2
- Interbase

DataExpress adaptor

In addition to the JDBC adaptor, there is also a DataExpress adaptor which allows you to access JDataStore databases natively. It is much faster than accessing JDataStore through JDBC, but the DataExpress adaptor has some limitations. It only supports a local database running on the same machine as the Naming Server. To access a remote JDataStore database, you must use the JDBC adaptor.

JNDI adaptor

A JNDI adaptor is also supported. Sun's JNDI (Java naming and directory interface) provides a standard interface to multiple naming and directory services throughout the enterprise. JNDI has a Service Provider Interface (SPI) with which different naming and service vendors must conform. There are different SPI modules available for Netscape LDAP server, Novell NDS, WebLogic Tengah, etc. By supporting JNDI, the VisiBroker Naming Service allows you to have portable access to these naming and directory services and other future SPI providers. However, the JNDI adaptor is only certified for the Netscape LDAP Server 4.0.

Configuration and use

Backing store adaptors are pluggable, which means that the type of adaptor used can be specified by user-defined information stored in a configuration (properties) file used when starting up the Naming Service. All adaptors, except the in-memory one, provide persistence. The in-memory adaptor should be used when, you want to use a lightweight Naming Service which keeps its namespace entirely in memory.

Note For the current version of the Naming Service, you cannot change settings while the Naming Service is running. To change a setting, you must bring down the service, make the change to the configuration file, and then restart the Naming Service.

Properties file

As with the Naming Service in general, which adaptor is to be used and any specific configuration of it is handled in Naming Service properties file. The default properties common to all adaptors are:

Table 18.2 Default properties common to all adaptors

Property	Default	Description
<code>vbroker.naming.backingStoreType</code>	<code>InMemory</code>	Specifies the naming service adaptor type to use. This property specifies which type of backing store you want the Naming Service to use. The valid options are: <code>InMemory</code> , <code>JDBC</code> , <code>Dx</code> , <code>JNDI</code> . The default is <code>InMemory</code> .
<code>vbroker.naming.cacheOn</code>	<code>0</code>	Specifies whether to use the naming service cache.
<code>vbroker.naming.cacheSize</code>	<code>5</code>	Specifies the size of the naming service cache if it's turned on.

JDBC Adaptor properties

`vbroker.naming.backingStoreType`

This property should be set to `JDBC`. The `poolSize`, `jdbcDriver`, `url`, `loginName`, and `loginPwd` properties must also be set for the JDBC adaptor.

`vbroker.naming.jdbcDriver`

This property specifies the JDBC driver that is needed to access the database used as your backing store. The Naming Service loads the appropriate JDBC driver specified. The default is the Java DataStore JDBC driver.

JDBC driver value	Description
<code>com.borland.datastore.jdbc.DataStoreDriver</code>	JDataStore driver
<code>com.sybase.jdbc.SybDriver</code>	Sybase driver
<code>oracle.jdbc.driver.OracleDriver</code>	Oracle driver
<code>interbase.interclient.Driver</code>	Interbase driver
<code>weblogic.jdbc.mssqlserver4.Driver</code>	WebLogic MS SQLServer driver
<code>COM.ibm.db2.jdbc.app.DB2Driver</code>	IBM DB2 driver

`vbroker.naming.loginName`

This property is the login name associated with the database. The default is `VisiNaming`.

`vbroker.naming.loginPwd`

This property is the login password associated with the database. The default value is `VisiNaming`.

`vbroker.naming.poolSize`

This property specifies the number of database connections in your connection pool when using the JDBC Adaptor as our backing store. The default value is 5, but it can be increased to whatever value the database can handle. If you expect many requests will be made to the Naming Service, you should make this value larger.

vbroker.naming.url

This property specifies the location of the database which you want to access. The setting is dependent on the database in use. The default is the `JDataStore` and the database location is the current directory and is called `rootDB.jds`. You can use any name you like not necessarily `rootDB.jds`. The configuration file needs to be updated accordingly.

URL value	Description
<code>jdbc:borland:dslocal:<db_name></code>	JDataStore URL
<code>jdbc:sybase:Tds:<host>:<port>/<db_name></code>	Sybase URL
<code>jdbc:oracle:thin:@<host>:<port>:<sid></code>	Oracle URL
<code>jdbc:interbase://<server>/<full_db_path></code>	Interbase ¹ URL
<code>jdbc:weblogic:mssqlserver4:<db_name>@<host>:<port></code>	WebLogic MS SQLServer URL
<code>jdbc:db2:<db_name></code>	IBM DB2 ² URL
<code><full_path_JDataStore_db></code>	DataExpress ³ URL for the native driver

1. You should start InterServer before accessing InterBase via JDBC. If the InterBase server resides on the local host, specify `<server>` as `localhost`; otherwise specify it as the host name. If the InterBase database resides on Windows NT, specify the `<full_db_path>` as `driver:\\dir1\dir2\db.gdb` (the first backslash `[\\]` is to escape the second backslash `[\\]`). If the InterBase database resides on UNIX, specify the `<full_db_path>` as `\dir1\dir2\db.gdb`. You can get more information from <http://www.interbase.com/>.
2. Before you access DB2 via JDBC, you must register the database by its alias `<db_name>` using the Client Configuration Assistant. After the database has been registered, you do not have to specify `<host>` and `<port>` for the `vbroker.naming.url` property.
3. If the JDataStore database resides on Windows NT, the `<full path of the JDataStore database>` should be `Driver:\\dir1\\dir2\\db.jds` (the first backslash `[\\]` is to escape the second backslash `[\\]`). If the JDataStore database resides on UNIX, the `<full path of the JDataStore database>` should be `/dir1/dir2/db.jds`.

DataExpress Adaptor properties

`vbroker.naming.backingStoreType`

This property should be set to `Dx`.

`vbroker.naming.loginName`

This property is the login name associated with the database. The default is `VisiNaming`.

`vbroker.naming.loginPwd`

This property is the login password associated with the database. The default value is `VisiNaming`.

`vbroker.naming.url`

This property specifies the location of the database.

JNDI adaptor properties

The following is an example of settings that might appear in the configuration file for a JNDI adaptor:

Table 18.3 Example of a JNDI adaptor configuration file

Setting	Description
<code>vbroker.naming.backingStoreType=JNDI</code>	Specifies the backing store type which is JNDI for the JNDI adaptor.
<code>vbroker.naming.loginName=<user name></code>	The user login name on the JNDI backing server.
<code>vbroker.naming.loginPwd=<password></code>	The password for the JNDI backing server user.
<code>vbroker.naming.jndiInitialFactory=com.sun.jndi.ldap.LdapCtxFactory</code>	Specifies the JNDI initial factory.
<code>vbroker.naming.jndiProviderURL=ldap://<hostname>:389/<initial root context></code>	Specifies the JNDI provider URL
<code>vbroker.naming.jndiAuthentication=simple</code>	Specifies the JNDI authentication type supported by the JNDI backing server.

Note The user must have the necessary privilege to add schemas/attributes to the Directory Server.

Caching facility

By turning on the caching facility, you can improve performance of the backing store. For example, in the case of the JDBC adaptor, directly accessing the database every time there is a resolve or bind operation is relatively slow. If you cache the results, you can reduce the number of database accesses. There are a number of caveats to keep in mind before turning on the caching facility. First, make sure that the Naming Service using the cache is the only Naming Service accessing the underlying data, otherwise, clients using the Naming Service may get the wrong data because the cache may contain stale data. You will only see improvement in the performance of the backing store if the same piece of data is accessed more than once.

Note Do not set caching on unless you are absolutely sure it will improve performance in your environment.

The caching facility implements a per-context cache. There will a cache installed in every context, which is used to cache both contexts and objects. The size of this cache is tunable. By default, the size of this cache is 5.

To use the caching facility add the following properties to your configuration file:

```
vbroker.naming.cacheOn=1  
vbroker.naming.CacheSize=5
```

Clusters

VisiBroker supports a clustering feature which allows a number of object bindings to be associated with a single name. The Naming Service can then perform load balancing among the different bindings in a cluster. You may decide on a load-balancing criterion at the time a cluster is created. Clients, which subsequently resolve name-object bindings against a cluster, would be load balanced amongst different cluster server members.

A cluster is a multi-bind mechanism that associates a `Name` with a group of object references. The creation of a cluster is done through a `ClusterManager` reference. At creation time, the `create_cluster` method for the `ClusterManager` takes in a `String` parameter which specifies the criterion to be used. This method returns a reference to a cluster, with which you are able to add, remove and iterate through its members. After deciding on the composition of a cluster, you can bind its reference with a particular name to any context in a Naming Service. By doing so, subsequent resolve operations against the `Name` will return a particular object reference in this cluster.

Clustering criteria

The Naming Service uses a `RoundRobin` criterion with clusters by default. After a cluster has been created, its criterion cannot be changed. User-defined criteria are not supported, but the list of supported criteria will grow as time goes on. Besides the default `RoundRobin` criterion, the only other criterion currently available is `SmartRoundRobin`. The difference between the `SmartRoundRobin` and `RoundRobin` is that `SmartRoundRobin` performs some verifications to ensure that the CORBA object reference is an active one; that the object reference is referring to a CORBA server which is in a ready state.

Note We do not recommend that you use `SmartRoundRobin` as the current implementation will activate any object that is verified to be active. Also, the cluster failover feature is only available with `RoundRobin` criterion.

Cluster and ClusterManager interfaces

Although a cluster is very similar to a naming context, there are certain methods found in a context that are not relevant to a cluster. For example, it would not make sense to bind a naming context to a cluster, because a cluster should contain a set of object references, not naming contexts. However, a cluster interface shares many of the same methods with the `NamingContext` interface, such as `bind`, `rebind`, `resolve`, `unbind` and `list`. This common set of operations mainly pertains to operations on a group. The only cluster-specific operation is `pick`. Another crucial difference between the two is that a cluster does not support compound names. It can only use a single component name, because clusters do not have a hierarchical directory structure, rather it stores its object references in a flat structure.

IDL sample 18.4 IDL Specification for the Cluster interface

```

CosNamingExt module {
    typedef sequence<Cluster> ClusterList;
    enum ClusterNotFoundReason {
        missing_node,
        not_context,
        not_cluster_context
    };
    exception ClusterNotFound {
        ClusterNotFoundReason why;
        CosNaming::Name rest_of_name;
    };
    exception Empty {};
    interface Cluster {
        Object select() raises(Empty);
        void bind(in CosNaming::NameComponent n, in Object obj)
            raises(CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName,
                  CosNaming::NamingContext::AlreadyBound);
        void rebind(in CosNaming::NameComponent n, in Object obj)
            raises(CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
        Object resolve(in CosNaming::NameComponent n)
            raises(CosNaming::NamingContext::NotFound,
                  CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
        void unbind(in CosNaming::NameComponent n)
            raises(CosNaming::NamingContext::NotFound,
                  CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
        void destroy()
            raises(CosNaming::NamingContext::NotEmpty);
        void list(in unsigned long how_many,
                  out CosNaming::BindingList bl,
                  out CosNaming::BindingIterator bi);
    };
};

```

IDL sample 18.5 IDL Specification for the ClusterManager interface

```

CosNamingExt module {
    interface ClusterManager
    {
        Cluster create_cluster(in string algo);
        Cluster find_cluster(in CosNaming::NamingContext ctx, in CosNaming::Name n)
            raises(ClusterNotFound, CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
        Cluster find_cluster_str(in CosNaming::NamingContext ctx, in string n)
            raises(ClusterNotFound, CosNaming::NamingContext::CannotProceed,
                  CosNaming::NamingContext::InvalidName);
        ClusterList clusters();
    };
};

```

Creating a cluster

To create a cluster you use the Cluster Manager interface. A single `ClusterManager` object is automatically created when a Naming Server starts up. There is only one `ClusterManager` per Naming Server. The role of a `ClusterManager` is to create, retrieve and keep track of the clusters that are in the Naming Server.

- 1 Bind to the Naming Server with which you wish to create cluster objects.
- 2 Get a reference to the Cluster Manager by calling `get_cluster_manager` method on the factory reference.
- 3 Create a cluster using a specified cluster criterion.
- 4 Bind objects to an Name using the cluster.
- 5 Bind the `Cluster` object itself to a Name.
- 6 Resolve through the Cluster reference for the specified cluster criterion.

Code sample 18.4 Creating and using a Cluster object

```

. . .
ExtendedNamingContextFactory myFactory =
    ExtendedNamingContextFactoryHelper.bind(orb, "NamingService");
ClusterManager clusterMgr = myFactory.get_cluster_manager();
Cluster clusterObj = clusterMgr.create_cluster("RoundRobin");
clusterObj.bind(new NameComponent("member1", "aCluster"), obj1);
clusterObj.bind(new NameComponent("member2", "aCluster"), obj2);
clusterObj.bind(new NameComponent("member3", "aCluster"), obj3);
NameComponent myClusterName = new NameComponent("ClusterName", "");
root.bind(myClusterName, clusterObj);
root.resolve(myClusterName) // a member of the Cluster is returned.
root.resolve(myClusterName) // the next member of the Cluster is returned.
root.resolve(myClusterName) // the last member of the Cluster is returned.
. . .

```

Explicit and implicit clusters

The clustering feature can be turned on automatically for a Naming Service. The caveat is that once this facility is on, a cluster will be created transparently to bind the object. The criterion used is fixed to be round robin. The implication is that it is possible to bind several objects to the same name in the Naming Server. Conversely, resolving that name will return one of those objects, and an `unbind` operation would destroy the cluster associated with that name. This would mean that the Naming Service is no longer compliant to the CORBA specification. The *Interoperable Naming Specification* explicitly forbids the ability to bind several objects to the same name. For in a compliant Naming Service, an `AlreadyBound` exception would be thrown if a client tries to use the same name to bind to a different object. The user has to decide, right from the beginning, whether to use this feature for a particular server and should stick with that decision.

Note Do not switch from an implicit cluster mode to an explicit cluster mode as this can corrupt the backing store.

Once a Naming Server is used with the implicit clustering feature, it should continue to be activated with that feature turned on. To turn the feature on, you define the following property value in the configuration file:

```
vbroker.naming.propBindOn=1
```

Note For an example of both explicit and implicit clustering, see the code in the `examples/ins/implicit_clustering` and `examples/ins/explicit_clustering` directories.

Load balancing

Both the ClusterManager and the Smart Agent provide round robin load balancing facilities, but they are of very different nature. You get load balancing from Smart Agent for free. When a server startups, it registered itself automatically with the Smart Agent, and this in turn allows VisiBroker to provide an easy but proprietary way for the client to get a reference to the server. However, all these automation comes at a price. You have no choice in determining what constitutes a group and the members of a group. The Smart Agent makes all the decisions for you. This is where a Cluster comes in to provide an alternative. It provides a programmatic way to define and create the properties of a Cluster. You are allowed to define the criterion to impose on a Cluster and full flexibility in choosing the members of a Cluster. Though the criterion is fixed at creation time, the client can add or remove members from the Cluster throughout its lifetime.

Failover

The Naming Service implements a failover feature using a Master/Slave model. Two naming servers must be running at the same time, the master in active mode and the slave in standby mode. Both the master and slave naming servers must support the same underlying data in a persistent backing store, and the caching facility for both servers must be off, which forces each server to deal directly with its backing store ensuring that its data remains constant.

If both naming servers are active, the master is always preferred by clients that are using Naming Service. In the event that the master terminates unexpectedly, the slave naming server will take over. This changeover from master to slave is seamless and transparent to clients. However, the slave naming server does not become the master server. Instead, it provides temporary backup when the master server is unavailable. Meantime, the user should take whatever remedial actions are necessary to revive the crashed master server. After the master comes back up again, only requests from the new clients are sent to the master server. Clients that are already bound to a slave naming server will not automatically switch back to the master

When failover occurs, it is transparent to the client, but there may be a slight delay because server objects on the slave naming server may have to be activated on demand by the requests that are coming in. Also, of the kinds of object references which a client may be holding, transient ones like iterator references are no longer valid. This is normal because clients using transient iterator references must be prepared for those references becoming invalid. In general, a naming server never

keeps too many resource-intensive iterator objects, and it may invalidate a client's iterator reference at any time. Other than these transient references, any other client request using persistent references will be re-routed to the slave naming server.

Note Clients that are already bound to a slave naming server will not automatically switch back to the master, providing only one level of failover support. Therefore, if the slave naming server also dies, the Naming Service becomes unavailable.

Configuring the Naming Service for fault tolerance

Two naming servers must be running. You must designate one of them as the master and the other as the slave. The same property file can be used for both the servers. The relevant property values in the property file are illustrated in Code sample 18.5.

Code sample 18.5 Configuration for using fault tolerance

```
vbroker.naming.enableSlave=1
vbroker.naming.masterServerName=<Master Naming Server Name>
vbroker.naming.masterHost=<host ip address for Master>
vbroker.naming.masterPort=<port number that Master is listening on>
vbroker.naming.slaveServerName=<Slave Naming Server Name>
vbroker.naming.slaveHost=<host ip address for Slave>
vbroker.naming.slavePort=<Slave Naming Server Name>
```

In order to force the Naming Server to start on a particular port, the Naming Server should be started with the following command line option

```
prompt> nameserv -J-Dvbroker.se.iioptp.scm.iioptp.listener.port=<port number> \
com.inprise.vbroker.naming.ExtFactory <Naming_Server_Name>
```

Note There is no restriction on the order in which the master and the slave servers should be started.

Import statements for Java

The following import statement should be used by any Java class that wishes to use the VisiBroker extensions to the Naming Service:

```
import com.inprise.vbroker.CosNamingExt.*;
. . .
```

The following packages are needed if you are interested in accessing the OMG compliant features of the Naming Service.

```
import org.omg.CosNaming.*
import org.omg.CosNaming.NamingContextPackage.*
import org.omg.CosNaming.NamingContextExtPackage.*
```

Sample programs

Several example programs that illustrate the use of the Naming Service are provided with VisiBroker. They illustrate all of the new features that are now available with the Naming Service and they can be found in the `examples/ins` directory. In addition, a Bank Naming example that illustrates basic usage of the Naming Service can be found in the `examples/basic/bank_naming` directory.

Before running the example programs, you must first start the naming service, as described in “Running the Naming Service” on page 18-6. Furthermore, you must ensure that at least one naming context has been created by doing one of the following:

- Start the Naming Service, as described in “Running the Naming Service” on page 18-6, which will automatically create an initial context.
- Use the VisiBroker Console, described in “What is the VisiBroker Console?” on page 11-1.
- Have your client bind to the `NamingContextFactory` and use the `create_context` method.
- Have your client use the `ExtendedNamingContextFactory`.

Warning If no naming context has been created, a `CORBA.NO_IMPLEMENT` exception will be raised when the client attempts to issue a `bind`.

Binding a name in Java

The Bank Naming example uses the `AccountManager` interface to open an `Account` and to query the balance in that account. The `Server` class below illustrates the usage of the Naming Service for binding a name to an object reference. The server publishes its IOR into the root context of the Naming Server, which is then retrieved by the client.

From this example, you learn how to:

- 1 Use the `resolve_initial_references` method on an ORB instance to get a reference to the root context of the Naming Service. (In the example, you need to start the Naming Service with the default name of `NameService`.)
- 2 Cast the reference for the root context by using the `narrow` method of the `NamingContextExtHelper` class.
- 3 Create a POA and servant for your `AccountManagerImpl` object.
- 4 Finally use the `bind` method of the `NamingContext` interface to bind the Name “BankManager” to the object reference for the `AccountManagerImpl` object.

Code sample 18.6 `Server.java`

```
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;

public class Server {
```

```

public static void main(String[] args) {
    try {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // get a reference to the root POA
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        // get a reference to the Naming Service root context
        org.omg.CORBA.Object rootObj = orb.resolve_initial_references("NameService");
        NamingContextExt root = NamingContextExtHelper.narrow(rootObj);

        // Create policies for our persistent POA
        org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
        };
        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA( "bank_agent_poa", rootPOA.the_POAManager(),
            policies );
        // Create the servant
        AccountManagerImpl managerServant = new AccountManagerImpl();
        // Decide on the ID for the servant
        byte[] managerId = "BankManager".getBytes();
        // Activate the servant with the ID on myPOA
        myPOA.activate_object_with_id(managerId, managerServant);

        // Activate the POA manager
        rootPOA.the_POAManager().activate();

        // Associate the bank manager with the name at the root context
        // Note that casting is needed as a workaround for a JDK 1.1.x bug.
        ((NamingContext)root).bind(root.to_name("BankManager"),
            myPOA.servant_to_reference(managerServant));

        System.out.println(myPOA.servant_to_reference(managerServant)
            + " is ready.");
        // Wait for incoming requests
        orb.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Using the Event Service

This chapter describes the VisiBroker Event Service.

Note The OMG (Object Management Group) Event Service has been superseded by the OMG Notification Service. We strongly recommend the separately available OpenFusion Notification Service to VisiBroker customers needing such functionality.

Overview

The Event Service package provides a facility that de-couples the communication between objects. It provides a *supplier-consumer* communication model that allows multiple *supplier objects* to send data asynchronously to multiple *consumer objects* through an event channel. The supplier-consumer communication model allows an object to communicate an important change in state, such as a disk running out of free space, to any other objects that might be interested in such an event.

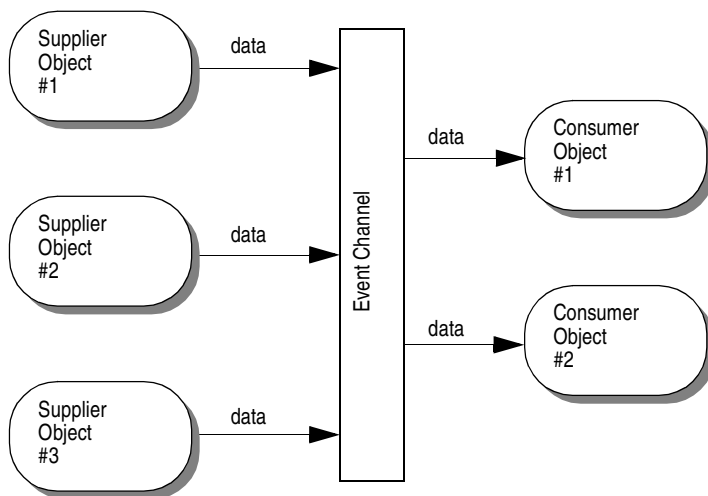
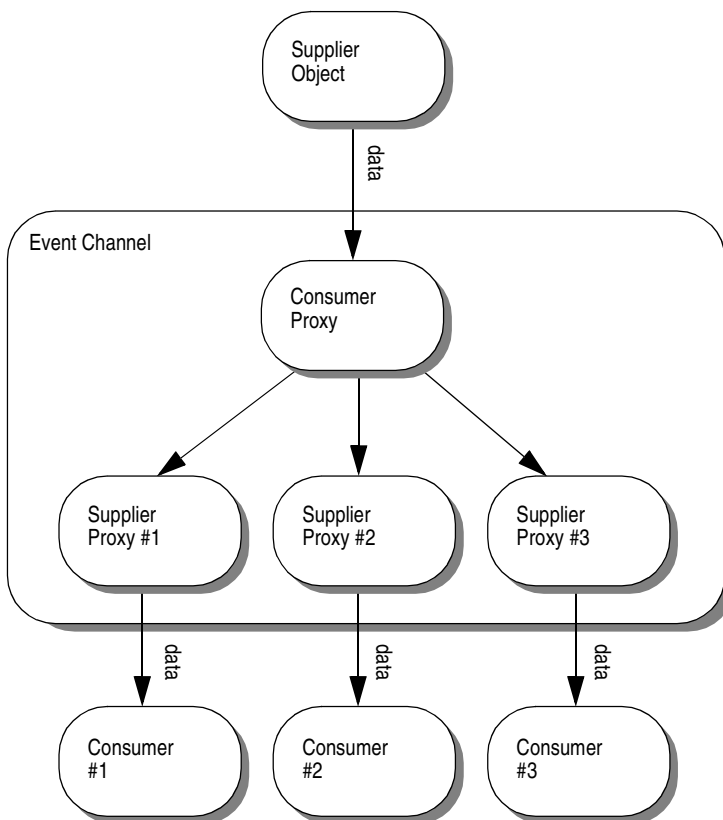
Figure 19.1 Supplier-Consumer communication model

Figure 19.1 shows three supplier objects communicating through an event channel with two consumer objects. The flow of data into the event channel is handled by the supplier objects, while the flow of data out of the event channel is handled by the consumer objects. If each of the three suppliers shown in Figure 19.1 sends one message every second, then each consumer will receive three messages every second and the event channel will forward a total of six messages per second.

The event channel is both a consumer of events and a supplier of events. The data communicated between suppliers and consumers is represented by the `Any` class, allowing any CORBA type to be passed in a type safe manner. Supplier and consumer objects communicate through the event channel using standard CORBA requests.

Proxy consumers and suppliers

Consumers and suppliers are completely de-coupled from one another through the use of *proxy objects*. Instead of interacting with each other directly, they obtain a proxy object from the `EventChannel` and communicate with it. Supplier objects obtain a *consumer proxy* and consumer objects obtain a *supplier proxy*. The `EventChannel` facilitates the data transfer between consumer and supplier proxy objects. Figure 19.2 shows how one supplier can distribute data to multiple consumers.

Figure 19.2 Consumer and supplier proxy objects

Note The event channel is shown in Figure 19.2 as a separate process, but it may also be implemented as part of the supplier object's process. See "Starting the Event Service" on page 19-14 for more information.

OMG common object services specification

The VisiBroker Event Service implementation conforms to the OMG Common Object Services Specification, with the following exceptions:

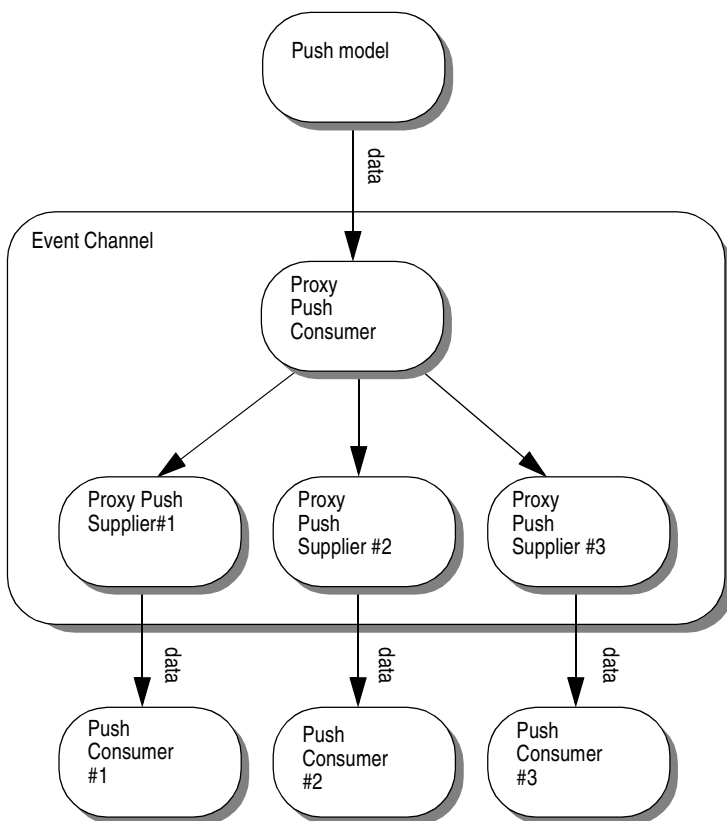
- The VisiBroker Event Service only supports generic events. There is currently no support for typed events in the VisiBroker Event Service.
- The VisiBroker Event Service offers no confirmation of the delivery of data to either the event channel or to consumer applications. TCP/IP is used to implement the communication between consumers, suppliers and the event channel and this provides reliable delivery of data to both the channel and the consumer. However, this does not guarantee that all of the data that is sent is actually processed by the receiver.

Communication models

The event service provides both a *pull* and *push* communication model for suppliers and consumers. In the *push model*, supplier objects control the flow of data by *pushing* it to consumers. In the *pull model*, consumer objects control the flow of data by *pulling* data from the supplier.

The `EventChannel` insulates suppliers and consumers from having to know which model is being used by other objects on the channel. This means that a pull supplier can provide data to a push consumer and a push supplier can provide data to a pull consumer.

Figure 19.3 Push model



Note The event channel is shown in Figure 19.3 as a separate process, but it may also be implemented as part of the supplier object's process. See "Starting the Event Service" on page 19-14 for more information.

Push model

The *push model* is the more common of the two communication models. An example use of the push model is a supplier that monitors available free space on a disk and notifies interested consumers when the disk is filling up. The push supplier sends data to its `ProxyPushConsumer` in response to events that it is monitoring.

The push consumer spends most of its time in an event loop, waiting for data to arrive from the `ProxyPushSupplier`. The `EventChannel` facilitates the transfer of data from the `ProxyPushSupplier` to the `ProxyPushConsumer`.

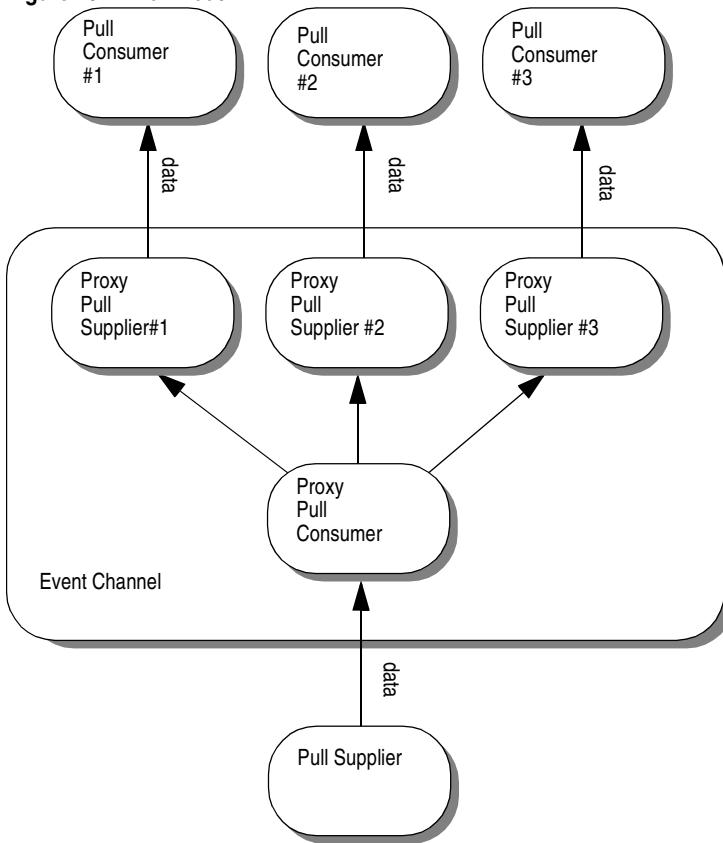
Figure 19.3 shows a push supplier and its corresponding `ProxyPushConsumer` object. It also shows three push consumers and their respective `ProxyPushSupplier` objects.

Pull model

In the *pull model*, the event channel regularly pulls data from a supplier object, puts the data in a queue, and makes it available to be *pulled* by a consumer object. An example of a pull consumer would be one or more network monitors that periodically poll a network router for statistics.

The pull supplier spends most of its time in an event loop waiting for data requests to be received from the `ProxyPullConsumer`. The pull consumer requests data from the `ProxyPullSupplier` when it is ready for more data. The `EventChannel` pulls data from the supplier to a queue and makes it available to the `ProxyPullSupplier`.

Figure 19.4 shows a pull supplier and its corresponding `ProxyPullConsumer` object. It also shows three pull consumers and their respective `ProxyPullSupplier` objects.

Figure 19.4 Pull model

Note The event channel is shown in Figure 19.4 as a separate process, but it may also be implemented as part of the supplier object's process. See "In-process event channel" on page 19-15 for more information.

Using event channels

To create an EventChannel, connect a supplier or consumer to it and use it:

1 Create and start the EventChannel.

To create and start the event channel,

Windows

```
prompt> start vbj com.inprise.vbroker.CosEvent.EventServer -ior <iorFilename>
<channelName>
```

UNIX

```
prompt> vbj com.inprise.vbroker.CosEvent.EventServer -ior <iorFilename> <channelName> &
```

where `<channelName>` is the user-specified object name of the event channel and `<iorFilename>` is a user-specified filename of the file to which the `ior` of the object is to be written.

Another way to create the EventChannel is to run PushModelChannel:

```
prompt> vbj PushModelChannel
```

PushModelChannel first creates an EventChannel and publishes its `ior` to the file `<iorFilename>` given by the user. Other clients (for example, PushModel) can then bind to the EventChannel by using initial reference.

To run this:

```
prompt> vbj -DORBInitRef=EventService=file:<fullpath + iorFilename> PushModel
```

Regardless of how the event channel is created, make sure that the name specified in `<iorFilename>` is created in the specified directory.

Note: Only one instance of the EventChannel is supported. All binding to the EventChannel is done through the call to `orb.resolve_initial_references("EventService")`, where "EventService" is the hardcoded EventChannel name.

- 2 Connect to the EventChannel.
- 3 Obtain an administrative object from the channel and use it to obtain a proxy object.
- 4 Connect to the proxy object.
- 5 Begin transferring or receiving data.

The methods used for these steps vary, depending on whether the object being connected is a supplier or a consumer, and on the communication model being used. Table 19.1 shows the appropriate methods for suppliers and Table 19.2 shows the methods for consumers.

Table 19.1 Connecting Suppliers to an EventChannel

Steps	Push supplier	Pull supplier
Bind to the EventChannel	<code>EventChannelHelper.narrow(orb.resolve_initial_references("EventService"))</code>	<code>EventChannelHelper.narrow(orb.resolve_initial_references("EventService"))</code>
Get a SupplierAdmin	<code>EventChannel::for_suppliers()</code>	<code>EventChannel::for_suppliers()</code>
Get a consumer proxy	<code>SupplierAdmin::obtain_push_consumer()</code>	<code>SupplierAdmin::obtain_pull_consumer()</code>
Add the supplier to the EventChannel	<code>ProxyPushConsumer::connect_push_supplier()</code>	<code>ProxyPullConsumer::connect_pull_supplier()</code>
Data transfer	<code>ProxyPushConsumer::push()</code>	Implements <code>pull()</code> and <code>try_pull()</code>

Table 19.2 Connecting Consumers to an EventChannel

Steps	Push consumer	Pull consumer
Bind to the EventChannel	EventChannelHelper.narrow(orb. resolve_initial_references("EventService"))	EventChannelHelper.narrow(orb. resolve_initial_references("EventService"))
Get a ConsumerAdmin	EventChannel::for_consumers()	EventChannel::for_consumers()
Obtain a supplier proxy	ConsumerAdmin::obtain_push_supplier()	ConsumerAdmin::obtain_pull_supplier()
Add the consumer to the EventChannel	ProxyPushSupplier::connect_push_consumer()	ProxyPushSupplier::connect_pull_consumer()
Data transfer	Implements push()	ProxyPushSupplier::pull() and try_pull()

Example push supplier and consumer

Running the Push model example

To run the PushModel example, at the prompt, enter:

```
prompt> vbj -DORBInitRef=EventService=file:<fullpath of iorFilename> PushModel
```

Select **e** to bind to an event channel, **p** to get a proxy to a push consumer from the event channel, **m** to instantiate a PushModel, and **c** to connect the event channel. Continuous sentences indicating the content of the message being pushed to the EventChannel will be displayed. You can continue to make selections regardless of what is displayed on the screen. You can specify the number of seconds between events using the **s** option. Lastly, select **d** to disconnect and **q** to quit.

To run the PushView, at the prompt, enter:

```
prompt> vbj -DORBInitRef=EventService=file:<fullpath of iorFilename> PushView
```

Select **e** to bind to an event channel, **p** to get a proxy to a push supplier from the event channel, **v** to instantiate a PushView, **c** to connect the event channel, **d** to disconnect and **q** to quit. To run this example, a supplier of type Push or Pull must be running on another terminal, continuously sending data to the same event channel in order for PushView to receive the data. The supplier and consumer can be started in any order.

Running the Pull model example

To run the PullModel example, at the prompt, enter:

```
prompt> vbj -DORBInitRef=EventService=file:<fullpath of iorFilename> PullModel
```

Select **e** to bind to an event channel, **p** to get a proxy to a push consumer from the event channel, **m** to instantiate a PullModel, **c** to connect the event channel, **d** to disconnect and **q** to quit.

To run the PullView, at the prompt, enter:

```
prompt> vbj -DORBInitRef=EventService=file:<fullpath of iorFilename> PullView
```

Select **e** to bind to an event channel, **p** to get a proxy to a push supplier from the event channel, **v** to instantiate a PushView, **c** to connect the event channel. Then select **a** to pull asynchronously or **s** to pull synchronously. To exit, select **d** to disconnect and **q** to quit.

To run this example, a supplier of type Push or Pull must be running on another terminal, continuously sending data to the same event channel in order for PullView to receive the data. The supplier and consumer can be started in any order.

This section describes the example *push* supplier and consumer applications. The files PullSupply.java and PullConsume.java implement the supplier and consumer. These files can be found in the `java_examples/events` directory, under the directory where the VisiBroker for Java distribution was installed on your system.

To run these examples, you need a supplier-consumer pair. You can pair a consumer of type Push or Pull can be paired with any supplier of type Push or Pull. The order in which you invoke the supplier and consumer does not matter. However, the event channel must be the same object instance.

PullSupply

The PullSupply class is derived from the PullSupplierPOA class and provides implementations for the `main`, `pull` and `try_pull` methods. The `pull` method, shown in Code sample 19.1, returns a numbered “hello” message. The `try_pull` method always sets the `hasEvent` flag to true and calls the `pull` method to provide the message. Once a PullSupply object is connected to an EventChannel, these methods are used by the channel to pull data from the supplier.

The `main` method, shown in Code sample 19.2, performs the usual ORB and POA creation, connects to the specified EventChannel, obtains a ProxyPullConsumer from the EventChannel, instantiates a PullSupply object, activates the PullSupply object on the POA, then connects this pull supplier to proxy pull consumers.

Executing PullSupply

After compiling `PullSupply.java` and starting the event service, described in “In-process event channel” on page 19-15, you can execute the supplier with the following command:

```
vbj -DORBInitRef = <channel_name> = file:<fullpath of iOrFilename> PullSupply
```

Code sample 19.1 Implementation of the pull and try_pull methods

```
// PullSupply.java
import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
import org.omg.PortableServer.*;

public class PullSupply extends PullSupplierPOA {
    private POA _myPOA;
    private PullConsumer _pullConsumer;
    private int _counter;
    PullSupply(PullConsumer pullConsumer, POA myPOA) {
        _pullConsumer = pullConsumer;
        _myPOA = myPOA;
    }
    public void disconnect_pull_supplier() {
        System.out.println("Model::disconnect_pull_supplier()");
        try {
            _myPOA.deactivate_object("PullSupply".getBytes());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public org.omg.CORBA.Any pull() throws Disconnected {
        if (_pullConsumer == null) {
            throw new Disconnected();
        }
        try {
            Thread.currentThread().sleep(1000);
        } catch (Exception e) {
        }
        //org.omg.CORBA.Any message =
            new org.omg.CORBA.Any().from_string("Hello #" + ++_counter);
        org.omg.CORBA.Any message = _orb().create_any();
        message.insert_string("Hello #" + ++_counter);
        System.out.println("Supplier being pulled: " + message);
        return message;
    }
    public org.omg.CORBA.Any try_pull(org.omg.CORBA.BooleanHolder hasEvent) throws
        org.omg.CORBA.SystemException, Disconnected {
        hasEvent.value = true;
        return pull();
    }
    . . .
}
```


Code sample 19.2 main method of PullSupply

```

// PullSupply.java
import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
import org.omg.PortableServer.*;

public class PullSupply extends PullSupplierPOA {
    . . .
    public static void main(String[] args) {
        try {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // get a reference to the root POA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Create policies for our persistent POA
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
            };
            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA("event_service_poa",
                rootPOA.the_POAManager(), policies);
            EventChannel channel = null;
            PullSupply model = null; ProxyPullConsumer pullConsumer = null;
            channel =
                EventChannelHelper.narrow(orb.resolve_initial_references("EventService"));
            System.out.println("Located event channel: " + channel);
            pullConsumer = channel.for_suppliers().obtain_pull_consumer();
            System.out.println("Obtained pull consumer: " + pullConsumer);
            model = new PullSupply(pullConsumer, myPOA);
            myPOA.activate_object_with_id("PullSupply".getBytes(), model);
            myPOA.the_POAManager().activate();
            System.out.println("Created model: " + model);
            System.out.println("Connecting ...");
            pullConsumer.connect_pull_supplier(model._this());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

PullConsume

The `PullConsume` class is derived from `PullConsumerPOA` class and provides a command line interface for pulling data from the `PullSupply` class. Code sample 19.3 shows how the application connects to any available `EventChannel`, obtains a `ProxyPullSupplier`, connects to the channel and displays a command prompt. Table 19.3 summarizes the commands that may be entered.

Table 19.3 PullConsume commands

Command	Description
a	Asynchronously pulls data from the event channel, using the <code>try_pull</code> method. If no data is currently available, the command will return with a “no data” message.
s	Synchronously pulls data from the event channel, using the <code>pull</code> method. If there is no data currently available, the command will block until data is available.
q	Disconnects from the channel and exits the tool.

Executing PullConsume

After compiling `PullConsume.java` and starting the event service, described in “In-process event channel” on page 19-15, you can execute the consumer with the following command:

```
vbj -DORBInitRef = <channel_name> = file:<fullpath of iOr_filename> PullConsume
```

Code sample 19.3 Example pull consumer

```
// PullConsume.java
import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
import org.omg.PortableServer.*;
import java.io.*;

public class PullConsume extends PullConsumerPOA {
    public void disconnect_pull_consumer() {
        System.out.println("View.disconnect_pull_consumer");
    }

    public static void main(String[] args) {
        try {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // get a reference to the root POA
            POA rootPOA =
                POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Create policies for our persistent POA
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
            };
            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA("event_service_poa",
                rootPOA.the_POAManager(), policies );
            EventChannel channel = null;
            PullConsume view = null;
            ProxyPullSupplier pullSupplier = null;
```

```

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
channel =
    EventChannelHelper.narrow(orb.resolve_initial_references("EventService"));
System.out.println("Located event channel: " + channel);
view = new PullConsume();
myPOA.activate_object_with_id("PullConsume".getBytes(), view);
myPOA.the_POAManager().activate();
System.out.println("Created view: " + view);
pullSupplier = channel.for_consumers().obtain_pull_supplier();
System.out.println("Obtained pull supplier: " + pullSupplier);
System.out.println("Connecting...");
System.out.flush();
pullSupplier.connect_pull_consumer(view._this());
while(true) {
    System.out.print("-> ");
    System.out.flush();
    if (System.getProperty("VM_THREAD_BUG") != null) {
        while(!in.ready()) {
            try {
                Thread.currentThread().sleep(100);
            } catch (InterruptedException e) {
            }
        }
    }
    String line = in.readLine();
    if(line.startsWith("a")) {
        org.omg.CORBA.BooleanHolder hasEvent = new org.omg.CORBA.BooleanHolder();
        org.omg.CORBA.Any result = pullSupplier.try_pull(hasEvent);
        System.out.println("try_pull: " +
            (hasEvent.value ? result.toString() : "NO DATA"));
        continue;
    } else if(line.startsWith("s")) {
        org.omg.CORBA.Any result = pullSupplier.pull();
        System.out.println("pull: " + result);
        continue;
    } else if(line.startsWith("q")) {
        System.out.println("Disconnecting...");
        pullSupplier.disconnect_pull_supplier();
        System.out.println("Quitting...");
        break;
    }
    System.out.println("Commands: a  [a]synchronous pull\n" +
        "          s  [s]ynchronous pull\n" +
        "          q  [q]uit\n");
}
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

Starting the Event Service

When using VisiBroker for Java, the event service can be started by using the following command.

```
vbj [-Dvbroker.events.debug] [-Dvbroker.events.interactive]
[-Dvbroker.events.max_queue_length=<number>] [-Dvbroker.events.debug.factory] \
[-Dvbroker.events.vm_thread_bug] com.inprise.vbroker.CosEvent.EventServer
-ior <ior filename> <channel name>
```

Option	Description
-Dvbroker.events.debug	Optional parameter that enables the output of debugging messages to stdout.
-Dvbroker.events.interactive	Specifies that the event channel is to execute in a console-driven, interactive mode.
- Dvbroker.events.maxQueueLength	Specifies the number of messages to be queued for slow consumers. The default maximum queue length is 100 messages for each consumer.
-Dvbroker.events.factory	Specifies that an event channel factory is to be instantiated instead of an event channel.
channel_name	The name of the channel or channel factory.

Note There is a known bug in some implementations of the Java Virtual Machine, including Solaris, that may cause this command to hang. If you experience difficulties, try specifying the `-Dvbroker.events.vm_thread_bug` parameter when you start the event service.

Setting the queue length

In some environments, consumer applications may run slower than supplier applications. The `maxQueueLength` parameter prevents out-of-memory conditions by limiting the number of outstanding messages that will be held for each consumer that cannot keep up with the rate of messages from the supplier.

If a supplier generates 10 messages per second and a consumer can only process one message per second, the queue will quickly fill up. Messages in the queue have a fixed maximum length and if an attempt is made to add a message to a queue that is full, the channel will remove the oldest message in the queue to make room for the new message.

Each consumer has a separate queue, so a slow consumer may miss messages while another, faster consumer may not lose any. Code sample 19.4 shows how to limit each consumer to 15 outstanding messages.

Code sample 19.4 Setting the message queue length

```
vbj -Dvbroker.events.maxQueueLength=15 CosEvent.EventServer -ior myChannel.ior MyChannel
```

Note If `maxQueueLength` is not specified or if an invalid number is specified, a default queue length of 100 is used.

In-process event channel

In addition to running an `EventChannel` as a separate, stand-alone server, the Event Service also allows you to create an `EventChannel` within your server or client application. This frees you from having to start a separate process to provide the `EventChannel` for your supplier or consumer applications.

For Java applications, an `EventLibrary` class is provided that provides methods for creating an `EventChannel` which, in turn, handles the loading of the necessary classes. To create an in-process `EventChannel` object within a supplier/consumer application, make the following call:

```
EventLibrary.create_Channel("MyChannel",whetherToDebug,maxQueueLength);
```

So, to create a channel named `MyChannel` with debugging off and a maximum queue length of 100, you would write:

```
EventLibrary.create_Channel("MyChannel",false,100);
```

After this call completes, the resulting client application can bind to the `EventChannel` as it would bind to any other CORBA object.

For example, you might have a supplier application creating the channel in-process and want the consumer application to connect to the same channel. To accomplish this, you need to pass the channel object from the supplier application to the consumer application. To do this, convert the `EventChannel` object to an ior string and write the string to a file:

```
try {
    EventChannel channel = EventLibrary.create_Channel("MyChannel",false,100);
    PrintWriter pw = new PrintWriter(new FileWriter(ior_filename));
    pw.println(orb.object_to_string(channel));
    pw.close();
}
catch(IOException e) {
    System.out.println("Error writing the IOR to file " ior_filename);
}
```

The `ior_filename` specifies the name of the file to which the ior string of the channel will be written.

To run `PushModelChannel`:

```
vbj PushModelChannel <ior_filename>
```

`PushModelChannel` is a push supplier. You can connect either a push consumer or pull consumer to the event channel created in `PushModelChannel`:

```
vbj -DORBInitRef=EventService=file:<fullpath of ior_filename> PushView
```

where `<fullpath of ior_filename>` is the full path of the `ior_filename` passed into `PushModelChannel` and `EventService` is the name (or identifier) bound to the ior contained in `ior_filename`. From within `PushView`, you can bind to the event channel as follows:

```
EventChannel channel =
    EventChannelHelper.narrow(orb.resolve_initial_references("EventService"));
```

Java usage

If your application uses the in-process event channel feature, you must add the following `import` statement:

```
import com.inprise.vbroker.CosEvent.*;
```

Java EventLibrary class

The `EventLibrary` class provides several methods for creating an `EventChannel` within an application’s process. See “`EventLibrary (Java)`” on page 19-17 for a complete description of these methods.

Java example

The file `PushModelChannel.java` implements a push supplier that uses an in-process event channel. This application presents a command prompt and allows you to enter one of the commands shown below.

Command	Description
e	Creates an event channel.
s <number_of_seconds>	Sets the delay for the event channel to the number of seconds specified, which must be a non-negative number.
p	Obtains a push consumer proxy object.
m	Creates a <code>PushModelChannel</code> and activates it on the POA.
c	Connects the push supplier.
d	Disconnects the push consumer.
q	Quits the application.

Code sample 19.5 contains an excerpt from `PushModelChannel.java` that shows how you can use the `ChannelLib.create_channel` method.

Code sample 19.5 Portion of `PushModelChannel.java` that shows the use of the `create_channel` method

```
public static void main(String[] args) {  
    . . .  
    channel = EventLibrary.create_channel("channel_server", false, 100);  
    . . .  
}
```

Import statements for Java

The following import statements should be used by Java applications that wish to use the event service:

```
import org.omg.CosEventComm.*;  
import org.omg.CosEventChannelAdmin.*;  
. . .
```

Interface reference

The remainder of this chapter provides reference information on all of the Event Service interfaces.

EventChannel

The `EventChannel` provides the administrative operations for adding suppliers and consumers to the channel and for destroying the channel.

`ConsumerAdmin` **for_consumers()**;

This method returns a `ConsumerAdmin` object that can be used to obtain proxy suppliers.

`SupplierAdmin` **for_suppliers()**;

This method returns a `SupplierAdmin` object that can be used to obtain proxy suppliers.

`void` **destroy()**;

This method destroys this `EventChannel`.

EventLibrary (Java)

The `EventLibrary` class provides several methods creating an `EventChannel` within an application's process. Using an in-process event channel frees you from having to start a separate event channel or event channel factory process.

EventLibrary methods

`static org.omg.CosEventChannelAdmin.EventChannel` **create_channel**
(`String` name, `boolean` debug, `int` maxQueueLength);

This method creates an `EventChannel` with the specified name, debug, and queue length settings.

Parameter	Description
name	The name to be used for this channel.
debug	If set to <code>true</code> , debugging output is enabled. If set to <code>false</code> , debugging output is disabled.
maxQueueLength	The maximum number of messages that may be queued for each consumer.

```
static org.omg.CosEventChannelAdmin.EventChannel create_channel
(String name, boolean debug);
```

This method creates an `EventChannel` with the specified name and debug settings. The queue length for each consumer is set to 100.

Parameter	Description
name	The name to be used for this channel.
debug	If set to <code>true</code> , debugging output is enabled. If set to <code>false</code> , debugging output is disabled.

```
static org.omg.CosEventChannelAdmin.EventChannel create_channel
( String name);
```

This method creates an `EventChannel` with the specified name. The `EventChannel` object's debug flag is set to `false` and the queue length is set to 100.

Parameter	Description
name	The name to be used for this channel.

```
static org.omg.CosEventChannelAdmin.EventChannel create_channel( );
```

This method creates an `EventChannel`. The `EventChannel` object is given no name, the debug flag is set to `false`, and the queue length is set to 100.

ConsumerAdmin

This interface is used by consumer applications to obtain a reference to a proxy supplier object. This is the second step in connecting a consumer application to an `EventChannel`.

Code sample 19.6 ConsumerAdmin interface

```
module CosEventChannelAdmin {
    interface ConsumerAdmin {
        ProxyPushSupplier obtain_push_supplier();
        ProxyPullSupplier obtain_pull_supplier();
    };
};
```

The `obtain_push_supplier` method is invoked if the calling consumer application is implemented using the push model. If the application is implemented using the pull model, the `obtain_pull_supplier` method should be invoked.

The returned reference is used to invoke either the `connect_push_consumer`, described in “ProxyPushConsumer” on page 19-20, or the `connect_pull_consumer` method, described in “EventLibrary (Java)” on page 19-17.

SupplierAdmin

This interface is used by supplier applications to obtain a reference to the proxy consumer object. This is the second step in connecting a supplier application to an EventChannel.

Code sample 19.7 SupplierAdmin interface

```
module CosEventChannelAdmin {
    interface SupplierAdmin {
        ProxyPushConsumer obtain_push_consumer();
        ProxyPullConsumer obtain_pull_consumer();
    };
};
```

Invoke the `obtain_push_consumer` method if the supplier application is implemented using the push model. If the application is implemented using the pull model, the `obtain_pull_consumer` method should be invoked.

The returned reference is used to invoke either the `connect_push_supplier`, described in “ProxyPushSupplier” on page 19-20, or the `connect_pull_supplier` method, described in “ProxyPullSupplier” below.

ProxyPullConsumer

This interface is used by a pull supplier application and provides the `connect_pull_supplier` method for connecting the supplier’s `PullSupplier`-derived object to the EventChannel. An `AlreadyConnected` exception is raised if `ProxyConsumer` is already connected to a `PullSupplier`.

Code sample 19.8 ProxyPullConsumer interface

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPullConsumer : CosEventComm::PullConsumer {
        void connect_pull_supplier(
            in CosEventComm::PullSupplier pull_supplier)
            raises(AlreadyConnected);
    };
};
```

ProxyPushConsumer

This interface is used by a push supplier application and provides the `connect_push_supplier` method, used for connecting the supplier's `PushSupplier`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if `ProxyConsumer` is already connected to a `PullSupplier`.

Code sample 19.9 ProxyPushConsumer interface

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPushConsumer : CosEventComm::PushConsumer {
        void connect_push_supplier(
            in CosEventComm::PushSupplier push_supplier)
            raises(AlreadyConnected);
    };
};
```

ProxyPullSupplier

This interface is used by a pull consumer application and provides the `connect_pull_consumer` method, used for connecting the consumer's `PullConsumer`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if `ProxyConsumer` is already connected to a `PullConsumer`.

Code sample 19.10 ProxyPullSupplier interface

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPullSupplier : CosEventComm::PullSupplier {
        void connect_pull_consumer(
            in CosEventComm::PullConsumer pull_consumer)
            raises(AlreadyConnected);
    };
};
```

ProxyPushSupplier

This interface is used by a push consumer application and provides the `connect_push_consumer` method, used for connecting the consumer's `PushConsumer`-derived object to the `EventChannel`. An `AlreadyConnected` exception is raised if `ProxyConsumer` is already connected to a `PullSupplier`.

Code sample 19.11 ProxyPushSupplier interface

```
module CosEventChannelAdmin {
    exception AlreadyConnected();
    interface ProxyPushSupplier : CosEventComm::PushSupplier {
        void connect_push_consumer(
            in CosEventComm::PushConsumer push_consumer)
            raises(AlreadyConnected);
    };
};
```

PullConsumer

This interface is used to derive consumer objects that use the pull model of communication. The `pull` method is called by a consumer whenever it wants data from the supplier. A `Disconnected` exception is raised if the supplier is already disconnected.

The `disconnect_push_consumer` method is used to deactivate this consumer if the channel is destroyed.

```
module CosEventComm {
    exception Disconnected {};
    interface PullConsumer {
        void disconnect_pull_consumer();
    };
};
```

The only method that must be implemented in the derived classes of `PullConsumer` is `disconnect_pull_consumer`, which is used to disconnect the `PullConsumer` from the `EventChannel`. For instance, in the `PullModel` example, the `PullSupplier` uses it to disconnect the pull consumer.

PushConsumer

This interface is used to derive consumer objects that use the push model of communication. The `push` method is used by a supplier whenever it has data for the consumer. It raises a `Disconnected` exception if the consumer has already been disconnected.

Code sample 19.12 PushConsumer interface

```
module CosEventComm {
    exception Disconnected();
    interface PushConsumer {
        void push(in any data) raises(Disconnected);
        void disconnect_push_consumer();
    };
};
```

The `PushConsumer` implements the `push(in any data)` method. This method is called by the `PushSupplier` continuously to receive data until the `PushSupplier` is explicitly disconnected from the `PushConsumer` by a call to `disconnect_push_supplier` on the `ProxyPushSupplier` object.

PullSupplier

This interface is used to derive supplier objects that use the pull model of communication.

Code sample 19.13 PullSupplier interface

```
module CosEventComm {
    exception Disconnected{};
    interface PullSupplier {
        any pull() raises(Disconnected);
        any try_pull() raises(Disconnected);
        void disconnect_pull_supplier();
    };
};
```

The PullConsumer pulls data from a PullSupplier. Once connected to a ProxyPullSupplier, PullConsumer can pull() or try_pull() on the ProxyPullSupplier object. try_pull() is for asynchronous pull (returns immediately, even if the data is not yet available) and pull() is for synchronous pull (returns when the data is available).

PullConsumer calls disconnect_pull_supplier() on ProxyPullServer when the consumer wants to disconnect from the ProxyPullSupplier. The pull() and try_pull() methods return CORBA::Any objects. In the example, the returned Any object contains a numbered string that contains the value "Hello."

PullSupplier methods

any pull();

This method blocks until there is data available from the supplier. The data is returned an Any type. If the consumer has disconnected, this method raises a Disconnected exception.

any try_pull(out boolean has_event);

This non-blocking method attempts to retrieve data from the supplier. When this method returns, has_event is set to CORBA::TRUE and the data is returned as an Any type if there was data available. If has_event is set to CORBA::FALSE, then no data was available and the return value will be NULL.

void disconnect_pull_supplier();

This method deactivates this pull server if the channel is destroyed.

PushSupplier

This interface is used to derive supplier objects that use the push model of communication. The `disconnect_push_supplier` method is used by the `EventChannel` to disconnect supplier when it is destroyed.

Code sample 19.14 PushSupplier interface

```
module CosEventComm {
    exception Disconnected();
    interface PushSupplier {
        void disconnect_push_supplier();
    };
};
```

`PushSupplier` should be implemented so that it constantly “pushes” data to the consumer. In the `PushModel` example, once a `PushModel` object (a `PushSupplier`-derived object) is created, it starts a new `Thread` that keeps calling `push(CORBA.Any)` on the `ProxyPushConsumer` at intervals. The pushed data is an `Any` with a message string (numbered Hello string) inserted.

The only method that must be implemented in the derived classes of `PushSupplier` is `disconnect_pull_consumer`, which is used to disconnect the `PullConsumer` from the `EventChannel`. for instance, in the `PushView` example, the `PushConsumer` uses it to disconnect the `ProxyPushSupplier`.

Using the Object Activation Daemon

This chapter discusses how to use the Object Activation Daemon.

Automatic activation of objects and servers

The Object Activation Daemon (OAD) is VisiBroker's implementation of the Implementation Repository. The Implementation Repository provides a runtime repository of information about the classes a server supports, the objects that are instantiated, and their IDs. In addition to the services provided by a typical Implementation Repository, the OAD is used to automatically activate an implementation when a client references the object. You can register an object implementation with the OAD to provide this automatic activation behavior for your objects.

Object implementations can be registered using a command-line interface (`oadutil`). There is also an ORB interface to the OAD, described in "IDL interface to the OAD" on page 20-14. In each case, the repository id, object name, the activation policy, and the executable program representing the implementation must be specified.

Note You can use the VisiBroker for Java OAD to instantiate servers generated with VisiBroker for Java (any release) and VisiBroker for C++ release 3.0.

The OAD is a separate process that only needs to be started on those hosts where object servers are to be activated on demand.

Locating the implementation repository data

Activation information for all object implementations registered with the OAD are stored in the implementation repository. By default, the implementation repository data is stored in a file named `impl_rep`. This file's path name is dependent on the value of the `VBROKER_ADM` variable. If VisiBroker was installed in `/usr/local/vbroker/`, the path to this file would be `/usr/local/vbroker/adm/impl_dir/impl_rep`. These defaults can be overridden using the OAD environment variables, described in Chapter 3, "Setting up your environment."

Activating servers

The OAD activates servers in response to client requests. The following types of clients can activate servers through the OAD:

- VisiBroker for Java 4.x clients.
- VisiBroker for Java 3.x clients.
- Non-VisiBroker IIOP-compliant clients. Any client that follows the IIOP can activate a VisiBroker server when that server's reference is used. The server's exported Object Reference points to the OAD and the client can be forwarded to the spawned server in accordance with the rules of IIOP. To allow true persistification of the server's object references (such as through a Name Service), the OAD must always be started on the same port. In the following example, the OAD is started on port 16050.

```
prompt> oad -VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.port=16050
```

Note Port 16000 is the default port, but it can be changed by setting the `listener.port` property.

Starting the Object Activation Daemon

The object activation daemon is an optional feature that allows you to register objects that are to be started automatically when clients attempt to access them. Before starting the OAD, you should first start the Smart Agent.

Starting the Object Activation Daemon on a Windows platform

To start the OAD under Windows, select its icon from the VisiBroker Program Group or enter the following command at the DOS prompt:

```
prompt> oad
```


The `oad` command accepts the following command line arguments:

Option	Description
<code>-verbose</code>	Turns on verbose mode.
<code>-version</code>	Prints the version of this tool.
<code>-path <path></code>	Specifies the platform-specific directory for storing the implementation repository. This overrides any setting provided through the use of environment variables.
<code>-filename <repository filename></code>	Specifies the name of the implementation repository. If you do not specify it, the default is <code>impl_rep</code> . This overrides any user environment variable settings.
<code>-t <# of seconds></code>	Specifies the amount of time the OAD will wait for a spawned server process to activate the requested ORB object. The default time-out is 20 seconds. Set this value to 0 if you wish to wait indefinitely. If a spawned server process does not activate the requested object within the time-out interval, the OAD will kill the spawned process and the client will see a <code>CORBA::NO_IMPLEMENT</code> exception. Turn on the verbose option to see more detailed information.
<code>-ior <IOR filename></code>	Specifies the filename to store the OAD's stringified IOR.
<code>-kill</code>	Stipulates that an object's child process should be killed once all of its object are unregistered with the OAD.
<code>-no_verify</code>	Turns off check for validity of registrations.
<code>-?</code>	Displays command usage.
<code>-readonly</code>	When the OAD is started with the <code>-readonly</code> option, no changes can be made to the registered objects. Attempts to register or unregister objects will return an error. The <code>-readonly</code> option is usually used after you've made changes to the implementation repository, and have restarted the OAD in <code>readonly</code> mode to prevent any additional changes.

The OAD is installed as an NT service under Windows NT, allowing you to control it with the Service Manager provided with Windows NT. You may also start the OAD in console mode from the DOS prompt entering the following command:

```
prompt> oad -C
```

Starting the Object Activation Daemon on a UNIX platform

To start the OAD on a UNIX system, enter the following command.

```
prompt> oad &
```

Using the Object Activation Daemon utilities

The `oadutil` commands provides a way for you to manually register, unregister, and list the object implementations available on your VisiBroker system. The `oadutil` commands are implemented in Java and use a command line interface. Each command is accessed by invoking the `oadutil` command, passing the type of operation to be performed as the first argument.

Note An object activation daemon process (`oad`) must be started on at least one host in your network before you can use the `oadutil` commands.

The `oadutil` command has the following syntax:

Syntax `oadutil {list|reg|unreg} [options]`

The options for this tool vary, depending on whether you specify `list`, `reg` or `unreg`.

Converting interface names to repository IDs

Interface names and repository IDs are both ways of representing the type of interface the activated object should implement. All interfaces defined in IDL are assigned a unique repository identifier. This string is used to identify a type when communicating with the Interface Repository, the OAD, and most calls to the ORB itself.

When registering or unregistering an object with the OAD, the `oadutil` commands allow you to specify either an object's IDL interface name or its repository id.

An interface name is converted to a repository ID as follows:

- 1 Prepend "IDL:" to the interface name.
- 2 Replace all non-leading instances of the scope resolution operator (`::`) with a slash (`/`) character.
- 3 Append `":1.0"` to the interface name.

For example, the IDL interface name

```
::Module1::Module2::IntfName
```

would be converted to the following repository ID

```
IDL:Module1/Module2/IntfName:1.0
```

The `#pragma id` and `#pragma prefix` mechanisms can be used to override the default generation of repository id's from interface names. If the `#pragma id` mechanism is used in user-defined IDL files to specify non-standard repository IDs, the conversion process outlined above will not work. In these cases, you must use `-r repository id` argument and specify the object's repository ID.

To obtain the repository id of the object implementation's most derived interface, use the method `java: <interfaceName>Helper.id()` defined for all CORBA objects.

Listing objects with oadutil list

The `oadutil list` command returns all ORB object implementations registered with the Object Activation Daemon. Each OAD has its own implementation repository database where the registration information is stored.

Note An object activation daemon process (`oad`) must be started on at least one host in your network before you can use the `oadutil list` command.

The `oadutil list` command has the following syntax:

Syntax `oadutil list [options]`

The `oadutil list` command accepts the following command line arguments:

Option	Description
<code>-i <interface name></code>	Lists the implementation information for objects of a particular IDL interface name. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> . Note: All communications with the ORB reference an object's repository id instead of the interface name. For more information about the conversion performed when specifying an interface name, see "Converting interface names to repository IDs" on page 20-4.
<code>-r <repository id></code>	Lists the implementation information of a specific repository id. See "Converting interface names to repository IDs" on page 20-4 for details on specifying repository IDs. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> .
<code>-s <service name></code>	Lists the implementation information for a specific service name. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> .
<code>-poa <poa_name></code>	Lists the implementation information for a specific POA name. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> .
<code>-o <object name></code>	Lists the implementation information for a specific object name. You can use this only if the interface or repository id is specified in the command statement. This option is not applicable when an <code>-s</code> or <code>-poa</code> arguments is used.
<code>-h <OAD host name></code>	Lists the implementation information for objects registered with an OAD running on a specific remote host.
<code>-verbose</code>	Turns verbose mode on, causing messages to be output to stdout.
<code>-version</code>	Prints the version of this tool.
<code>-full</code>	Lists the status of all implementations registered with the OAD.

Description

The `oadutil list` utility allows you to list all ORB object implementations registered with the Object Activation Daemon. The information for each object includes:

- Interface names of the ORB objects.
- Instance names of the object offered by that implementation.
- Full path name of the server implementation's executable.
- Activation policy of the ORB object (shared or unshared).
- Reference data specified when the implementation was registered with the OAD.
- List of arguments to be passed to the server at activation time.
- List of environment variables to be passed to the server at activation time.

The following is an example of a local list request, specifying an interface name and object name:

Example `oadutil list -i Bank::AccountManager -o InpriseBank`

The following is an example of a remote list request, specifying a host IP address:

Example `oadutil list -h 206.64.15.198`

Registering objects with oadutil

The `oadutil` command can be used to register an object implementation from the command line or from within a script. The parameters are either the interface name and object name, the service name, or the POA name, and path name to the executable that starts the implementation. If the activation policy is not specified, the shared server policy will be used by default. You may write an implementation and start it manually during the development and testing phases. When your implementation is ready to be deployed, you can simply use `oadutil` to register your implementation with the OAD.

Note When registering an object implementation, use the same object name that is used when the implementation object is constructed. Only named objects (those with a global scope) may be registered with the OAD.

The `oadutil reg` command has the following syntax:

Syntax `oadutil reg [options]`

Note An `oad` process must be started on at least one host in your network before you can use the `oadutil reg` command.

The options for the `oadutil reg` command accepts the following command-line arguments:

Option	Description
Required	
<code>-i <interface name></code>	Specifies a particular IDL interface name. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> . See “Converting interface names to repository IDs” on page 20-4 for details on specifying repository IDs.
<code>-r <repository id></code>	Specifies a particular repository id. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> .
<code>-s <service name></code>	Specifies a particular service name. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> .
<code>-poa <poa_name></code>	Use this option to register the POA instead of an object implementation. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> .
<code>-o <object name></code>	Specifies a particular object. You can use this only if the interface name or repository id is specified in the command statement. This option is not applicable when an <code>-s</code> or <code>-poa</code> argument is used.
<code>-cpp <file name to execute></code>	Specifies the full path of an executable file that must create and register an object that matches the <code>-o/-r/-s/-poa</code> arguments. Applications registered with the <code>-cpp</code> argument must be stand-alone executables.
<code>-java <full class name></code>	Specifies the full name of a Java class containing a main routine. This application must create and register an Object that matches the <code>-o/-r/-s/-poa</code> argument. Classes registered with the <code>-java</code> argument will be executed with the command <code>vbj <full_classname></code> .
Optional	
<code>-host <OAD host name></code>	Specifies a specific remote host where the OAD is running.
<code>-verbose</code>	Turns verbose mode on, causing messages to be output to stdout.
<code>-version</code>	Prints the version of this tool.
<code>-d <referenceData></code>	Specifies reference data to be passed to the server upon activation.
<code>-a arg1</code> <code>-a arg2</code>	Specifies the arguments to be passed to the spawned executable as command-line arguments. Arguments can be passed with multiple <code>-a (arg)</code> parameters. They will be propagated in order to create the spawned executable.
<code>-e env1</code> <code>-e env2</code>	Specifies environment variables to be passed to the spawned executable. Arguments can be passed with multiple <code>-e (env)</code> parameters. They will be propagated in order to create the spawned executable.
<code>-p {shared unshared}</code>	Specifies the activation policy of the spawned objects. The default policy is <code>SHARED_SERVER</code> . Shared: Multiple clients of a given object share the same implementation. Only one server is activated by an OAD at a particular time. Unshared: Only one client of a given implementation will bind to the activated server. If multiple clients wish to bind to the same object implementation, a separate server is activated for each client application. A server exits when its client application disconnects or exits.

Example 1: Specifying repository ID

The following command will register with the OAD the VisiBroker program factory. It will be activated upon request for objects of repository ID `IDL:ehTest/Factory:1.0` (which corresponds to the interface name `ehTest::Factory`). The instance name of the object to be activated is `ReentrantServer`, and that name is also passed to the spawned executable as a command-line argument. This server has the unshared policy, by which it will be terminated when the requesting client breaks its connection to the spawned server.

Example `prompt> oadutil reg -r IDL:ehTest/Factory:1.0 -o ReentrantServer \
 -java factory_r -a ReentrantServer -p unshared`

Note In the example above, the specified Java class must be found in the CLASSPATH.

Example 2: Specifying IDL interface name

The following command will register with the OAD, the VisiBroker class `Server`. In this example, the specified class must activate an object of repository ID `IDL:Bank/AccountManager:1.0` (corresponding to the interface name `IDL` name `Bank::AccountManager`) and instance name `CreditUnion`. The server will be started with unshared policy, ensuring that it will terminate when the requesting client breaks its connection.

Example `prompt> oadutil reg -i Bank::AccountManager -o CreditUnion \
 -java Server -a CreditUnion -p unshared -e DEBUG=1`

Note In the previous example, the specified Java class must be found in the CLASSPATH.

The previous registration tells the OAD to execute the following command when spawning the requested server:

```
vbj -DDEBUG=1 Server CreditUnion
```

Remote registration to an OAD

To register an implementation with an OAD on a remote host, use the `-h` argument to `oadutil reg`.

The following is an example of how to perform a remote registration to an OAD on Windows NT from a UNIX shell. The double backslashes are necessary to avoid having the shell interpret the backslashes before passing them to `oadutil`.

Example `prompt> oadutil reg -r IDL:Library:1.0 Harvard \
 -cpp c:\\vbroker\\examples\\library\\libsrv.exe -p shared -h 100.64.15.198`

Accessing a server without using the Smart Agent

When a client needs to access a server via the OAD and is not using the Smart Agent, use `oadutil` to set the property `vbroker.orb.activationIOR` to an IOR, or to the location of an `.ior` file. For example, if the `server.ior` file is located in the `/home/username` directory, the command would be

```
oadutil -VBJprop vbroker.orb.activationIOR = file:///home/username/server.ior
```

For additional information, see Chapter 18, "Using the Naming Service."

Distinguishing between multiple instances of an object

Your implementation can use `ReferenceData` to distinguish between multiple instances of the same object. The value of the reference data is chosen by the implementation at object creation time and remains constant during the lifetime of the object. The `ReferenceData` typedef is portable across platforms and ORBs.

Setting activation properties using the `CreationImplDef` class

The `CreationImplDef` class contains the properties the OAD requires to activate an ORB object—`path_name`, `activation_policy`, `args`, and `env`. IDL sample 20.1 shows the `CreationImplDef` struct.

The `path_name` property specifies the exact path name of the executable program that implements the object. The `activation_policy` property represents the server's activation policy, discussed in “`CreationImplDef` interface” on page 20-11. The `args` and `env` properties represent command line arguments and environment settings for the server.

IDL sample 20.1 `CreationImplDef` IDL

```
module extension {
...

    enum Policy {
        SHARED_SERVER,
        UNSHARED_SERVER
    };

    struct CreationImplDef {
        CORBA::RepositoryId repository_id;
        string                object_name;
        CORBA::ReferenceData  id;
        string                path_name;
        Policy                activation_policy;
        CORBA::StringSequence args;
        CORBA::StringSequence env;
    };
...
};
```

Dynamically changing an ORB implementation

IDL sample 20.2 shows the `change_implementation()` member function which can be used to dynamically change an object's registration. You can use this member function to change the object's activation policy, path name, arguments, and environment variables.

IDL sample 20.2 `change_implementation`

```
module Activation
{
    ...
    void change_implementation(in extension::CreationImplDef old_info,
                              in extension::CreationImplDef new_info)
        raises ( NotRegistered, InvalidPath, IsActive );
    ...
};
```

Caution Although you can change an object's implementation name and object name with the `change_implementation()` member function, you should exercise caution. Doing so will prevent client programs from locating the object with the old name.

OAD Registration using `OAD::reg_implementation`

Instead of using the `oadutil reg` command manually or in a script, VisiBroker allows client applications to use the `OAD::reg_implementation` operation to register one or more objects with the activation daemon. Using this operation results in an object implementation being registered with the OAD and the `osagent`. The OAD will store the information in the implementation repository, allowing the object implementation to be located and activated when a client attempts to bind to the object.

IDL sample 20.3 `OAD::reg_implementation` operation

```
module Activation {
    ...
    typedef sequence<ObjectStatus> ObjectStatus List;
    ...
    typedef sequence<ImplementationStatus> ImplStatusList;
    ...
    interface OAD {
        // Register an implementation.
        Object reg_implementation(in extension::CreationImplDef impl)
            raises (DuplicateEntry, InvalidPath);
    }
}
```

The `CreationImplDef` struct contains the properties the OAD requires. The properties are `repository_id`, `object_name`, `id`, `path_name`, `activation_policy`, `args`, and `env`. Operations for setting and querying their values are also provided. These additional properties are used by the OAD to activate an ORB object.

IDL sample 20.4 CreationImplDef interface

```

struct CreationImplDef {
    CORBA::RepositoryId repository_id;
    string object_name;
    CORBA::ReferenceData id;
    string path_name;
    Policy activation_policy;
    CORBA::StringSequence args;
    CORBA::StringSequence env;
};

```

The `path_name` property specifies the exact path name of the executable program that implements the object. The `activation_policy` property represents the server's activation policy. The `args` and `env` properties represent optional arguments and environment settings to be passed to the server.

Example of object creation and registration

Code sample 20.1 shows how to use the `CreationImplDef` class and the `OAD.reg_implementation()` member function to register a server with the OAD. This mechanism may be used in a separate, administrative program, not necessarily in the object implementation itself. If used in the object implementation, these tasks must be performed prior to activating the object implementation.

Code sample 20.1 Creating an ORB object and registering with the OAD

```

// Register.java
import com.inprise.vbroker.Activation.*;
import com.inprise.vbroker.extension.*;

public class {

    public static void main(String[] args) {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // Locate an OAD

        try {
            OAD anOAD =
                OADHelper.bind(orb);

            // Create an ImplDef
            CreationImplDef _implDef = new com.inprise.vbroker.extension.CreationImplDef();
            _implDef.repository_id = "IDL:Bank/AccountManager:1.0";
            _implDef.object_name = "BankManager";
            _implDef.path_name = "vbj";
            _implDef.id = new byte[0];
            _implDef.activation_policy = com.inprise.vbroker.extension.Policy.SHARED_SERVER;
            _implDef.env = new String[0];

            String[] str = new String[1];
            str[0] = "Server";
            _implDef.args = str;

```

```
        try {
            anOAD.reg_implementation(_implDef);
        } catch (Exception e) {
            System.out.println("Caught " + e);
        }

    }

    catch (org.omg.CORBA.NO_IMPLEMENT e) {
    }
}
}
```

Arguments passed by the OAD

When the OAD starts an object implementation it passes all of the arguments that were specified when the implementation was registered with the OAD.

Un-registering objects

When the services offered by an object are no longer available or temporarily suspended, the object should be unregistered with the OAD. When an ORB object is unregistered, it is removed from the implementation repository. The object is also removed from the Smart Agent's dictionary. Once an object is unregistered, client programs will no longer be able to locate or use it. In addition, you will be unable to use the `OAD.change_implementation()` member function to change the object's implementation. As with the registration process, un-registering may be done either at the command line or programmatically. There is also an ORB object interface to the OAD, described in "Un-registering objects" on page 20-12.

Un-registering objects using the oadutil tool

The `oadutil unreg` command allows you to unregister one or more object implementations registered with the Object Activation Daemon. Once an object is unregistered, it can no longer be automatically activated by the OAD if a client requests the object. Only objects that have been previously registered via the `oadutil reg` command may be unregistered with `oadutil unreg`.

If you specify only an interface name, all ORB objects associated with that interface will be unregistered. Alternatively, you may identify a specific ORB object by its interface name and object name. When you unregister an object, all processes associated with that object will be terminated.

Note An `oad` process must be started on at least one host in your network before you can use the `oadutil reg` command.

The `oadutil unreg` command has the following syntax:

Syntax `oadutil unreg [options]`

The options for the `oadutil unreg` command accepts the following command line arguments:

Option	Description
Required	
<code>-i <interface name></code>	Specifies a particular IDL interface name. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> . See “Converting interface names to repository IDs” on page 20-4 for details on specifying repository IDs.
<code>-r <repository id></code>	Specifies a particular repository id. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> .
<code>-s <service name></code>	Specifies a particular service name. Only one of the following options may be specified at a particular time: <code>-i</code> , <code>-r</code> , <code>-s</code> , or <code>-poa</code> .
<code>-o <object name></code>	Specifies a particular object name. You can use this only if the interface name or repository id is included in the command statement. This option is not applicable when a <code>-s</code> or <code>-poa</code> argument is used.
<code>-poa <POA_name></code>	Unregisters the POA registered using <code>oadutil reg -poa <POA_name></code> .
Optional	
<code>-host <host name></code>	Specifies the host name where the OAD is running.
<code>-verbose</code>	Enables verbose mode, causing messages to be output to stdout.
<code>-version</code>	Prints the version of this tool.

Unregistration example

The `oadutil unreg` utility unregisters one or more ORB objects from these three locations:

- Object Activation Daemon
- Implementation repository
- Smart Agent

The following is an example of how to use the `oadutil unreg` command. It unregisters the implementation of the `Bank::AccountManager` named `InpriseBank` from the local OAD.

Example `oadutil unreg -i Bank::AccountManager -o InpriseBank`

Un-registering with the OAD operations

An object’s implementation can use any one of the operations or attributes in the OAD interface to un-register an ORB object.

- `unreg_implementation(in CORBA::RepositoryId repId, in string object_name)`
- `unreg_interface(in CORBA::RepositoryId repId)`
- `unregister_all()`

- attribute `boolean destroy_on_unregister`
- | | |
|-------------------------------------|---|
| <code>unreg_implementation()</code> | Use this operation when you want to un-registered implementations using a specific repository id and object name. This operation terminates all processes currently implementing the specified repository id and object name. |
| <code>unreg_interface()</code> | Use this operation when you want to un-registered implementations by using a specific repository id only. This operation terminates all processes currently implementing the specified repository id. |
| <code>unregister_all()</code> | Use this operation to un-registered all implementations. Unless <code>destroyActive</code> is set to true, all active implementations continue to execute. For backward compatibility, <code>unregister_all()</code> is <i>not</i> destructive; it is equivalent to invoking <code>unregister_all_destroy(false)</code> . |
| <code>destroy_on_unregister</code> | Use this attribute to destroy any spawned processes on unregistration of the relevant implementation. The default value is false. |

IDL sample 20.5 OAD un-registered operation

```

module Activation {
    ...
    interface OAD {
        ...
        void unreg_implementation(in CORBA::RepositoryId repId, in string
                                object_name)
        raises(NotRegistered);
        ...
    }
}

```

Displaying the contents of the implementation repository

You can use the `oadutil` tool to list the contents of a particular implementation repository. For each implementation in the repository the `oadutil` tool lists all the object instance names, the path name of the executable program, the activation mode and the reference data. Any arguments or environment variables that are to be passed to the executable program are also listed.

IDL interface to the OAD

The OAD is implemented as an ORB object, allowing you to create a client program that binds to the OAD and uses its interface to query the status of objects that have been registered. IDL sample 20.6 shows the IDL interface specification for the OAD.

IDL sample 20.6 OAD interface specification

```

module Activation
{
    enum state {
        ACTIVE,
        INACTIVE,
        WAITING_FOR_ACTIVATION
    };
    struct ObjectStatus {
        long unique_id;
        State activation_state;
        Object objRef;
    };
    typedef sequence<ObjectStatus> ObjectStatusList;
    struct ImplementationStatus {
        extension::CreationImplDef impl;
        ObjectStatusList status;
    };
    typedef sequence<ImplementationStatus> ImplStatusList;
}

```

```

exception DuplicateEntry {};
exception InvalidPath {};
exception NotRegistered {};
exception FailedToExecute {};
exception NotResponding {};
exception IsActive {};
exception Busy {};

interface OAD {
    Object reg_implementation(in extension::CreationImplDef impl)
        raises (DuplicateEntry, InvalidPath);
    extension::CreationImplDef get_implementation(
        in CORBA::RepositoryId repId,
        in string object_name)
        raises ( NotRegistered);
    void change_implementation(in extension::CreationImplDef old_info,
        in extension::CreationImplDef new_info)
        raises (NotRegistered,InvalidPath,IsActive);
    attribute boolean destroy_on_unregister;
    void unreg_implementation(in CORBA::RepositoryId repId,
        in string object_name)
        raises ( NotRegistered );
    void unreg_interface(in CORBA::RepositoryId repId)
        raises ( NotRegistered );
    void unregister_all();
    ImplementationStatus get_status(in CORBA::RepositoryId repId,
        in string object_name)
        raises ( NotRegistered);
    ImplStatusList get_status_interface(in CORBA::RepositoryId repId)
        raises (NotRegistered);
    ImplStatusList get_status_all();
};

```

Using interface repositories

An interface repository (IR) contains descriptions of CORBA object interfaces. The data in an IR is the same as in IDL files—descriptions of modules, interfaces, operations, and parameters—but it is organized for runtime access by clients. A client can browse an interface repository (perhaps serving as an online reference tool for developers) or can look up the interface of any object for which it has a reference (perhaps in preparation for invoking the object with the Dynamic Invocation Interface).

Reading this chapter will enable you to create an interface repository and access it with VisiBroker utilities or with your own code.

What is an interface repository?

An interface repository (IR) is like a database of CORBA object interface information that enables clients to learn about or update interface descriptions at runtime. In contrast to the VisiBroker Location Service, described in Chapter 17, “Using the Location Service,” which holds data describing object *instances*, an IR’s data describes *interfaces* (types). There may or may not be available instances that satisfy the interfaces stored in an IR. The information in an IR is equivalent to the information in an IDL file (or files), but it is represented in a way that is easier for clients to use at runtime.

Clients that use interface repositories may also use the Dynamic Invocation Interface (DII) described in Chapter 22, “Using the Dynamic Invocation Interface.” Such clients use an interface repository to learn about an unknown object’s interface, and they use the DII to invoke methods on the object. However, there is no necessary connection between an IR and the DII. For example, someone could use the IR to write an “IDL browser” tool for developers—in such a tool, dragging a method description from the browser to an editor would insert a template method invocation into the developer’s source code. In this example, the IR is used without the DII.

You create an interface repository with the VisiBroker `irep` program, which is the IR server (implementation). You can update or populate an interface repository with the VisiBroker `idl2ir` program, or you can write your own IR client that inspects an interface repository, updates it, or does both.

What does an interface repository contain?

An interface repository contains hierarchies of objects whose methods divulge information about interfaces. Although interfaces are usually thought of as describing objects, using a collection of objects to describe interfaces makes sense in a CORBA environment because it requires no new mechanism such as a database.

As an example of the kinds of objects an IR can contain, consider that IDL files can contain IDL module definitions, and modules can contain interface definitions, and interfaces can contain operation (method) definitions. Correspondingly, an interface repository can contain `ModuleDef` objects which can contain `InterfaceDef` objects, which can contain `OperationDef` objects. Thus, from an IR `ModuleDef`, you can learn what `InterfaceDefs` it contains. The reverse is also true—given an `InterfaceDef` you can learn what `ModuleDef` it is contained in. All other IDL constructs—including exceptions, attributes, and valuetypes—can be represented in an interface repository.

An interface repository also contains typecodes. Typecodes are not explicitly listed in IDL files, but are automatically derived from the types (`long`, `string`, `struct`, and so on) that are defined or mentioned in IDL files. Typecodes are used to encode and decode instances of the CORBA `any` type—a generic type that stands for any type and is used with the dynamic invocation interface.

How many interface repositories can you have?

Interface repositories are like other objects—you can create as many as you like. There is no VisiBroker-mandated policy governing the creation or use of IRs. You determine how interface repositories are deployed and named at your site. You may, for example, adopt the convention that a central interface repository contains the interfaces of all “production” objects, and developers create their own IRs for testing.

Note Interface repositories are writable and are not protected by access controls. An erroneous or malicious client can corrupt an IR or obtain sensitive information from it.

If you want to use the `_get_interface_def` method defined for all objects, you must have at least one interface repository server running so the ORB can look up the interface in the IR. If no interface repository is available, or if the IR that the ORB binds to has not been loaded with an interface definition for the object, `_get_interface_def` raises a `NO_IMPLEMENT` exception.

Creating and viewing an interface repository with irep

The VisiBroker interface repository server is called `irep`, and is located in the `bin` directory. The `irep` program runs as a daemon. You can register `irep` with the Object Activation Daemon as you would any object implementation. The `oadutil` tool requires the object ID—for example, `IDL:org.omg/CORBA/Repository:2.3` (as opposed to an interface name such as `CORBA::Repository`).

Creating an interface repository with irep

Use the `irep` program to create an interface repository and view its contents. The usage syntax for the `irep` program is as follows:

Syntax `irep <driverOptions> <otherOptions> IRepName [file.idl]`

The syntax for creating an interface repository in the `irep` is described in the following table:

Syntax	Description
<code>IRepName</code>	Specifies the instance name of the interface repository. Clients can bind to this interface repository instance by specifying this name.
<code>file.idl</code>	Specifies the IDL file whose contents <code>irep</code> will load into the interface repository it creates and will store the IR contents into when it exits. If no file is specified, <code>irep</code> creates an empty interface repository.

The `irep` arguments are defined in the following table.

Argument	Description
Driver options	
<code>-J<java option></code>	Pass the option to JVM directly.
<code>-VBJversion</code>	Print VBJ version
<code>-VBJdebug</code>	Print VBJ debug information.
<code>-VBJclasspath</code>	Specify classpath, precedes CLASSPATH env variable.
<code>-VBJprop <name>[=<value>]</code>	Pass name/value pair to JVM.
<code>-VBJjavavm <jvmpath></code>	Specify JVM path.
<code>-VBJaddJar <jarfile></code>	Append jarfile to the CLASSPATH before execing the JVM.

Argument	Description
Other options	
-D, -define foo[=bar]	Define a preprocessor macro, optionally with value.
-I, -include <dir>	Specify additional directory for #include searching.
-P, -no_line_directives	Do not emit #line directives from preprocessor. The default is off.
-H, -list_includes	Display #included file names as they are encountered. The default is off.
-C, -retain_comments	Retain comments in preprocessed output. The default is off.
-U, -undefine foo	Undefine a preprocessor macro.
-[no_]idl_strict	Strict OMG-standard interpretation of IDL source. The default is off.
-[no_]warn_unrecognized_pragmas	Warn if a #pragma is not recognized. The default is on.
-[no_]back_compat_mapping	Use mapping that is compatible with VisiBroker 3.x.
-h, -help, -usage, -?	Print this usage information.
-version	Display software version numbers.
-install <service name>	Install as a NT service.
-remove <service name>	Deinstall this NT service.

The following example shows how an interface repository named `TestIR` can be created from a file called `Bank.idl`.

Example `irep TestIR Bank.idl`

Viewing the contents of the interface repository

You can view the contents of the interface repository with either the VisiBroker `ir2idl` utility, or the VisiBroker Console application. The syntax for the `ir2idl` utility is:

Syntax `ir2idl [-irep IRname]`

The syntax for viewing the contents of an interface repository in the `irep` is described in the following table:

Syntax	Description
-irep <i>IRname</i>	Directs the program to bind to the interface repository instance named <i>IRname</i> . If the option is not specified, it binds to any interface repository returned by the Smart Agent.

For more details on the `ir2idl` utility arguments see Chapter 2, “Programmer tools,” of the VisiBroker for Java *Reference*.

Updating an interface repository with idl2ir

You can update an interface repository with the VisiBroker `idl2ir` utility, which is an IR client. The syntax for the `idl2ir` utility is:

Syntax `idl2ir [arguments] idl_file_list`

For more details on the `idl2ir` utility arguments see Chapter 2, “Programmer tools,” of the VisiBroker for Java *Reference*.

The following example shows how the `TestIR` interface repository would be updated with definitions from the `Bank.idl` file.

Example `idl2ir -irep TestIR -replace Bank.idl`

Entries in an interface repository cannot be removed using the `idl2ir` or `irep` utilities. To remove an item,

- Exit or quit the `irep` program.
- Edit the IDL file named in the `irep` command line.
- Start `irep` again with the updated file.

Interface repositories have a simple transaction service. If the specified IDL file fails to load, the interface repository rolls back its content to its previous state. After loading the IDL, the interface repository commits its state to be used in subsequent transactions. For any repository, there is a file `IRname.rollback` in the home directory that contains the state of the last uncommitted transaction.

Note If you wish to remove all entries in the Interface Repository, you can replace the contents with a new empty IDL file. For example, using an IDL file named `Empty.idl`, you could run the following command:

```
idl2ir -irep TestIR -replace Empty.idl
```

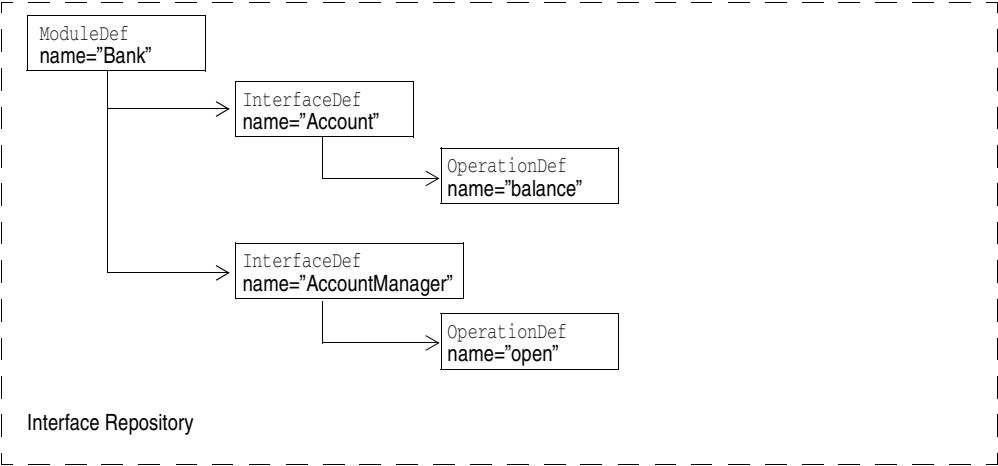
Understanding the structure of the interface repository

An interface repository organizes the objects it contains into a hierarchy that corresponds to the way interfaces are defined in an IDL specification. Some objects in the interface repository contain other objects, just as an IDL module definition might contain several interface definitions. Consider how the example IDL file shown in IDL sample 21.1 would translate to a hierarchy of objects in an interface repository.

IDL sample 21.1 `Bank.idl` file

```
// Bank.idl
module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

Figure 21.1 Interface repository object hierarchy for Bank.idl



OperationDef object contains references to additional data structures (not interfaces) that hold the parameters and return type.

Identifying objects in the interface repository

The following table shows the objects that are provided to identify and classify interface repository objects.

Table 21.1 Objects used to identify and classify interface repository objects

Item	Description
name	A character string that corresponds to the identifier assigned in an IDL specification to a module, interface, operation, and so forth. An identifier is not necessarily unique.
id	A character string that uniquely identifies an <code>IRObject</code> . A <code>RepositoryID</code> contains three components, separated by colon (:) delimiters. The first component is “IDL:” and the last is a version number such as “:1.0”. The second component is a sequence of identifiers separated by slash (/) characters. The first identifier is typically a unique prefix.
def_kind	An enumeration that defines values which represent all the possible types of interface repository objects.

Types of objects that can be stored in the interface repository

Table 21.2 summarizes the objects that can be contained in an interface repository. Most of these objects correspond to IDL syntax elements. A `StructDef`, for example, contains the same information as an IDL struct declaration, an `InterfaceDef` contains the same information as an IDL interface declaration, all the way down to a `PrimitiveDef` which contains the same information as an IDL primitive (boolean, long, and so forth.) declaration.

Table 21.2 Objects that can be stored in the interface repository

Object type	Description
Repository	Represents the top-level module that contains all other objects.
ModuleDef	Represents an IDL module declaration that can contain <code>ModuleDefs</code> , <code>InterfaceDefs</code> , <code>ConstantDefs</code> , <code>AliasDefs</code> , <code>ExceptionDefs</code> , and the IR equivalents of other IDL constructs that can be defined in IDL modules.
InterfaceDef	Represents an IDL interface declaration and contain <code>OperationDefs</code> , <code>ExceptionDefs</code> , <code>AliasDefs</code> , <code>ConstantDefs</code> , and <code>AttributeDefs</code> .
AttributeDef	Represents an IDL attribute declaration.
OperationDef	Represents an IDL operation (method) declaration. Defines an operation on an interface. It includes a list of parameters required for this operation, the return value, a list of exceptions that may be raised by this operation, and a list of contexts.
ConstantDef	Represents an IDL constant declaration.
ExceptionDef	Represents an IDL exception declaration.
ValueDef	Represents a valuetype definition containing lists of constants, types, valuemembers, exceptions, operations, and attributes.
ValueBoxDef	Represents a simple boxed valuetype of another IDL type.
ValueMemberDef	Represents a member of the valuetype.
NativeDef	Represents a native definition. Users can not define their own natives.
StructDef	Represents an IDL structure declaration.
UnionDef	Represents an IDL union declaration.
EnumDef	Represents an IDL enumeration declaration.
AliasDef	Represents an IDL typedef declaration. Note that the IR <code>TypedefDef</code> interface is a base interface that defines common operations for <code>StructDefs</code> , <code>UnionDefs</code> , and others.
StringDef	Represents an IDL bounded string declaration.
SequenceDef	Represents an IDL sequence declaration.
ArrayDef	Represents an IDL array declaration.
PrimitiveDef	Represents an IDL primitive declaration: null, void, long, ushort, ulong, float, double, boolean, char, octet, any, <code>TypeCode</code> , <code>Principal</code> , string, objref, longlong, ulonglong, longdouble, wchar, wstring.

Inherited interfaces

Three non-instantiatable (that is, abstract) IDL interfaces define common methods that are inherited by many of the objects contained in an IR (see Table 21.2). Table 21.3 summarizes these widely inherited interfaces. For more information on the other methods for these interfaces, see Chapter 7, “Interface repository interfaces and classes,” in *VisiBroker for Java Reference*.

Table 21.3 Interfaces inherited by many IR objects

Interface	Inherited by	Principal query methods
IRObject	All IR objects including Repository	<code>def_kind()</code> —Returns an IR object’s definition kind, for example, module or interface <code>destroy()</code> —Destroys an IR object
Container	IR objects that can contain other IR objects, for example, module or interface	<code>lookup()</code> —Looks up a contained object by name <code>contents()</code> —Lists the objects in a Container <code>describe_contents()</code> —Describes the objects in a Container
Contained	IR objects that can be contained in other objects, that is, Containers	<code>name()</code> —Name of this object <code>defined_in()</code> —Container that contains an object <code>describe()</code> —Describe an object <code>move ()</code> —Moves an object into another container.

Accessing an interface repository

Your client program can use an interface repository’s IDL interface to obtain information about the objects it contains. Your client program can bind to the `Repository` and then invoke the methods shown in Code sample 21.1. A complete description of this interface can be found in the *VisiBroker for Java Reference*.

Code sample 21.1 Repository interface

```
package org.omg.CORBA;
public interface Repository extends Container {
    . . .
    org.omg.CORBA.Contained lookup_id(string id);
    org.omg.CORBA.PrimitiveDef get_primitive(org.omg.CORBA.PrimitiveKind kind);
    org.omg.CORBA.StringDef create_string(long bound);
    org.omg.CORBA.SequenceDef create_sequence(long bound,
        org.omg.CORBA.IDLType element_type);
    org.omg.CORBA.ArrayDef create_array(long length,
        org.omg.CORBA.IDLType element_type);
    . . .
}
```

Example programs

This section describes a simple Interface Repository example which contains a simple `AccountManager` interface to create an account and (re)open an account. The code is in the `examples\ir` directory. At the initialization time the `AccountManager` implementation bootstraps the Interface Repository definition for the managed `Account` interface. This exposes the additional operation that has been already implemented by this particular `Account` implementation to the clients. The clients now can access all known operations (which are described in IDL) and, additionally, they can verify with the Interface Repository support for other operations and invoke them. The example illustrates how we can manage Interface Repository definition objects and how to introspect remote objects using the Interface Repository.

Before this program can be tested, the following conditions should exist:

- `OSAgent` should be up and running.
- Interface repository should be started using `irep`.
- Interface Repository should be loaded with an IDL file either by the command line when you start the Interface Repository, or by using `idl2ir`.
- Start the client program.

Code sample 21.2 Looking up an interface's operations and attributes in an IR

```
//Client.java
import org.omg.CORBA.InterfaceDef;
import org.omg.CORBA.InterfaceDefHelper;
import org.omg.CORBA.Request;
import java.util.Random;

public class Client {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // Get the manager Id
            byte[] managerId = "BankManager".getBytes();
            // Locate an account manager. Give the full POA name and the servant ID.
            Bank.AccountManager manager =
                Bank.AccountManagerHelper.bind(orb, "/bank_ir_poa", managerId);
            // use args[0] as the account name, or a default.
            String name = args.length > 0 ? args[0] : "Jack B. Quick";
            // Request the account manager to open a named account.
            Bank.Account account = manager.open(name);
            // Get the balance of the account.
            float balance = account.balance();
            // Print out the balance.
            System.out.println("The balance in " + name + "'s account is $" + balance);
            // Calculate and set a new balance
            balance = args.length > 1 ? Float.parseFloat(args[1]) :
                Math.abs(new Random().nextInt()) % 100000 / 100f;
            account.balance(balance);
        }
    }
}
```

Example programs

```
        // Get the balance description if it is possible and print it
        String desc = getDescription(account);
        System.out.println("Balance description:\n" + desc);
    } catch (org.omg.CORBA.SystemException e) {
        System.err.println("System exception caught:" + e);
    } catch (Exception e) {
        System.err.println("Unexpected exception caught:");
        e.printStackTrace();
    }
}

static String getDescription (Bank.Account account) {
    // Get the interface repository definition for this interface
    InterfaceDef accountDef = InterfaceDefHelper.narrow(account._get_interface_def());
    // Check if this *particular* implementation supports "describe" operation
    if (accountDef.lookup("describe") != null) {
        // We cannot use the static skeleton's method here because at the
        // time of its creation this method was not present in the IDL's
        // version of the Account interface. Use DII instead.
        Request request = account._request("describe");
        request.result().value().insert_string("");
        request.invoke();
        return request.result().value().extract_string();
    } else {
        return "<no description>";
    }
}
}
```


Advanced concepts

This part of the VisiBroker for Java *Programmer's Guide* includes these chapters.

- Chapter 22 "Using the Dynamic Invocation Interface"
- Chapter 23 "Using the Dynamic Skeleton Interface"
- Chapter 24 "Using interceptors"
- Chapter 25 "Using object wrappers"
- Chapter 26 "Using RMI over IIOP"
- Chapter 27 "Using the dynamically managed types"
- Chapter 28 "Using valuetypes"
- Chapter 29 "Using URL naming"

Using the Dynamic Invocation Interface

The developers of most client programs know the types of the CORBA objects their code will invoke, and they include the compiler-generated stubs for these types in their code. By contrast, developers of generic clients cannot know what kinds of objects their users will want to invoke. Such developers use the Dynamic Invocation Interface (DII) to write clients that can invoke any method on any CORBA object from knowledge obtained at runtime.

What is the Dynamic Invocation Interface?

The Dynamic Invocation Interface (DII) enables a client program to invoke a method on a CORBA object whose type was unknown at the time the client was written. The DII contrasts with the default static invocation, which requires that the client source code include a compiler-generated stub for each type of CORBA object that the client intends to invoke. In other words, a client that uses static invocation declares in advance the types of objects it will invoke. A client that uses the DII makes no such declaration because its programmer doesn't know what kinds of objects will be invoked. The advantage of the DII is flexibility—it can be used to write generic clients that can invoke any object, including objects whose interfaces did not exist when the client was compiled. The DII has two disadvantages:

- It is more difficult to program (in essence, your code must do the work of a stub).
- Invocations take longer because more work is done at runtime.

The DII is purely a client interface—static and dynamic invocations are identical from an object implementation's point of view.

You can use the DII to build clients like these:

- Bridges or adapters between script environments and CORBA objects. For example, a script calls your bridge, passing object and method identifiers and parameter values. Your bridge constructs and issues a dynamic request, receives the result, and returns it to the scripting environment. Such a bridge could not use static invocation because its developer could not know in advance what kinds of objects the script environment would want to invoke.
- Generic object testers. For example, a client takes an arbitrary object identifier, looks up its interface in the interface repository (see Chapter 21, “Using interface repositories”), and then invokes each of its methods with artificial argument values. Again, this style of generic tester could not be built with static invocation.

Note Clients *must* pass valid arguments in DII requests. Failure to do so can produce unpredictable results, including server crashes. Although it is possible to dynamically type-check parameter values with the interface repository, it is expensive. For best performance, ensure that the code (for example, script) that invokes a DII-using client can be trusted to pass valid arguments.

Introducing the main DII concepts

The dynamic invocation interface is actually distributed among a handful of CORBA interfaces. Furthermore, the DII frequently offers more than one way to accomplish a task—the trade-off being programming simplicity versus performance in special situations. As a result, DII is one of the more difficult CORBA facilities to grasp. This section is a starting point, a high-level description of the main ideas. Details, including code examples, are provided later in the chapter.

To use the DII you need to understand these concepts, starting from the most general:

- Request objects
- Any and Typecode objects
- Request sending options
- Reply receiving options

Using request objects

A `Request` object represents one invocation of one method on one CORBA object. If you want to invoke two methods on the same CORBA object, or the same method on two different objects, you need two `Request` objects. To invoke a method you first need an object reference representing the CORBA object—the target reference. Using the target reference, you create a `Request`, populate it with arguments, send the `Request`, wait for the reply, and obtain the result from the `Request`.

There are two ways to create a `Request`. The simpler way is to invoke the target object's `_request` method, which all CORBA objects inherit. This does not, in fact, invoke the target object. You pass `_request` the IDL name of the method you intend to invoke in the `Request`, for example, “`get_balance`.” To add argument values to a `Request` created with `_request`, you invoke the `Request`'s `add_value` method for each argument

required by the method you intend to invoke. To pass one or more `Context` objects to the target, you must add them to the `Request` with its `ctx` method.

Although not intuitively obvious, you must also specify the type of the `Request`'s result with its `result` method. For performance reasons, the messages exchanged between ORBs do not contain type information. By specifying a place holder result type in the `Request`, you give the ORB the information it needs to properly extract the result from the reply message sent by the target object. Similarly, if the method you are invoking can raise user exceptions, you must add place holder exceptions to the `Request` before sending it.

The more complicated way to create a `Request` object is to invoke the target object's `_create_request` method, which, again, all CORBA objects inherit. This method takes several arguments which populate the new `Request` with arguments and specify the types of the result and user exceptions, if any, that it may return. To use the `_create_request` method you must have already built the components that it takes as arguments. The potential advantage of the `_create_request` method is performance. You can reuse the argument components in multiple `_create_request` calls if you invoke the same method on multiple target objects.

Note There are two overloaded forms of the `_create_request` method—one that includes `ContextList` and `ExceptionList` parameters, and one that does not. If you want to pass one or more `Context` objects in your invocation, and/or the method you intend to invoke can raise one or more user exceptions, you must use the `_create_request` method that has the extra parameters.

Encapsulating arguments with the Any type

The target method's arguments, result, and exceptions are each specified in special objects called `Any`s. An `Any` is a generic object that encapsulates an argument of any type. An `Any` can hold any type that can be described in IDL. Specifying an argument to a `Request` as an `Any` allows a `Request` to hold arbitrary argument types and values without making the compiler complain of type mismatches. (The same is true of results and exceptions.)

An `Any` consists of a `TypeCode` and a value. A value is just a value, and a `TypeCode` is an object that describes how to interpret the bits in the value (that is, the value's type). Simple `TypeCode` constants for simple IDL types, such as `long` and `Object`, are built into the header files produced by the `idl2java` compiler. `TypeCodes` for IDL constructed types, such as `structs`, `unions`, and `typedefs`, have to be constructed. Such `TypeCodes` can be recursive because the types they describe can be recursive. Consider a `struct` consisting of a `long` and a `string`. The `TypeCode` for the `struct` contains a `TypeCode` for the `long` and a `TypeCode` for the `string`. You can get a `TypeCode` at runtime from an interface repository (see Chapter 21, "Using interface repositories") or by asking the ORB to create one by invoking `ORB::create_struct_tc` or `ORB::create_exception_tc`.

If you use the `_create_request` method, you need to put the `Any`-encapsulated target method arguments in another special object called an `NVList`. No matter how you create a `Request`, its result is encoded as an `NVList`. Everything said about arguments in this paragraph applies to results as well. `NV` stands for named value, and an `NVList` consists of a count and number of items, each of which has a name, a value, and a flag. The name is the argument name, the value is the `Any` encapsulating the

argument, and the flag denotes the argument's IDL mode (for example, in or out). The result of `Request` is represented a single named value.

Options for sending requests

Once you've created and populated a `Request` with arguments, a result type, and exception types, you send it to the target object. There are several ways to send a `Request`,

- The simplest is to call the `Request`'s `invoke` method, which blocks until the reply message is received.
- More complex, but not blocking, is the `Request`'s `send_deferred` method. This is an alternative to using threads for parallelism. For many operating systems the `send_deferred` method is more economical than spawning a thread.
- If your motivation for using the `send_deferred` method is to invoke multiple target objects in parallel, you can use the ORB object's `send_multiple_requests_deferred` method instead. It takes a sequence of `Request` objects.
- Use the `Request`'s `send_oneway` method if, and only if, the target method has been defined in IDL as `oneway`.
- You can invoke multiple `oneway` methods in parallel with the ORB's `send_multiple_requests_oneway` method.

Options for receiving replies

If you send a `Request` by calling its `invoke` method, there is only one way to get the result—use the `Request` object's `env` method to test for an exception, and if none, extract the `NamedValue` from the `Request` with its `result` method. If you used the `send_oneway` method then there is no result. If you used the `send_deferred` method, you can periodically check for completion by calling the `Request`'s `poll_response` method which returns a code indicating whether the reply has been received. If, after polling for a while, you want to block waiting for completion of a deferred send, use the `Request`'s `get_response` method.

If you have sent `Requests` with the `send_multiple_requests_deferred` method, you can find out if a particular `Request` is complete by invoking that `Request`'s `get_response` method. To learn when any outstanding `Request` is complete, use the ORB's `get_next_response` method. To do the same thing without risking blocking, use the ORB's `poll_next_response` method.

Steps for invoking object operations dynamically

To summarize, here are the steps that a client follows when using the DII,

- 1 Obtain a generic reference to the target object you wish to use.
- 2 Create a `Request` object for the target object.
- 3 Initialize the request parameters and the result to be returned.
- 4 Invoke the request and wait for the results.
- 5 Retrieve the results.

Location of example programs for using the DII

Several example programs that illustrate the use of the DII are included in the `examples/bank_dynamic` directory of the VisiBroker distribution. These example programs will be used to illustrate DII concepts in this chapter.

Using the `idl2java` compiler

The `idl2java` compiler has a flag (`-dynamic_marshal`) which, when switched on, generates stub code using DII. To understand how to do any type of DII: create an IDL file, generate with `-dynamic_marshal`, and look at the stub code.

Obtaining a generic object reference

When using the DII, a client program does not have to use the traditional bind mechanism to obtain a reference to the target object, because the class definition for the target object may not have been known to the client at compile time. Code sample 22.1 shows how your client program can use the `bind` method offered by the ORB object to bind to any object by specifying its name. This method returns a generic `org.omg.CORBA.Object`.

Code sample 22.1 Obtaining a generic object reference

```

. . .
org.omg.CORBA.Object account;
try {
    // initialize the ORB.
    org.omg.CORBA.ORB.init(args, null);
} catch(Exception e) {
    System.err.println ("Failure during ORB_init");
    e.printStackTrace();
}
. . .

try {
    // Request ORB to bind to the object supporting the account interface.
    account = orb.bind("IDL:Account:1.0");
} catch(const CORBA::Exception& excep) {
    System.err.println ("Error binding to account" );
    excep.printStackTrace();
}
System.out.println ("Bound to account object");
. . .

```

Creating and initializing a request

When your client program invokes a method on an object, a `Request` object is created to represent the method invocation. The `Request` object is written, or *marshalled*, to a buffer and sent to the object implementation. When your client program uses client stubs, this processing occurs transparently. Client programs that wish to use the DII must create and send the `Request` object themselves.

Note There is no constructor for this class. The `Object`'s `_request` method or `Object`'s `_create_request` method are used to create a `Request` object.

Request interface

The following code sample shows the `Request` interface. The target of the request is set implicitly from the object reference used to create the `Request`. The name of the operation must be specified when the `Request` is created.

Code sample 22.2 Request interface

```
package org.omg.CORBA;

public abstract class Request {
    public abstract org.omg.CORBA.Object target();
    public abstract java.lang.String operation();
    public abstract org.omg.CORBA.NVList arguments();
    public abstract org.omg.CORBA.NamedValue result();
    public abstract org.omg.CORBA.Environment env();
    public abstract org.omg.CORBA.ExceptionList exceptions();
    public abstract org.omg.CORBA.ContextList contexts();
    public abstract void ctx(org.omg.CORBA.Context ctx);
    public abstract org.omg.CORBA.Context ctx();
    public abstract org.omg.CORBA.Any add_in_arg();
    public abstract org.omg.CORBA.Any add_named_in_arg(
    public abstract org.omg.CORBA.Any add_inout_arg();
    public abstract org.omg.CORBA.Any add_named_inout_arg(
    public abstract org.omg.CORBA.Any add_out_arg();
    public abstract org.omg.CORBA.Any add_named_out_arg(
    public abstract void set_return_type(
    public abstract org.omg.CORBA.Any return_value();
    public abstract void invoke();
    public abstract void send_oneway();
    public abstract void send_deferred();
    public abstract void get_response();
    public abstract boolean poll_response();
}
```

Ways to create and initialize a DII request

Once you have issued a `bind` to an object and obtained an object reference, you can use one of two methods for creating a `Request` object. The following code sample shows the methods offered by the `org.omg.CORBA.Object` interface.

Code sample 22.3 Three methods for creating a Request object

```

package org.omg.CORBA;
public interface Object {
    . . .

    public org.omg.CORBA.Request _request(java.lang.String operation;

    public org.omg.CORBA.Request _create_request(
        org.omg.CORBA.Context ctx,
        java.lang.String operation,
        org.omg.CORBA.NVList arg_list,
        org.omg.CORBA.NamedValue result
    )

    public org.omg.CORBA.Request _create_request(
        org.omg.CORBA.Context ctx,
        java.lang.String operation,
        org.omg.CORBA.NVList arg_list,
        org.omg.CORBA.NamedValue result,
        org.omg.CORBA.ExceptionList exceptions,
        org.omg.CORBA.ContextList contexts
    )
    . . .
}

```

Using the create_request method

You can use the `_create_request` method to create a `Request` object, initialize the `Context`, the operation name, the argument list to be passed, and the result. Optionally, you can set the `ContextList` for the request, which corresponds to the attributes defined in the request's IDL. The request parameter points to the `Request` object that was created for this operation.

Using the _request method

Code sample 22.4 shows the use of the `_request` method to create a `Request` object, specifying only the operation name. After creating a float request, calls to its `add_in_arg` method add an input parameter Account name and its result type is initialized to be of `Object` reference type via a call to `set_return_type` method. After a call has been made, the return value is extracted with the result's call to the method `result`. The same steps are repeated to invoke another method on an `Account Manager` instance with the only difference being in-parameters and return types.

The `req`, an `Any` object is initialized with the desired account name and added to the request's argument list as an input argument. The last step in initializing the request is to set the `result` value to receive a float.

Example of creating a Request object

A Request object maintains ownership of all memory associated with the operation, the arguments, and the result so you should never attempt to free these items.

Code sample 22.4 Creating a request object

```
// Client.java
public class Client {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.out.println("Usage: vbj Client <manager-name> <account-name>\n");
            return;
        }
        String managerName = args[0];
        String accountName = args[1];
        org.omg.CORBA.Object accountManager, account;
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
        accountManager = orb.bind("IDL:Bank/AccountManager:1.0",
            managerName, null, null);
        org.omg.CORBA.Request request = accountManager._request("open");
        request.add_in_arg().insert_string(accountName);
        request.set_return_type(orb.get_primitive_tc(
            org.omg.CORBA.TCKind.tk_objref)
        );
        request.invoke();
        account = request.result().value().extract_Object();
        org.omg.CORBA.Request request = account._request("balance");
        request.set_return_type(orb.get_primitive_tc(
            org.omg.CORBA.TCKind.tk_float)
        );
        request.invoke();
        float balance = request.result().value().extract_float();
        System.out.println("The balance in " + accountName + "'s account is $" + balance);
    }
}
```

Setting arguments for the request

The arguments for a Request are represented with a `NVList` object, which stores name-value pairs as `NamedValue` objects. You can use the `arguments` method to obtain a pointer to this list. This pointer can then be used to set the names and values of each of the arguments.

Note Always initialize the arguments before sending a `Request`. Failure to do so will result in marshalling errors and may even cause the server to abort.

Implementing a list of arguments with the NVList

This class implements a list of `NamedValue` objects that represent the arguments for a method invocation. Methods are provided for adding, removing, and querying the objects in the list.

Code sample 22.5 NVList class

```

package org.omg.CORBA;

public abstract class NVList {
    public int count();
    public void add(int flags);
    public void add_item(java.lang.String name, int flags);
    public void add_value(
        java.lang.String name,
        org.omg.CORBA.Any value,
        int flags
    );
    public org.omg.CORBA.NamedValue item(int index);
    public void remove(int index);
}

```

Setting input and output arguments with the NamedValue Class

This class implements a name-value pair that represents both input and output arguments for a method invocation request. The `NamedValue` class is also used to represent the result of a request that is returned to the client program. The `name` property is simply a character string and the `value` property is represented by an `Any` class.

Note There is no constructor for this class. The `ORB.create_named_value` method is used to obtain a reference to a `NamedValue` object.

Code sample 22.6 NamedValue interface

```

package org.omg.CORBA;

public abstract class NamedValue {
    public java.lang.String name();
    public org.omg.CORBA.Any value();
    public int flags();
}

```

The following table describes the methods in the `NamedValue` class.

Table 22.1 NamedValue methods

Method	Description
<code>name</code>	Returns a pointer to the name of the item that you can then use to initialize the name.
<code>value</code>	Returns a pointer to an <code>Any</code> object representing the item's value that you can then use to initialize the value. For more information, see "Passing type safely with the <code>Any</code> class" on page 22-10.
<code>flags</code>	Indicates if this item is an input argument, an output argument, or both an input and output argument. If the item is both an input and output argument, you can specify a flag indicating that the ORB should make a copy of the argument and leave the caller's memory intact. Flags are <div style="margin-left: 20px;"> <code>ARG_IN</code> <code>ARG_OUT</code> <code>ARG_INOUT</code> </div>

Passing type safely with the Any class

This class is used to hold an IDL-specified type so that it may be passed in a type-safe manner. Objects of this class have a reference to a `TypeCode` that defines the contained object's type and a reference to the contained object. Methods are provided to construct, copy, and release an object as well as initialize and query the object's value and type. In addition, streaming methods are provided to read and write the object from and to a stream.

Code sample 22.7 Any class

```
package org.omg.CORBA;
public abstract class Any {
    public abstract TypeCode type();
    public abstract void type(TypeCode type);
    public abstract void read_value(InputStream input, TypeCode type);
    public abstract void write_value(OutputStream output);
    public abstract boolean equal(Any rhs);
    . . .
}
```

Representing argument or attribute types with the TypeCode class

This class is used by the Interface Repository and the IDL compiler to represent the type of arguments or attributes. `TypeCode` objects are also used in a `Request` object to specify an argument's type, in conjunction with the `Any` class. `TypeCode` objects have a `kind` and `parameter list` property, represented by one of the values defined by the `TCKind` class.

Note There is no constructor for this class. Use the `ORB.get_primitive_tc` method or one of the `ORB.create_*_tc` methods to create a `TypeCode` object. For more details, see Chapter 5, “Core interfaces and classes,” of the *VisiBroker for Java Reference*.

The following table shows the kinds and parameters for the `TypeCode` objects.

Table 22.2 TypeCode kinds and parameters

Kind	Parameter list
tk_abstract_interface	repository_id, interface_name
tk_alias	repository_id, alias_name, TypeCode
tk_any	None
tk_array	length, TypeCode
tk_boolean	None
tk_char	None
tk_double	None
tk_enum	repository_id, enum-name, enum-id ¹ , enum-id ² , ... enum-id ⁿ
tk_except	repository_id, exception_name, StructMembers
tk_fixed	digits, scale
tk_float	None

Table 22.2 TypeCode kinds and parameters (continued)

Kind	Parameter list
tk_long	None
tk_longdouble	None
tk_longlong	None
tk_native	id, name
tk_null	None
tk_objref	repository_id, interface_id
tk_octet	None
tk_Principal	None
tk_sequence	TypeCode, maxlen
tk_short	None
tk_string	maxlen-integer
tk_struct	repository_id, struct-name, {member ¹ , TypeCode ¹ }, {member ⁿ , TypeCode ⁿ }
tk_TypeCode	None
tk_ulong	None
tk_ulonglong	None
tk_union	repository_id, union-name, switch TypeCode, {label-value ¹ , member-name ¹ , TypeCode ¹ }, {label ¹ -value ⁿ , member-name ⁿ , TypeCode ⁿ }
tk_ushort	None
tk_value	repository_id, value_name, boxType
tk_value_box	repository_id, value_name, typeModifier, concreteBase, members
tk_void	None
tk_wchar	None
tk_wstring	None

Code sample 22.8 TypeCode interface

```

public abstract class TypeCode extends java.lang.Object
    implements org.omg.CORBA.portable.IDLEntity {
    public abstract boolean equal(org.omg.CORBA.TypeCode tc);
    public boolean equivalent(org.omg.CORBA.TypeCode tc);
    public abstract org.omg.CORBA.TCKind kind();
    public TypeCode get_compact_typecode()
    public abstract java.lang.String id()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public abstract java.lang.String name()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public abstract int member_count()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public abstract java.lang.String member_name(int index)
        throws org.omg.CORBA.TypeCodePackage.BadKind,
            org.omg.CORBA.TypeCodePackage.Bounds;
    public abstract org.omg.CORBA.TypeCode member_type(int index)
        throws org.omg.CORBA.TypeCodePackage.BadKind,
            org.omg.CORBA.TypeCodePackage.Bounds;

    public abstract org.omg.CORBA.Any member_label(int index)
        throws org.omg.CORBA.TypeCodePackage.BadKind,
            org.omg.CORBA.TypeCodePackage.Bounds;
    public abstract org.omg.CORBA.TypeCode discriminator_type()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public abstract int default_index()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public abstract int length()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public abstract org.omg.CORBA.TypeCode content_type()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public short fixed_digits()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public short fixed_scale()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public short member_visibility(int index)
        throws org.omg.CORBA.TypeCodePackage.BadKind,
            org.omg.CORBA.Bounds;
    public short type_modifier()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
    public TypeCode concrete_base_type()
        throws org.omg.CORBA.TypeCodePackage.BadKind;
}

```

Sending DII requests and receiving results

The `Request` class, shown in Code sample 22.2 on page 22-6, provides several methods for sending a request, once it has been properly initialized.

Invoking a request

The simplest way to send a request is to call its `invoke` method, which sends the request and waits for a response before returning to your client program. The `return_value` method returns a reference to an `Any` object that represents the return value.

Code sample 22.9 Sending a request with invoke

```
try {
    . . .
    // Create request that will be sent to the account object
    request = account._request("balance");
    // Set the result type
    request.set_return_type(orb.get_primitive_tc
        (org.omg.CORBA.TCKind.tk_float));
    // Execute the request to the account object
    request.invoke();
    // Get the return balance
    float balance;
    org.omg.CORBA.Any balance_result = request.return_value();
    balance = balance_result.extract_float();
    // Print out the balance
    System.out.println("The balance in " + name + "'s account is $" +
        balance);
} catch(Exception e) {
    e.printStackTrace();
}
```

Sending a deferred DII request with the `send_deferred` method

A non-blocking method, `send_deferred`, is also provided for sending operation requests. It allows your client to send the request and then use the `poll_response` method to determine when the response is available. The `get_response` method blocks until a response is received. The following code shows how these methods are used.

Code sample 22.10 Using the `send_deferred` and `poll_response` methods to send a deferred DII request

```
try {
    . . .
    // Create request that will be sent to the manager object
    org.omg.CORBA.Request request = manager._request("open");
    // Create argument to request
    org.omg.CORBA.Any customer = orb.create_any();
    customer.insert_string(name);
    org.omg.CORBA.NVList arguments = request.arguments();
    arguments.add_value("name" , customer, org.omg.CORBA.ARG_IN.value);
    // Set result type
    request.set_return_type(orb.get_primitive_tc
        (org.omg.CORBA.TCKind.tk_objref));
    // Creation of a new account can take some time
    // Execute the deferred request to the manager object-plist
    request.send_deferred();
    Thread.currentThread().sleep(1000);
    while (!request.poll_response()) {
        System.out.println(" Waiting for response...");
        Thread.currentThread().sleep(1000); // Wait one second between polls
    }
    request.get_response();
    // Get the return value
    org.omg.CORBA.Object account;
    org.omg.CORBA.Any open_result = request.return_value();
    account = open_result.extract_Object();
    . . .
} catch(Exception e) {
    e.printStackTrace();
}
```

Sending an asynchronous DII request with the `send_oneway` method

The `send_oneway` method can be used to send an asynchronous request. Oneway requests do not involve a response being returned to the client from the object implementation.

Sending multiple requests

A sequence of DII Request objects can be created using array of Request objects. A sequence of requests can be sent using the ORB methods

`send_multiple_requests_oneway` or `send_multiple_requests_deferred`. If the sequence of requests is sent as oneway requests, no response is expected from the server to any of the requests.

Receiving multiple requests

When a sequence of requests is sent using `send_multiple_requests_deferred`, the `poll_next_response` and `get_next_response` methods are used to receive the response the server sends for each request.

The ORB method `poll_next_response` can be used to determine if a response has been received from the server. This method returns true if there is at least one response available. This method returns false if there are no responses available.

The ORB method `get_next_response` can be used to receive a response. If no response is available, this method will block until a response is received. If you do not wish your client program to block, use the `poll_next_response` method to first determine when a response is available and then use the `get_next_response` method to receive the result.

Code sample 22.11 ORB methods for sending multiple requests and receiving the results

```
package org.omg.CORBA;
public abstract class ORB {
    public abstract org.omg.CORBA.Environment create_environment();
    public abstract void send_multiple_requests_oneway(org.omg.CORBA.Request[] reqs);
    public abstract void send_multiple_requests_deferred(org.omg.CORBA.Request[] reqs);
    public abstract boolean poll_next_response();
    public abstract org.omg.CORBA.Request get_next_response();
    . . .
}
```

Using the interface repository with the DII

One source of the information needed to populate a DII `Request` object is an interface repository (IR) (see Chapter 21, “Using interface repositories”). The following example uses an interface repository to get obtain the parameters of an operation. Note that the example, atypical of real DII applications, has built-in knowledge of a remote object’s type (`Account`) and the name of one of its methods (`balance`). An actual DII application would get that information from an outside source—for example, a user.

The example

- Binds to any `Account` object.
- Looks up the `Account`’s `balance` method in the IR and builds an operation list from the IR `OperationDef`.
- Creates argument and result components and passes these to the `_create_request` method. Note that the `balance` method does not return an exception.
- Invokes the `Request`, extracts and prints the result.

```
// Java TBD (no example)
```

Using the Dynamic Skeleton Interface

This chapter describes how object servers can dynamically create object implementations at run time to service client requests.

What is the Dynamic Skeleton Interface?

The Dynamic Skeleton Interface (DSI) provides a mechanism for creating an object implementation that does not inherit from a generated skeleton interface. Normally, an object implementation is derived from a skeleton class generated by the `idl2java` compiler. The DSI allows an object to register itself with the ORB, receive operation requests from a client, process the requests, and return the results to the client without inheriting from a skeleton class generated by the `idl2java` compiler.

Note From the perspective of a client program, an object implemented with the DSI behaves just like any other ORB object. Clients do not need to provide any special handling to communicate with an object implementation that uses the DSI.

The ORB presents client operation requests to a DSI object implementation by calling the object's `invoke` method and passing it a `ServerRequest` object. The object implementation is responsible for determining the operation being requested, interpreting the arguments associated with the request, invoking the appropriate internal method or methods to fulfill the request, and returning the appropriate values.

Implementing objects with the DSI requires more manual programming activity than using the normal language mapping provided by object skeletons. Nevertheless, an object implemented with the DSI can be very useful in providing inter-protocol bridging.

Using the idl2java compiler

The `idl2java` compiler has a flag (`-dynamic_marshal`) which, when switched on, generates skeleton code using DSI. To understand how to do any type of DSI: create an IDL file, generate with `-dynamic_marshal`, and look at the skeleton code.

Steps for creating object implementations dynamically

To create object implementations dynamically using the DSI, follow these steps:

- 1 Use the `-dynamic_marshal` flag when compiling your IDL.
- 2 Design your object implementation so that it is derived from the `org.omg.PortableServer.DynamicImplementation` interface instead of deriving your object implementation from a skeleton class.
- 3 Declare and implement the `invoke` method, which the ORB will use to dispatch client requests to your object.
- 4 Register your object implementation (POA servant) with the POA manager as the default servant.

Location of an example program for using the DSI

An example program that illustrates the use of the DSI is included in the `examples/basic/bank_dynamic` directory of the VisiBroker distribution. This example is used to illustrate DSI concepts in this chapter. The `Bank.idl` file, shown in IDL sample 23.1, illustrates the interfaces implemented in this example.

IDL sample 23.1 `Bank.idl` file used in the DSI example

```
// Bank.idl
module Bank {
    interface Account {
        float balance();
    };
    interface AccountManager {
        Account open(in string name);
    };
};
```

Extending the DynamicImplementation class

To use the DSI, object implementations should be derived from the `DynamicImplementation` base class shown below. This class offers several constructors and the `invoke` method, which you must implement.

Code sample 23.1 `DynamicImplementation` abstract class

```
package org.omg.CORBA;

public abstract class DynamicImplementation extends Servant {
    public abstract void invoke(ServerRequest request);
    . . .
}
```

Example of designing objects for dynamic requests

Code sample 23.2 shows the declaration of the `AccountImpl` class that is to be implemented with the DSI. It is derived from the `DynamicImplementation` class, which declares the `invoke` method. The ORB will call the `invoke` method to pass client operation requests to the implementation in the form of `ServerRequest` objects.

Also note the `Account` class constructor as shown in Code sample 23.2.

Code sample 23.2 `AccountImpl` class from the dynamic example

```
import java.util.*;
import org.omg.PortableServer.*;

public class AccountImpl extends DynamicImplementation {
    public AccountImpl(org.omg.CORBA.ORB orb, POA poa) {
        _orb = orb;
        _poa = poa;
    }

    public synchronized org.omg.CORBA.Object get(String name) {
        org.omg.CORBA.Object obj;
        // Check if account exists
        Float balance = (Float)_registry.get(name);
        if (balance == null) {
            // simulate delay while creating new account
            try {
                Thread.currentThread().sleep(3000);
            } catch (Exception e) {
                e.printStackTrace();
            }
            // Make up the account's balance, between 0 and 1000 dollars
            balance = new Float(Math.abs(_random.nextInt()) % 100000 / 100f);
            // Print out the new account
            System.out.println("Created " + name + "'s account: " +
                balance.floatValue());
            _registry.put(name, balance);
        }
        // Return object reference
        byte[] accountId = name.getBytes();
        try {
```

```

        obj = _poa.create_reference_with_id(accountId, "IDL:Bank/Account:1.0");
    } catch (org.omg.PortableServer.POAPackage.WrongPolicy e) {
        throw new org.omg.CORBA.INTERNAL(e.toString());
    }
    return obj;
}

public String[] _all_interfaces(POA poa, byte[] objectId) { return null; }

public void invoke(org.omg.CORBA.ServerRequest request) {
    Float balance;
    // Get the account name from the object id
    String name = new String(_object_id());
    // Ensure that the operation name is correct
    if (!request.operation().equals("balance")) {
        throw new org.omg.CORBA.BAD_OPERATION();
    }
    // Find out balance and fill out the result
    org.omg.CORBA.NVList params = _orb.create_list(0);
    request.arguments(params);
    balance = (Float)_registry.get(name);
    if (balance == null) {
        throw new org.omg.CORBA.OBJECT_NOT_EXIST();
    }
    org.omg.CORBA.Any result = _orb.create_any();
    result.insert_float(balance.floatValue());
    request.set_result(result);
    System.out.println("Checked " + name + "'s balance: " +
        balance.floatValue());
}

private Random _random = new Random();
static private Hashtable _registry = new Hashtable();
private POA _poa;
private org.omg.CORBA.ORB _orb;
}

```

The code in Code sample 23.3 shows the implementation of the `AccountManagerImpl` class that needs to be implemented with the DSI. It is also derived from the `DynamicImplementation` class, which declares the `invoke` method. The ORB will call the `invoke` method to pass client operation requests to the implementation in the form of `ServerRequest` objects.

Code sample 23.3 AccountManagerImpl class from the dynamic example

```

import org.omg.PortableServer.*;
public class AccountManagerImpl extends DynamicImplementation {
    public AccountManagerImpl(org.omg.CORBA.ORB orb, AccountImpl accounts) {
        _orb = orb;
        _accounts = accounts;
    }

    public synchronized org.omg.CORBA.Object open(String name) {
        return _accounts.get(name);
    }
}

```

```

public String[] _all_interfaces(POA poa, byte[] objectId) { return null; }

public void invoke(org.omg.CORBA.ServerRequest request) {
    // Ensure that the operation name is correct
    if (!request.operation().equals("open")) {
        throw new org.omg.CORBA.BAD_OPERATION();
    }

    // Fetch the input parameter
    String name = null;
    try {
        org.omg.CORBA.NVList params = _orb.create_list(1);
        org.omg.CORBA.Any any = _orb.create_any();
        any.insert_string(new String(""));
        params.add_value("name", any, org.omg.CORBA.ARG_IN.value);
        request.arguments(params);
        name = params.item(0).value().extract_string();
    } catch (Exception e) {
        throw new org.omg.CORBA.BAD_PARAM();
    }

    // Invoke the actual implementation and fill out the result
    org.omg.CORBA.Object account = open(name);
    org.omg.CORBA.Any result = _orb.create_any();
    result.insert_Object(account);
    request.set_result(result);
}

private AccountImpl _accounts;
private org.omg.CORBA.ORB _orb;
}

```

Specifying repository ids

The `_primary_interface` method should be implemented to return supported repository identifiers. To determine the correct repository identifier to specify, start with the IDL interface name of an object and use the following steps:

- 1 Replace all non-leading instances of the delimiter scope resolution operator (`::`) with a slash (`/`).
- 2 Add `"IDL:"` to the beginning of the string.
- 3 Add `":1.0"` to the end of the string.

For example, Code sample 23.4 shows an IDL interface name and Code sample 23.5 shows the resulting repository identifier string.

Code sample 23.4 IDL interface name

```
Bank::AccountManager
```

Code sample 23.5 Resulting repository identifier

```
IDL:Bank/AccountManager:1.0
```

Looking at the ServerRequest class

A `ServerRequest` object is passed as a parameter to an object implementation's `invoke` method. The `ServerRequest` object represents the operation request and provides methods for obtaining the name of the requested operation, the parameter list, and the context. It also provides methods for setting the result to be returned to the caller and for reflecting exceptions.

Code sample 23.6 `ServerRequest` abstract class

```
package org.omg.CORBA;

public abstract class ServerRequest {
    public java.lang.String operation();
    public void arguments(org.omg.CORBA.NVList args);
    public void set_result(org.omg.CORBA.Any result);
    public void set_exception(org.omg.CORBA.Any except);
    public abstract org.omg.CORBA.Context ctx();
    // the following methods are deprecated
    public java.lang.String op_name(); // use operation()
    public void params(org.omg.CORBA.NVList params); // use arguments()
    public void result(org.omg.CORBA.Any result); // use set_result()
    public abstract void except(org.omg.CORBA.Any except); // use set_exception()
}
```

All arguments passed into the `arguments`, `set_result`, or `set_exception` methods are thereafter owned by the ORB. The memory for these arguments will be released by the ORB—you should not release them.

Note The following methods have been deprecated:

- `op_name`
- `params`
- `result`
- `exception`

Implementing the Account object

The `Account` interface declares only one method, so the processing done by the `AccountImpl` class' `invoke` method is fairly straightforward.

The `invoke` method first checks to see if the requested operation has the name "balance." If the name does not match, a `BAD_OPERATION` exception is raised. If the `Account` object were to offer more than one method, the `invoke` method would need to check for all possible operation names and use the appropriate internal methods to process the operation request.

Since the `balance` method does not accept any parameters, there is no parameter list associated with its operation request. The `balance` method is simply invoked and the result is packaged in an `Any` object that is returned to the caller, using the `ServerRequest` object's `set_result` method.

Implementing the AccountManager object

Like the `Account` object, the `AccountManager` interface also declares one method. However, the `AccountManagerImpl` object's `open` method does accept an account name parameter. This makes the processing done by the `invoke` method a little more complicated. Code sample 23.2 on page 23-3 shows the implementation of the `AccountManagerImpl` object's `invoke` method.

The method first checks to see that the requested operation has the name “open.” If the name does not match, a `BAD_OPERATION` exception is raised. If the `AccountManager` object were to offer more than one method, its `invoke` method would need to check for all possible operation names and use the appropriate internal methods to process the operation request.

Processing input parameters

Here are the steps the `AccountManagerImpl` object's `invoke` method uses to process the operation request's input parameters.

- 1 Create an `NVList` to hold the parameter list for the operation.
- 2 Create `Any` objects for each expected parameter and add them to the `NVList`, setting their `TypeCode` and parameter type (`ARG_IN`, `ARG_OUT`, or `ARG_INOUT`).
- 3 Invoke the `ServerRequest` object's `arguments` method, passing the `NVList`, to update the values for all the parameters in the list.

The `open` method expects an account name parameter; therefore, an `NVList` object is created to hold the parameters contained in the `ServerRequest`. The `NVList` class implements a parameter list containing one or more `NamedValue` objects. The `NVList` and `NamedValue` classes are described in Chapter 22, “Using the Dynamic Invocation Interface.”

An `Any` object is created to hold the account name. This `Any` is then added to `NVList` with the argument's name set to “name” and the parameter type set to `ARG_IN`.

Once the `NVList` has been initialized, the `ServerRequest` object's `arguments` method is invoked to obtain the values of all of the parameters in the list.

Note After invoking the `arguments` method, the `NVList` will be owned by the ORB. This means that if an object implementation modifies an `ARG_INOUT` parameter in the `NVList`, the change will automatically be apparent to the ORB. This `NVList` should not be released by the caller.

An alternative to constructing the `NVList` for the input arguments is to use the ORB object's `create_operation_list` method. This method accepts an `OperationDef` and returns an `NVList` object, completely initialized with all the necessary `Any` objects. The appropriate `OperationDef` object may be obtained from the interface repository, described in Chapter 21, “Using interface repositories.”

Setting the return value

After invoking the `ServerRequest` object's `arguments` method, the value of the `name` parameter can be extracted and used to create a new `Account` object. An `Any` object is

created to hold the newly created `Account` object, which is returned to the caller by invoking the `ServerRequest` object's `set_result` method.

Server implementation

The implementation of the `main` routine, shown in Code sample 23.7, is almost identical to the original example introduced in Chapter 4, “Developing an example application with VisiBroker.”

Code sample 23.7 Server implementation

```
import org.omg.PortableServer.*;

public class Server {
    public static void main(String[] args) {
        try {
            // Initialize the ORB
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // Get a reference to the root POA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Get the POA Manager
            POAManager poaManager = rootPOA.the_POAManager();
            // Create the account POA with the right policies
            org.omg.CORBA.Policy[] accountPolicies = {
                rootPOA.create_servant_retention_policy(
                    ServantRetentionPolicyValue.NON_RETAIN),
                rootPOA.create_request_processing_policy(
                    RequestProcessingPolicyValue.USE_DEFAULT_SERVANT)
            };
            POA accountPOA = rootPOA.create_POA("bank_account_poa",
                poaManager, accountPolicies);
            // Create the account default servant
            AccountImpl accountServant = new AccountImpl(orb, accountPOA);
            accountPOA.set_servant(accountServant);
            // Create the manager POA with the right policies
            org.omg.CORBA.Policy[] managerPolicies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
                rootPOA.create_request_processing_policy(
                    RequestProcessingPolicyValue.USE_DEFAULT_SERVANT)
            };
            POA managerPOA = rootPOA.create_POA("bank_agent_poa",
                poaManager, managerPolicies);
            // Create the manager default servant
            AccountManagerImpl managerServant = new AccountManagerImpl(orb, accountServant);
            managerPOA.set_servant(managerServant);
            // Activate the POA Manager
            poaManager.activate();
            System.out.println("AccountManager is ready");
            // Wait for incoming requests
            orb.run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

DSI implementation is instantiated as a default servant and the POA should be created with the support of corresponding policies. For more information see Chapter 7, “Using POAs.”

Using interceptors

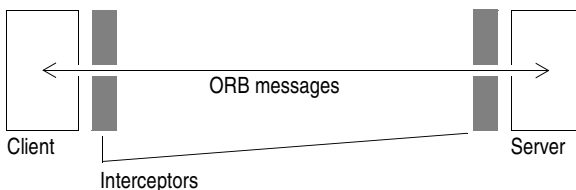
This chapter provides an overview of the new interceptors framework, walks through an interceptor example, and describes some advanced features such as interceptor factories and chaining interceptors.

Overview

The VisiBroker ORB provides a set of interfaces known as interceptors which provide a framework for plugging in additional ORB behavior such as security, transactions, or logging. These interceptor interfaces are based on a *callback* mechanism. For example, using interceptors, you can be notified of communications between clients and servers, and modify these communications if you wish, effectively altering the behavior of the VisiBroker ORB.

At its simplest usage, the interceptor is useful for tracing through code. Because you can see the messages being sent between clients and servers, you can determine exactly how the ORB is processing requests.

Figure 24.1 How interceptors work



If you are building a more sophisticated application such as a monitoring tool or security layer, interceptors give you the information and control you need to enable these lower level applications. For example, you could develop an application that monitors the activity of various servers and performs load balancing.

There are two kinds of interceptors:

- Client interceptors are system-level interceptors which are called when a method is invoked on a client object.
- Server interceptors are system-level interceptors which are called when a method is invoked on a server object.

To use interceptors you declare a class which implements one of the interceptor interfaces. Once you have instantiated an interceptor object, you register it with its corresponding interceptor manager. Your interceptor object will then be notified by its manager whenever, for example, an object has had one of its methods invoked or its parameters marshalled or demarshalled.

Note Use object wrappers, described in Chapter 25, “Using object wrappers,” if you want to intercept an operation request before it is marshalled on the client-side or if you want to intercept an operation request before it is process on the server-side.

Interceptor interfaces and managers

Interceptor developers derive classes from one or more of the following base interceptor API classes which are defined and implemented by the VisiBroker ORB.

- Client interceptors
 - `BindInterceptor`
 - `ClientRequestInterceptor`
- Server interceptors
 - `POALifeCycleInterceptor`
 - `ActiveObjectLifeCycleInterceptor`
 - `ServerRequestInterceptor`
 - `IORCreationInterceptor`
- `ServiceResolver` interceptor

Client interceptors

There are currently two kinds of client interceptor and their respective managers:

- `BindInterceptor` **and** `BindInterceptorManager`
- `ClientRequestInterceptor` **and** `ClientRequestInterceptorManager`

For more details about client interceptors see Chapter 12, “Interceptor and object wrapper interfaces and classes,” in the VisiBroker for Java *Reference*.

BindInterceptor

A `BindInterceptor` object is a global interceptor which is called on the client side before and after binds.

Code sample 24.1 BindInterceptor interface

```

package com.inprise.vbroker.InterceptorExt;
public interface BindInterceptor {
    public IORValue bind(IORValue ior,
        org.omg.CORBA.Object target,
        boolean rebind,
        Closure closure);
    public IORValue bind_failed(IORValue ior,
        org.omg.CORBA.Object target,
        Closure closure);
    public void bind_succeeded(IORValue ior,
        org.omg.CORBA.Object target,
        int Index,
        InterceptorManagerControl control,
        Closure closure);
    public void exception_occurred(IORValue ior,
        org.omg.CORBA.Object target,
        org.omg.CORBA.Environment env,
        Closure closure);
}

```

ClientRequestInterceptor

A **ClientRequestInterceptor** object may be registered during a `bind_succeeded` call of a **BindInterceptor** object, and it remains active for the duration of the connection. Two of its methods are called before the invocation on the client object, one (`preinvoke_premarshal`) before the parameters are marshalled and the other (`preinvoke_postmarshal`) after they are. The third method (`postinvoke`) is called after the request has completed.

Code sample 24.2 ClientRequestInterceptor interface

```

package com.inprise.vbroker.InterceptorExt;
public interface ClientRequestInterceptor {
    public void preinvoke_premarshal(org.omg.CORBA.Object target,
        String operation,
        ServiceContextListHolder service_contexts_holder,
        Closure closure);
    public void preinvoke_postmarshal(org.omg.CORBA.Object target,
        OutputStream payload,
        Closure closure);
    public void postinvoke(org.omg.CORBA.Object target,
        ServiceContext[] service_contexts,
        InputStream payload,
        org.omg.CORBA.Environment env,
        Closure closure);
    public void exception_occurred(org.omg.CORBA.Object target,
        org.omg.CORBA.Environment env,
        Closure closure);
}

```

Server interceptors

There are currently four kinds of server interceptors:

- POALifeCycleInterceptor **and** POALifeCycleInterceptorManager
- ActiveObjectLifeCycleInterceptor **and** ActiveObjectLifeCycleInterceptorManager
- ServerRequestInterceptor **and** ServerRequestInterceptorManager
- IORCreationInterceptor **and** IORCreationInterceptorManager

For more details about server interceptors see Chapter 12, “Interceptor and object wrapper interfaces and classes,” in the VisiBroker for Java *Reference*.

POALifeCycleInterceptor

A POALifeCycleInterceptor object is a global interceptor which is called every time a POA is created (via the `create` method) or destroyed (via the `destroy` method).

Code sample 24.3 POALifeCycleInterceptor interface

```
package com.inprise.vbroker.InterceptorExt;
public interface POALifeCycleInterceptor {
    public void create(org.omg.PortableServer.POA poa,
                      org.omg.CORBA.PolicyListHolder policies_holder,
                      IORValueHolder iorTemplate,
                      InterceptorManagerControl control) ;
    public void destroy(org.omg.PortableServer.POA poa);
}
```

ActiveObjectLifeCycleInterceptor

An ActiveObjectLifeCycleInterceptor object is called whenever an object is added to the Active Object Map (via the `create` method) or after an object has been deactivated and etherialized (via the `destroy` method). The interceptor may be registered by a POALifeCycleInterceptor on a per-POA basis at POA creation time. This interceptor may only be registered if the POA has the `RETAIN` policy.

Code sample 24.4 ActiveObjectLifeCycleInterceptor interface

```
package com.inprise.vbroker.InterceptorExt;
public interface ActiveObjectLifeCycleInterceptor {
    public void create(byte[] oid,
                      org.omg.PortableServer.Servant servant,
                      org.omg.PortableServer.POA adapter);
    public void destroy (byte[] oid,
                        org.omg.PortableServer.Servant servant,
                        org.omg.PortableServer.POA adapter);
}
```

ServerRequestInterceptor

A ServerRequestInterceptor object is called at various stages in the invocation of a server implementation of a remote object: before the invocation (via the `preinvoke` method) and after the invocation both before and after the marshalling of the reply (via the `postinvoke_premarshal` and `postinvoke_postmarshal` methods respectively). This

interceptor may be registered by a `POALifeCycleInterceptor` object at POA creation time on a per-POA basis.

Code sample 24.5 `ServerRequestInterceptor` interface

```
package com.inprise.vbroker.InterceptorExt;
public interface ServerRequestInterceptor {
    public void preinvoke(org.omg.CORBA.Object target,
        String operation,
        ServiceContext[] service_contexts,
        InputStream payload,
        Closure closure);
    public void postinvoke_premarshal(org.omg.CORBA.Object target,
        ServiceContextListHolder service_contexts_holder,
        org.omg.CORBA.Environment env,
        Closure closure);
    public void postinvoke_postmarshal(org.omg.CORBA.Object target,
        OutputStream payload,
        Closure closure);
    public void exception_occurred(org.omg.CORBA.Object target,
        org.omg.CORBA.Environment env,
        Closure closure);
}
```

IORCreationInterceptor

An `IORCreationInterceptor` object is called whenever a POA creates an object reference (via the `create` method). This interceptor may be registered by a `POALifeCycleInterceptor` at POA creation time on a per-POA basis.

IDL sample 24.1 `IORCreationInterceptor` interface

```
package com.inprise.vbroker.InterceptorExt;
public interface IORCreationInterceptor {
    public void create(org.omg.PortableServer.POA poa,
        IORValueHolder ior);
}
```

Service Resolver interceptor

This interceptor is used to install a user service that you can then dynamically load.

Code sample 24.6 `ServiceResolverInterceptor`

```
public interface ServiceResolverInterceptor {
    public org.omg.CORBA.Object resolve (java.lang.String name):
}
public interface ServiceResolverInterceptorManager extends
com.inprise.vbroker.interceptor.InterceptorManager {
    public void add (java.lang.String name,
com.inprise.vbroker.interceptor.ServiceResolverInterceptor \interceptor) ;
    public void remove (java.lang.String name):
}
```

When you do a `resolve_initial_references()`, the `resolve` on all user’s installed services gets called. The `resolve` then can return the appropriate object.

To write service initializers, you must obtain a `ServiceResolver` after getting an `InterceptorManagerControl` to be able to add your services.

Default interceptor classes

`VisiBroker` provides default interceptor Java classes that you can *extend* rather than implement. These default interceptor classes offer the same methods as the interceptor interfaces; however, when you extend the default interceptor class, you can choose which methods to implement or override. When you use these classes, you can accept the default behavior that they provide or change it.

- `DefaultBindInterceptor` class
- `DefaultClientInterceptor` class
- `DefaultServerInterceptor` class

Registering interceptors with the VisiBroker ORB

Each interceptor interface has a corresponding interceptor manager interface which is used to register your interceptor objects with the ORB. The following steps are those necessary to register an interceptor:

- 1 Get a reference to an `InterceptorManagerControl` object by calling the `resolve_initial_references` method on an ORB object with the parameter “`VisiBrokerInterceptorControl`.”
- 2 Call the `get_manager` method on the `InterceptorManagerControl` object with one of the String values in Table 24.1. (Be sure to cast the object reference to its corresponding interceptor manager interface.)

Table 24.1 String values to pass to the `get_manager` method of the `InterceptorManagerControl` object

Value	Corresponding interceptor interface
<code>ClientRequest</code>	<code>ClientRequestInterceptor</code>
<code>Bind</code>	<code>BindInterceptor</code>
<code>POALifeCycle</code>	<code>POALifeCycleInterceptor</code>
<code>ActiveObjectLifeCycle</code>	<code>ActiveObjectLifeCycleInterceptor</code>
<code>ServerRequest</code>	<code>ServerRequestInterceptor</code>
<code>IORCreation</code>	<code>IORCreationInterceptor</code>
<code>ServiceResolver</code>	<code>ServiceResolverInterceptor</code>

- 3 Create an instance of your interceptor.
- 4 Register your interceptor object with the manager object by calling the `add` method.
- 5 Load your interceptor objects when running your client and server programs.

Creating interceptor objects

Finally, you need to implement a factory class which creates instances of your interceptor and registers them with the ORB. Your factory class must implement the `ServiceLoader` interface.

Code sample 24.7 `ServiceLoader` interface

```
package com.inprise.vbroker.interceptor;

public interface ServiceLoader {
    // This method is called by the ORB when ORB.init() is called.
    public abstract void init(org.omg.CORBA.ORB orb);

    // Called after ORB.init() is done but control hasn't been returned to
    // the user. Can be used to disable certain resources that were only
    // made available to other service inits.
    public abstract void init_complete(org.omg.CORBA.ORB orb);

    // Called when the orb is being shutdown.
    public abstract void shutdown(org.omg.CORBA.ORB orb);
}
```

Note You may also create new instances of your interceptors and register them with the ORB from within other interceptors as in the example below.

Loading interceptors

To load your interceptor, you must set the `vbroker.ORB.dynamicLibs` property. This property can be set either in the properties file (see Chapter 14, “Setting properties”) or be passed into the ORB using the `-D` option.

Example interceptors

The example interceptor below uses all of the interceptor API methods (listed in Chapter 12, “Interceptor and object wrapper interfaces and classes,” in the *VisiBroker for Java Reference*) so that you can see how these methods are used, and when they are invoked.

Example code

In “Code listings” on page 24-10, each of the interceptor API methods is simply implemented to print informational messages to the standard output.

There are four example applications in the `examples/interceptors` directory in your VisiBroker installation:

- `active_object_lifecycle`
- `client_server`
- `ior_creation`
- `encryption`

Client-server interceptors example

To run the example, compile the files as you normally would. Then start up the Server and the Client as follows:

```
prompt>vbj -Dvbroker.orb.dynamicLibs=SampleServerLoader Server
prompt>vbj -Dvbroker.orb.dynamicLibs=SampleClientLoader Client Kate
```

You specify as ORB services the two classes which implement the `ServiceLoader` interface.

Note The `ServiceInit` class used in VisiBroker 3.x is replaced by implementing two interfaces: `ServiceLoader` and `ServiceResolverInterceptor`. For more information about the `ServiceResolverInterceptor` interface, see “Service Resolver interceptor” on page 24-5. For an example of how to do this, see “ServiceResolverInterceptor example” on page 24-9.

The results of executing the example interceptor are shown in Table 24.2. The execution by the client and server is listed in sequence.

Table 24.2 Results of executing the example interceptor

Client	Server
	=====>SampleServerLoader:Interceptors loaded
	=====>In POA /. Nothing to do.
	=====>In POA bank_agent_poa, 1
	ServerRequest interceptor installed
	Stub[repository_id=IDL:Bank/
	AccountManager:1.0,key=ServiceId[service=/
	bank_agent
	_poa,id={11 bytes:
	[B][a][n][k][M][a][n][a][g][e][r]] is ready.
Bind Interceptors loaded	
=====> SampleBindInterceptor bind	
=====> SampleBindInterceptor	
bind_succeeded	
=====> SampleClientInterceptor id	
MyClientInterceptor preinvoke_premarshal	
=> open	
=====> SampleClientInterceptor id	
MyClientInterceptor preinvoke_postmarshal	

Table 24.2 Results of executing the example interceptor (continued)

Client	Server
	<pre> =====> SampleServerInterceptor id MyServerInterceptor preinvoke => open Created john's account: Stub[repository_id=IDL:Bank/ Account:1.0,key=TransientId[poaName=/,id={4 bytes: (0) (0) (0) (0)},sec=0,usec=0]] </pre>
<pre> =====> SampleClientInterceptor id MyClientInterceptor postinvoke =====> SampleBindInterceptor bind =====> SampleBindInterceptor bind_succeeded =====> SampleClientInterceptor id MyClientInterceptor preinvoke_premarshal => balance =====> SampleClientInterceptor id MyClientInterceptor preinvoke_postmarshal </pre>	<pre> =====> SampleServerInterceptor id MyServerInterceptor postinvoke_premarshal =====> SampleServerInterceptor id MyServerInterceptor postinvoke_postmarshal </pre>
<pre> =====> SampleClientInterceptor id MyClientInterceptor postinvoke The balance in john's account is \$245.64 </pre>	

Since the OAD is not running, the `bind()` call fails and the server proceeds. The client binds to the account object, and then calls the `balance()` method. This request is received by the server, processed, and results are returned to the client. The client prints the results.

As demonstrated by the example code and results, the interceptors for both the client and server are installed when the respective process starts. Information about registering an interceptor is covered in “Registering interceptors with the VisiBroker ORB” on page 24-6.

ServiceResolverInterceptor example

The code below provides an example of how to implement a `ServiceLoader` interface:

```

import com.inprise.vbroker.properties.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.InterceptorExt.*;

public final class UtilityServiceLoader implements ServiceLoader,
    ServiceResolverInterceptor {
    private com.inprise.vbroker.orb.ORB _orb = null;
    private String[] _serviceNames = { "TimeService", "WeatherService" };

    public void init(org.omg.CORBA.ORB orb) {
        // Just in case they are needed by resolve()
    }

```

Example interceptors

```
_orb = (com.inprise.vbroker.orb.ORB) orb;

PropertyManager pm = _orb.getPropertyManager();
// use the PropertyManager to query property settings
// if needed (not used in this example)

/**** Installing the Initial Reference *****/
InterceptorManagerControl control = _orb.interceptorManager();
ServiceResolverInterceptorManager manager =
    (ServiceResolverInterceptorManager)control.get_manager("ServiceResolver");
for (int i = 0; i < _serviceNames.length; i++) {
    manager.add(_serviceNames[i], this);
}
/**** end of installation ***/

if (_orb.debug)
    _orb.println("UtilityServices package has been initialized");
}

public void init_complete(org.omg.CORBA.ORB orb) {
    // can be used for post-initialization processing if desired
}

public void shutdown(org.omg.CORBA.ORB orb) {
    _orb = null;
    _serviceNames = null;
}

public org.omg.CORBA.Object resolve(java.lang.String service) {
    org.omg.CORBA.Object srv = null;
    byte[] serviceId = service.getBytes();
    try {
        if (service == "TimeService") {
            srv = UtilityServices.TimeServiceHelper.bind(_orb, "/time_service_poa", serviceId);
        }
        else if (service == "WeatherService") {
            srv = UtilityServices.WeatherServiceHelper.bind(_orb, "/weather_service_poa",
                serviceId);
        }
    } catch (org.omg.CORBA.SystemException e) {
        if (_orb.debug)
            _orb.println("UtilityServices package resolve error: " + e);
        srv = null;
    }

    return srv;
}
}
```

Code listings

The `SampleServerLoader` object is responsible for loading `POALifeCycleInterceptor` class and instantiating an object. This class is linked to the ORB dynamically by

`vbroker.orb.dynamicLibs`. The `SampleServerLoader` class contains the `init()` method which is called by the ORB during initialization. Its sole purpose is to install a `POALifeCycleInterceptor` object by creating it and registering it with the `InterceptorManager`.

Code sample 24.8 SampleServerLoader.java

```

import java.util.*;
import com.inprise.vbroker.orb.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;

public class SampleServerLoader implements ServiceLoader {
    public void init(org.omg.CORBA.ORB orb) {
        try {
            InterceptorManagerControl control =
                InterceptorManagerControlHelper.narrow(
                    orb.resolve_initial_references("VisiBrokerInterceptorControl"));
            // Install a POA interceptor
            POALifeCycleInterceptorManager poa_manager =
                (POALifeCycleInterceptorManager) control.get_manager("POALifeCycle");
            poa_manager.add(new SamplePOALifeCycleInterceptor());
        } catch (Exception e) {
            e.printStackTrace();
            throw new org.omg.CORBA.INITIALIZE(e.toString());
        }
        System.out.println("=====>SampleServerLoader:Interceptors loaded");
    }
    public void init_complete(org.omg.CORBA.ORB orb) {
    }
    public void shutdown(org.omg.CORBA.ORB orb) {
    }
}

```

The `SamplePOALifeCycleInterceptor` object is invoked every time a POA is created or destroyed. Because we have two POAs in the `client_server` example, this interceptor is invoked twice, first during `rootPOA` creation and then at the creation of `myPOA`. We install the `SampleServerInterceptor` only at the creation of `myPOA`.

Code sample 24.9 SamplePOALifeCycleInterceptor.java

```

import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;
import com.inprise.vbroker.IOP.*;

public class SamplePOALifeCycleInterceptor implements POALifeCycleInterceptor {
    public void create(org.omg.PortableServer.POA poa,
        org.omg.CORBA.PolicyListHolder policies_holder,
        IORValueHolder iorTemplate,
        InterceptorManagerControl control) {
        if (poa.the_name().equals("bank_agent_poa")) {
            // Add the Request-level interceptor
            SampleServerInterceptor interceptor =
                new SampleServerInterceptor("MyServerInterceptor");
            // Get the IORCreation interceptor manager
            ServerRequestInterceptorManager manager =
                (ServerRequestInterceptorManager) control.get_manager("ServerRequest");
            // Add the interceptor
            manager.add(interceptor);
            System.out.println("=====>In POA " + poa.the_name() +
                ", 1 ServerRequest interceptor installed");
        }
    }
}

```



```

    } else
        System.out.println("=====>In POA " + poa.the_name() + ". Nothing to do.");
    }
    public void destroy(org.omg.PortableServer.POA poa) {
        // To be a trace!
        System.out.println("=====> SamplePOALifeCycleInterceptor destroy");
    }
}

```

The `SampleServerInterceptor` object is invoked every time a request is received at or a reply is made by the server.

Code sample 24.10 `SampleServerInterceptor.java`

```

import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.IOP.*;
import com.inprise.vbroker.CORBA.portable.*;

public class SampleServerInterceptor implements ServerRequestInterceptor {
    private String _id;
    public SampleServerInterceptor(String id) {
        _id = id;
    }
    public void preinvoke(org.omg.CORBA.Object target,
        String operation,
        ServiceContext[] service_contexts,
        InputStream payload,
        Closure closure) {
        // Put the _id of this ServerRequestInterceptor into the closure object
        closure.object = new String(_id);
        System.out.println("=====> SampleServerInterceptor id " +
            closure.object + " preinvoke => " + operation);
    }
    public void postinvoke_premarshal(org.omg.CORBA.Object target,
        ServiceContextListHolder service_contexts_holder,
        org.omg.CORBA.Environment env,
        Closure closure) {
        System.out.println("=====> SampleServerInterceptor id " +
            closure.object + " postinvoke_premarshal");
    }
    public void postinvoke_postmarshal(org.omg.CORBA.Object target,
        OutputStream payload,
        Closure closure) {
        System.out.println("=====> SampleServerInterceptor id " +
            closure.object + " postinvoke_postmarshal");
    }
    public void exception_occurred(org.omg.CORBA.Object target,
        org.omg.CORBA.Environment env,
        Closure closure) {
        System.out.println("=====> SampleServerInterceptor id " +
            closure.object + " exception_occurred");
    }
}

```

The `SampleClientInterceptor` is invoked every time a request is made by or a reply is received at the client.

Code sample 24.11 `SampleClientInterceptor.java`

```
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.IOP.*;
import com.inprise.vbroker.CORBA.portable.*;

public class SampleClientInterceptor implements ClientRequestInterceptor {
    private String _id;
    public SampleClientInterceptor(String id) {
        _id = id;
    }
    public void preinvoke_premarshal(org.omg.CORBA.Object target,
        String operation,
        ServiceContextListHolder service_contexts_holder,
        Closure closure) {
        // Put the _id of this ClientRequestInterceptor into the closure object
        closure.object = new String(_id);
        System.out.println("=====> SampleClientInterceptor id " +
            closure.object +
            " preinvoke_premarshal => " + operation);
    }
    public void preinvoke_postmarshal(org.omg.CORBA.Object target,
        OutputStream payload,
        Closure closure) {
        System.out.println("=====> SampleClientInterceptor id " +
            closure.object + " preinvoke_postmarshal");
    }
    public void postinvoke(org.omg.CORBA.Object target,
        ServiceContext[] service_contexts,
        InputStream payload,
        org.omg.CORBA.Environment env,
        Closure closure) {
        System.out.println("=====> SampleClientInterceptor id " +
            closure.object + " postinvoke");
    }
    public void exception_occurred(org.omg.CORBA.Object target,
        org.omg.CORBA.Environment env,
        Closure closure) {
        System.out.println("=====> SampleClientInterceptor id " +
            closure.object + " exception_occured");
    }
}
```

The loader responsible for loading `BindInterceptor` objects. This class is linked to the ORB dynamically by `vbroker.orb.dynamicLibs`. The `SampleClientLoader` class contains the `bind()` and `bind_succeeded()` methods. These methods are called by the ORB during object binding. When the bind succeeds, `bind_succeeded()` will be called by the ORB and a `BindInterceptor` object is installed by creating it and registering it the `InterceptorManager`.

Code sample 24.12 SampleClientLoader.java

```

import java.util.*;
import com.inprise.vbroker.orb.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;

public class SampleClientLoader implements ServiceLoader {
    public void init(org.omg.CORBA.ORB orb) {
        try {
            InterceptorManagerControl control =
                InterceptorManagerControlHelper.narrow(
                    orb.resolve_initial_references("VisiBrokerInterceptorControl"));
            BindInterceptorManager bind_manager =
                (BindInterceptorManager) control.get_manager("Bind");
            bind_manager.add(new SampleBindInterceptor());
        } catch (Exception e) {
            e.printStackTrace();
            throw new org.omg.CORBA.INITIALIZE(e.toString());
        }
        System.out.println("Bind Interceptors loaded");
    }
    public void init_complete(org.omg.CORBA.ORB orb) {
    }
    public void shutdown(org.omg.CORBA.ORB orb) {
    }
}

```

The `SampleBindInterceptor` is invoked when the client attempts to bind to an object. The first step on the client side after ORB initialization is to bind to an `AccountManager` object. This bind invokes the `SampleBindInterceptor` and a `SampleClientInterceptor` is installed when the bind succeeds.

Code sample 24.13 SampleBindInterceptor.java

```

import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.IOP.*;

public class SampleBindInterceptor implements BindInterceptor {
    public IORValue bind(IORValue ior, org.omg.CORBA.Object target,
        boolean rebind, Closure closure) {
        // To be a trace!
        System.out.println("=====> SampleBindInterceptor bind");
        return null;
    }
    public IORValue bind_failed(IORValue ior, org.omg.CORBA.Object target,
        Closure closure) {
        // To be a trace!
        System.out.println("=====> SampleBindInterceptor bind_failed");
        return null;
    }
    public void bind_succeeded(IORValue ior, org.omg.CORBA.Object target,
        int Index, InterceptorManagerControl control,
        Closure closure) {
        // To be a trace!
        System.out.println("=====> SampleBindInterceptor bind_succeeded");
    }
}

```

```
// Create the Client Request interceptor:
SampleClientInterceptor interceptor =
new SampleClientInterceptor("MyClientInterceptor");
// Get the manager
ClientRequestInterceptorManager manager =
    (ClientRequestInterceptorManager)control.get_manager("ClientRequest");
// Add CRQ to the list:
manager.add(interceptor);
}
public void exception_occurred(IORValue ior, org.omg.CORBA.Object target,
    org.omg.CORBA.Environment env,
    Closure closure) {
    // To be a trace!
    System.out.println("=====> SampleBindInterceptor exception_occured");
}
}
```

Passing information between your interceptors

Closure objects are created by the ORB at the beginning of certain sequences of interceptor calls. The same Closure object is used for all calls in that particular sequence. The Closure object contains a single public data field, `object`, of type `java.lang.Object` which may be set by the interceptor to keep state information. The sequences for which Closure objects are created vary depending on the interceptor type. In the `ClientRequestInterceptor`, a new Closure is created before calling `preinvoke_premarshal` and the same Closure is used for that request until the request completes, successfully or not. Likewise in the `ServerInterceptor`, a new Closure is created before calling `preinvoke`, and that Closure is used for all interceptor calls related to processing that particular request.

For an example of how Closure is used, see the `interceptors/client_server` directory in the `examples` directory in your VisiBroker for Java installation.

The Closure object can be cast to `ExtendedClosure` to obtain `response_expected` and `request_id` as follows:

```
int my_response_expected = ((ExtendedClosure)closure) . reqInfo.response_expected;
int my_request_id = ((ExtendedClosure)closure) . reqInfo.request_id;
```

Using object wrappers

This chapter describes the object wrapper feature of VisiBroker, which allows your applications to be notified or to trap an operation request for an object.

Overview

VisiBroker's object wrapper feature allows you to define methods that are called when a client application invokes a method on a bound object or when a server application receives an operation request. Unlike the interceptor feature described in Chapter 24 which is invoked at the ORB level, object wrappers are invoked before an operation request has been marshalled. In fact, you can design object wrappers to return results without the operation request having ever been marshalled, sent across the network, or actually presented to the object implementation.

Object wrappers may be installed on just the client-side, just the server-side, or they may be installed in both the client and server portions of a single application.

Here are a few examples of how you might use object wrappers in your application,

- Log information about the operation requests issued by a client or received by a server.
- Measure the time required for operation requests to complete.
- Cache the results of frequently issued operation requests so results can be immediately returned, without actually contacting the object implementation each time.

Note Externalizing a reference to an object for which object wrappers have been installed, using the ORB Object's `object_to_string` method, will not propagate those wrappers to the recipient of the stringified reference if the recipient is a different process.

Typed and un-typed object wrappers

VisiBroker offers two kinds of object wrappers; typed and un-typed. You can mix the use of both of these types of wrappers within a single application. For information on typed wrappers, see “Typed object wrappers” on page 25-8. Table 25.1 summarizes the important distinctions between these two types of object wrappers.

Table 25.1 Comparison of features for typed and un-typed object wrappers

Features	Typed	Un-typed
Receives all arguments that are to be passed to the stub.	Yes	No
Can return control to the caller without actually invoking the next wrapper, the stub, or the object implementation.	Yes	No
Will be invoked for all operation requests for all objects.	No	Yes

Special idl2java requirements

Whenever you plan to use typed or un-typed object wrappers, you must ensure that you use the `-obj_wrapper` option with the `idl2java` compiler when you generate the code for your applications. This will result in the generation of

- Object wrapper base class for each of your interfaces
- Additional Helper class methods for adding or removing object wrappers.

Example applications

The `examples/interceptors/objectWrappers` directory contains three sample client and server applications that will be used to illustrate both the typed and untyped object wrapper concepts in this chapter.

Un-typed object wrappers

Un-typed object wrappers allow you to define methods that are to be invoked before an operation request is processed, after an operation request is processed, or both. Un-typed wrappers can be installed for client or server applications and you can also install multiple versions.

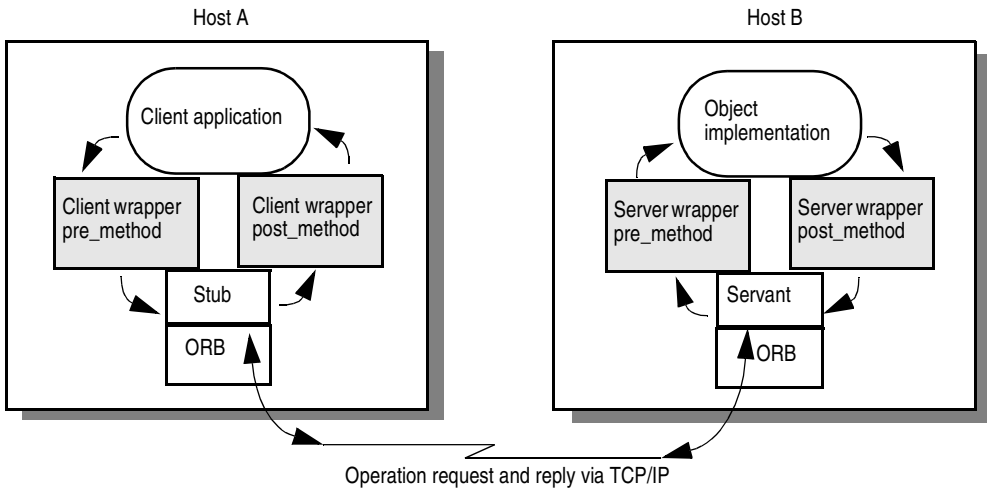
You may also mix the use of both typed and un-typed object wrappers within the same client or server application.

By default, un-typed object wrappers have a global scope and will be invoked for any operation request. You can design un-typed wrappers so that they have no effect for operation requests on object types in which you are not interested.

Note Unlike typed object wrappers, un-typed wrapper methods do not receive the arguments that the stub or object implementation would receive nor can they prevent the invocation of the stub or object implementation.

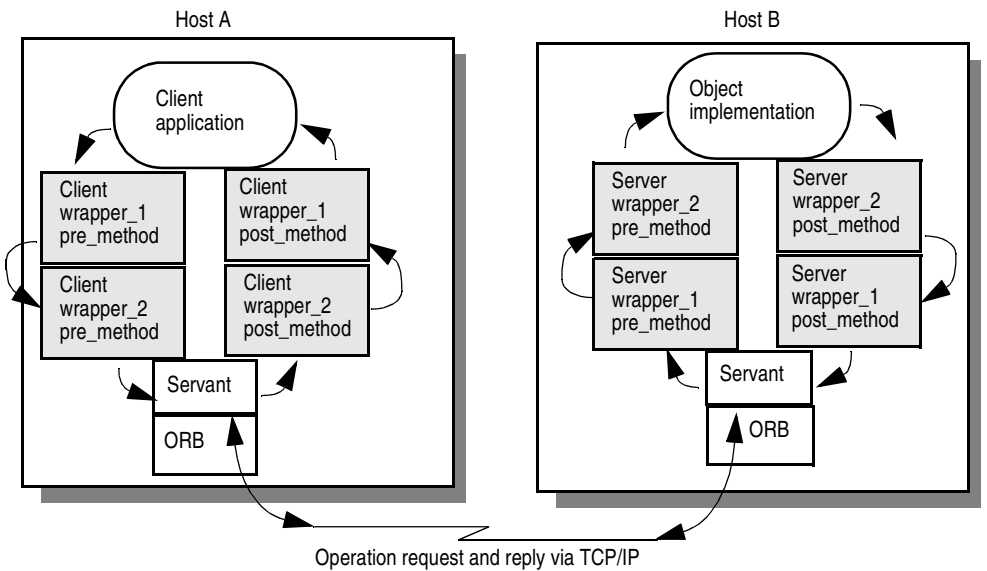
Figure 25.1 shows how an un-typed object wrapper's `pre_method` is invoked before the client stub method and how the `post_method` is invoked afterward. It also shows the calling sequence on the server-side with respect to the object implementation.

Figure 25.1 Single un-typed object wrapper



Using multiple, un-typed object wrappers

Figure 25.2 Multiple un-typed object wrappers



Order of pre_method invocation

When a client invokes a method on a bound object, each un-typed object wrapper `pre_method` will receive control before the client's stub routine is invoked. When a server receives an operation request, each un-typed object wrapper `pre_method` will be invoked before the object implementation receives control. In both cases, the first `pre_method` to receive control will be the one belonging to the object wrapper that was *registered first*.

Order of post_method invocation

When a server's object implementation completes its processing, each `post_method` will be invoked before the reply is sent to the client. When a client receives a reply to an operation request, each `post_method` will be invoked before control is returned to the client. In both cases, the first `post_method` to receive control will be the one belonging to the object wrapper that was *registered last*.

Note If you choose to use both typed and un-typed object wrappers, see "Combined use of un-typed and typed object wrappers" on page 25-14 for information on the invocation order.

Using un-typed object wrappers

You must use the following steps when using un-typed object wrappers. Each step is discussed, in turn.

- 1 Identify the interface, or interfaces, for which you want to create a un-typed object wrapper.
- 2 Generate the code from your IDL specification using the `idl2java` compiler using the `-obj_wrapper` option.
- 3 Create an implementation for your un-typed object wrapper factory, derived from the `UntypedObjectWrapperFactory` class.
- 4 Create an implementation for your un-typed object wrapper, derived from the `UntypedObjectWrapper` class.
- 5 Modify your client or server application to access the appropriate type of `ChainUntypedObjectWrapperFactory`.
- 6 Modify your application to create your un-typed object wrapper factory.
- 7 Use the `ChainUntypedObjectWrapperFactory`'s `add` method to add your factory to the chain.

Implementing an un-typed object wrapper factory

The implementation of the `TimingUntypedObjectWrapperFactory`, part of the `objectWrappers` sample applications, shows how to define an un-typed object wrapper

factory, derived from the `UntypedObjectWrapperFactory`. Your factory's `create` method will be invoked to create an un-typed object wrapper whenever a client binds to an object or a server invokes a method on an object implementation. The `create` method receives the target object, which allows you to design your factory to not create an un-typed object wrapper for those object types you wish to ignore. It also receives an enum specifying whether the object wrapper created is for the server-side object implementation or the client-side object.

Code sample 25.1 `TimingUntypedObjectWrapperFactory` implementation

```
package UtilityObjectWrappers;
import com.inprise.vbroker.interceptor.*;

public class TimingUntypedObjectWrapperFactory implements
    UntypedObjectWrapperFactory {
    public UntypedObjectWrapper create(org.omg.CORBA.Object target,
        com.inprise.vbroker.interceptor.Location loc) {
        return new TimingUntypedObjectWrapper();
    }
}
```

Implementing an un-typed object wrapper

The following code sample shows the implementation of the `TimingObjectWrapper`. Your un-typed wrapper must be derived from the `UntypedObjectWrapper` class, and you may provide an implementation for both the `pre_method` or `post_method` methods in your un-typed object wrapper.

Once your factory has been installed, either automatically by the factory's constructor or manually by invoking the `ChainUntypedObjectWrapper::add` method, an un-typed object wrapper object will be created automatically whenever your client binds to an object or when your server invokes a method on an object implementation.

The `pre_method` shown in the following code sample obtains the current time, saves it in a private variable, and prints a message. The `post_method` also obtains the current time, determines how much time that has elapsed since the `pre_method` was called, and prints the elapsed time.

Code sample 25.2 `TimingUntypedObjectWrapper` implementation

```
package UtilityObjectWrappers;
import com.inprise.vbroker.interceptor.*;

public class TimingUntypedObjectWrapper implements UntypedObjectWrapper {
    private long time;
    public void pre_method(String operation,
        org.omg.CORBA.Object target,
        Closure closure) {
        System.out.println("Timing: " +
            ((com.inprise.vbroker.CORBA.Object) target)._object_name() + "->"
            + operation + "()");
        time = System.currentTimeMillis();
    }
}
```

```
public void post_method(String operation,
    org.omg.CORBA.Object target,
    org.omg.CORBA.Environment env,
    Closure closure) {
    long diff = System.currentTimeMillis() - time;
    System.out.println("Timing: Time for call \"t\" + ((com.inprise.vbroker.CORBA.Object)
        target)._object_name() + \"->\" + operation + \"() = \" + diff + \" ms.\");
}
```

pre_method and post_method parameters

Both the pre_method and post_method receive these parameters:

Table 25.2 Common arguments for the pre_method and post_method methods

Parameter	Description
operation	Name of the operation that was requested on the target object.
target	Target object.
closure	Area where data can be saved across method invocations for this wrapper.

The post_method also receives an Environment parameter, which can be used to inform the user of any exceptions that might have occurred during the previous steps of the method invocation.

Creating and registering un-typed object wrapper factories

On the client side, objects will be wrapped only if untyped object wrapper factories are created and registered before the objects are bound. On the server side, untyped object wrappers factories which are created and registered before an object implementation is called.

The following code shows a portion of the sample file UntypedClient.java, which shows the creation and installation of two un-typed object wrapper factories for a client. The factories are created after the ORB has been initialized, but before the client binds to any objects.

Code sample 25.3 Installing two client-side, un-typed object wrapper factories

```
// UntypedClient.java
import com.inprise.vbroker.interceptor.*;

public class UntypedClient {
    public static void main(String[] args) throws Exception {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        doMain (orb, args);
    }
}
```

```

public static void doMain(org.omg.CORBA.ORB orb, String[] args) throws Exception {
    ChainUntypedObjectWrapperFactory Cfactory =
        ChainUntypedObjectWrapperFactoryHelper.narrow(
            orb.resolve_initial_references("ChainUntypedObjectWrapperFactory")
        );
    Cfactory.add(new UtilityObjectWrappers.TimingUntypedObjectWrapperFactory(),
        Location.CLIENT);
    Cfactory.add(new UtilityObjectWrappers.TracingUntypedObjectWrapperFactory(),
        Location.CLIENT);
    // Locate an account manager. . . .
}
}

```

The following code sample shows the sample file `UntypedServer.java`, which shows the creation and registration of un-typed object wrapper factories for a server. The factories are created after the ORB is initialized, but before any object implementations are created.

Code sample 25.4 Installing a server-side, un-typed object wrapper factory

```

// UntypedServer.java
import com.inprise.vbroker.interceptor.*;
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValue;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValueHelper;
import com.inprise.vbroker.PortableServerExt.BIND_SUPPORT_POLICY_TYPE;

public class UntypedServer {
    public static void main(String[] args) throws Exception {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        ChainUntypedObjectWrapperFactory Sfactory =
            ChainUntypedObjectWrapperFactoryHelper.narrow
                (orb.resolve_initial_references("ChainUntypedObjectWrapperFactory"));
        Sfactory.add(new UtilityObjectWrappers.TracingUntypedObjectWrapperFactory(),
            Location.SERVER);
        // get a reference to the root POA
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        // Create a BindSupport Policy that makes POA register each servant
        // with osagent
        org.omg.CORBA.Any any = orb.create_any();
        BindSupportPolicyValueHelper.insert(any, BindSupportPolicyValue.BY_INSTANCE);
        org.omg.CORBA.Policy bsPolicy =
            orb.create_policy(BIND_SUPPORT_POLICY_TYPE.value, any);
        // Create policies for our testPOA
        org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy
                (LifespanPolicyValue.PERSISTENT), bsPolicy
        };
        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA( "bank_agent_poa",
            rootPOA.the_POAManager(),
            policies );
    }
}

```

```
// Create the account manager object.
AccountManagerImpl managerServant = new AccountManagerImpl();
// Decide on the ID for the servant
byte[] managerId = "BankManager".getBytes();
// Activate the servant with the ID on myPOA
myPOA.activate_object_with_id(managerId, managerServant);
// Activate the POA manager
rootPOA.the_POAManager().activate();
System.out.println("AccountManager: BankManager is ready.");
for( int i = 0; i < args.length; i++ ) {
    if( args[i].equalsIgnoreCase("-runCoLocated") ) {
        if( args[i+1].equalsIgnoreCase("Client") ){
            Client.doMain(orb, new String[0]);
        } else if( args[i+1].equalsIgnoreCase("TypedClient") ){
            TypedClient.doMain(orb, new String[0]);
        }
        if( args[i+1].equalsIgnoreCase("UntypedClient") ){
            UntypedClient.doMain(orb, new String[0]);
        }
        System.exit(1);
    }
}
// Wait for incoming requests
orb.run();
}
```

Removing un-typed object wrappers

The `ChainUntypedObjectWrapperFactory` class' `remove` method can be used to remove an un-typed object wrapper factory from a client or server application. You must specify a location when removing a factory. This means that if you have added a factory with a location of `Both`, you can selectively remove it from the `Client` location, the `Server` location, or `Both`.

Note Removing one or more object wrapper factories from a client will not affect objects of that class that are already bound by the client. Only subsequently bound objects will be affected. Removing object wrapper factories from a server will not affect object implementations that have already been created. Only subsequently created object implementations will be affected.

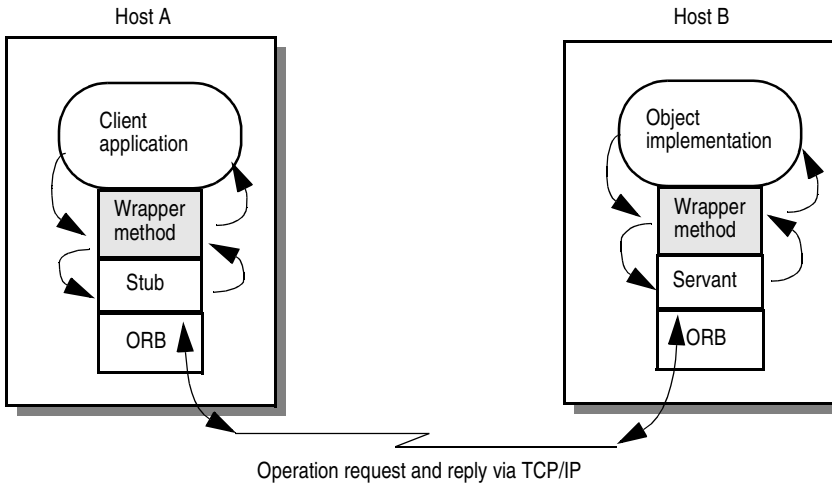
Typed object wrappers

When you implement a typed object wrapper for a particular class, you define the processing that is to take place when a method is invoked on a bound object. Figure 25.3 shows how an object wrapper method on the client is invoked before the client stub class method and how an object wrapper on the server-side is invoked before the server's implementation method.

Note Your typed object wrapper implementation is not required to implement all methods offered by the object it is wrapping.

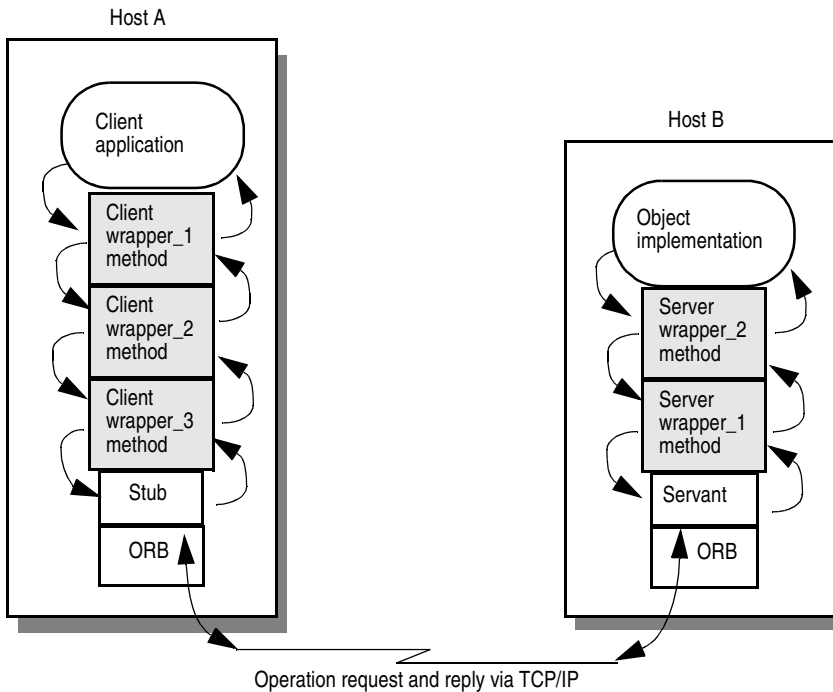
You may also mix the use of both typed and un-typed object wrappers within the same client or server application. For more information, see “Combined use of un-typed and typed object wrappers” on page 25-14.

Figure 25.3 Single typed object wrapper registered



Using multiple, typed object wrappers

You may implement and register more than one typed object wrapper for a particular class of object, as shown in Figure 25.4. On the client side, the first object wrapper registered is `client_wrapper_1`, so its methods will be the first to receive control. After performing its processing, the `client_wrapper_1` method may pass control to the next object's method in the chain or it may return control to the client. On the server side, the first object wrapper registered is `server_wrapper_1`, so its methods will be the first to receive control. After performing its processing, the `server_wrapper_1` method may pass control to the next object's method in the chain or it may return control to the servant.

Figure 25.4 Multiple, typed object wrappers registered

Order of invocation

The methods for a typed object wrapper that are register for a particular class will receive all of the arguments that are normally passed to the stub method on the client side or to skeleton on the server side. Each object wrapper method can pass control to the next wrapper method in the chain by invoking the parent class' method, `super.<method_name>`. If an object wrapper wishes to return control without calling the next wrapper method in the chain, it can `return` with the appropriate return value.

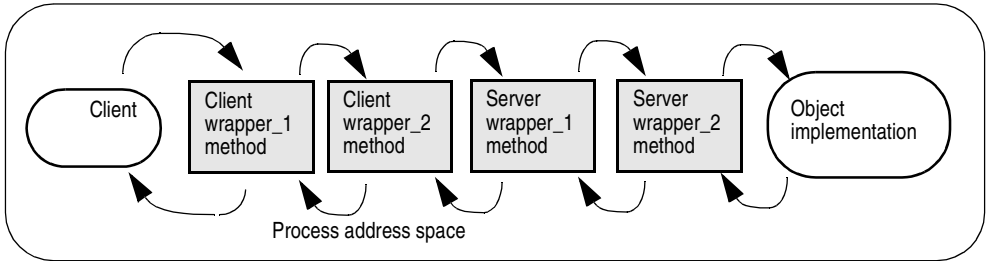
A typed object wrapper method's ability to return control to the previous method in the chain allows you to create a wrapper method that never invokes a client stub or object implementation. For example, you can create an object wrapper method that caches the results of a frequently requested operation. In this scenario, the first invocation of a method on the bound object results in an operation request being sent to the object implementation. As control flows back through the object wrapper method, the result is stored. On subsequent invocations of the same method, the object wrapper method can simply return the cached result without actually issuing the operation request to the object implementation.

If you choose to use both typed and un-typed object wrappers, see "Combined use of un-typed and typed object wrappers" on page 25-14 for information on the invocation order.

Typed object wrappers with co-located client and servers

When the client and server are both packaged in the same process, the first object wrapper method to receive control will belong to the first client-side object wrapper that was installed. Figure 25.5 illustrates the invocation order.

Figure 25.5 Typed object wrapper invocation order



Using typed object wrappers

You must use the following steps when using typed object wrappers. Each step is discussed in turn.

- 1 Identify the interface, or interfaces, for which you want to create a typed object wrapper.
- 2 Generate the code from your IDL specification using the `idl2java` compiler using the `-obj_wrapper` option.
- 3 Derive your typed object wrapper class from the `<interface_name>ObjectWrapper` class generated by the `idl2java` compiler and provide an implementation of those methods you wish to wrap.
- 4 Modify your application to register the typed object wrapper.

Implementing typed object wrappers

You derive typed object wrappers from the `<interface_name>ObjectWrapper` class that is generated by the `idl2java` compiler. The following code shows the implementation of a typed object wrapper for the `Account` interface. Notice that this class is derived from the `AccountObjectWrapper` interface and provides a simple caching implementation of the `balance` method, which provides these processing steps:

- 1 Check the `_initialized` flag to see if this method has been invoked before.
- 2 If this is the first invocation, the `balance` method on the next object in the chain is invoked and the result is saved to `_balance`, `_initialized` is set to `true`, and the value is returned.
- 3 If this method has been invoked before, simply return the cached value.

Code sample 25.5 Portion of the `CachingAccountObjectWrapper` implementation

```

package BankWrappers;
public class CachingAccountObjectWrapper extends Bank.AccountObjectWrapper {
    private boolean _initialized = false;
    private float _balance;
    public float balance() {
        System.out.println("+ CachingAccountObjectWrapper: Before calling balance: ");
        try {
            if( !_initialized ) {
                _balance = super.balance();
                _initialized = true;
            } else {
                System.out.println("+ CachingAccountObjectWrapper: Returning Cached value");
            }
            return _balance;
        } finally {
            System.out.println("+ CachingAccountObjectWrapper: After calling balance: ");
        }
    }
}

```

Registering typed object wrappers for a client

A typed object wrapper is registered on the client-side by invoking the `addClientObjectWrapperClass` method that is generated for the class by the `idl2java` compiler. Client-side object wrappers must be registered after the `ORB.init` method has been called, but before any objects are bound. The following code shows a portion of the `TypedClient.java` file that creates and registers a typed object wrapper.

Code sample 25.6 Installing a client-side, typed object wrapper

```

// TypedClient.java
import com.inprise.vbroker.interceptor.*;
public class TypedClient {
    public static void main(String[] args) throws Exception {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        doMain (orb, args);
    }
    public static void doMain(org.omg.CORBA.ORB orb, String[] args) {
        // Add a typed object wrapper for Account objects
        Bank.AccountHelper.addClientObjectWrapperClass(orb,
            BankWrappers.CachingAccountObjectWrapper.class);
        // Locate an account manager.
        Bank.AccountManager manager =
            Bank.AccountManagerHelper.bind(orb, "BankManager");
        . . .
    }
}

```

The ORB keeps track of any object wrappers that have been registered for it on the client-side. When a client invokes the `_bind` method to bind to an object of that type, the necessary object wrappers will be created. If a client binds to more than one

instance of a particular class of object, each instance will have its own set of wrappers.

Registering typed object wrappers for a server

As with a client application, a typed object wrapper is registered on the server-side by invoking the `addServerObjectWrapperClass` method offered by the `Helper` class. Server-side, typed object wrappers must be registered after the `ORB.init` method has been called, but before an object implementation services a request. The following code shows a portion of the `TypedServer.java` file that installs a typed object wrapper.

Code sample 25.7 Installing a server-side, typed object wrapper

```
// TypedServer.java
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValue;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValueHelper;
import com.inprise.vbroker.PortableServerExt.BIND_SUPPORT_POLICY_TYPE;

public class TypedServer {
    public static void main(String[] args) throws Exception {
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // Add two typed object wrappers for AccountManager objects
        Bank.AccountManagerHelper.addServerObjectWrapperClass(orb,
            BankWrappers.SecureAccountManagerObjectWrapper.class);
        Bank.AccountManagerHelper.addServerObjectWrapperClass(orb,
            BankWrappers.CachingAccountManagerObjectWrapper.class);
        // get a reference to the root POA
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        // Create a BindSupport Policy that makes POA register each servant
        // with osagent
        org.omg.CORBA.Any any = orb.create_any();
        BindSupportPolicyValueHelper.insert(any, BindSupportPolicyValue.BY_INSTANCE);
        org.omg.CORBA.Policy bsPolicy =
            orb.create_policy(BIND_SUPPORT_POLICY_TYPE.value, any);
        // Create policies for our testPOA
        org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT), bsPolicy
        };
        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA( "lilo", rootPOA.the_POAManager(), policies );
        // Create the account manager object.
        AccountManagerImpl managerServant = new AccountManagerImpl();
        // Decide on the ID for the servant
        byte[] managerId = "BankManager".getBytes();
        // Activate the servant with the ID on myPOA
        myPOA.activate_object_with_id(managerId, managerServant);
        // Activate the POA manager
        rootPOA.the_POAManager().activate();
        System.out.println("AccountManager: BankManager is ready.");
    }
}
```

```
for( int i = 0; i < args.length; i++ ) {
    if ( args[i].equalsIgnoreCase("-runCoLocated") ) {
        if( args[i+1].equalsIgnoreCase("Client") ){
            Client.doMain(orb, new String[0]);
        } else if( args[i+1].equalsIgnoreCase("TypedClient") ){
            TypedClient.doMain(orb, new String[0]);
        }
        if( args[i+1].equalsIgnoreCase("UntypedClient") ){
            UntypedClient.doMain(orb, new String[0]);
        }
        System.exit(1);
    }
}
// Wait for incoming requests
orb.run();
}
```

If a server creates more than one instance of a particular class of object, a set of wrappers will be created for each instance.

Removing typed object wrappers

The `Helper` class also provides methods for removing a typed object wrapper from a client or server application. See the *VisiBroker for Java Reference* for more information.

Note Removing one or more object wrappers from a client will not affect objects of that class that are already bound by the client. Only subsequently bound objects will be affected. Removing object wrappers from a server will not affect object implementations that have already serviced requests. Only subsequently created object implementations will be affected.

Combined use of un-typed and typed object wrappers

If you choose to use both typed and un-typed object wrappers in your application, all `pre_method` methods defined for the un-typed wrappers will be invoked prior to any typed object wrapper methods defined for an object. Upon return, all typed object wrapper methods defined for the object will be invoked prior to any `post_method` methods defined for the un-typed wrappers.

The sample applications `Client.java` and `Server.java` make use of a sophisticated design that allows you to use command-line properties to specify which, if any, typed and un-typed object wrappers are to be used.

Command-line arguments for typed wrappers

The typed wrappers may be enabled by specifying the following on the command-line:

- 1 `-Dvbroker.orb.dynamicLibs=BankWrappers.Init`
- 2 Using the one or more of the properties summarized Table 25.3.

Table 25.3 Command-line properties for enabling or disabling BankWrappers

BankWrappers properties	Description
<code>-DCachingAccount[=<client server>]</code>	Installs a typed object wrapper that caches the results of the <code>balance</code> method for a client or a server. If no value for sub-property is specified, both the client and server wrappers are installed.
<code>-DCachingAccountManager[=<client server>]</code>	Installs a typed object wrapper that caches the results of the <code>open</code> method for a client or a server. If no value for the sub-property is specified, both the client and server wrappers are installed.
<code>-DSecureAccountManager[=<client server>]</code>	Installs a typed object wrapper that detects unauthorized users passed on the <code>open</code> method for a client or a server. If no value for sub-property is specified, both the client and server wrappers are installed.

Initializer for typed wrappers

The typed wrappers are defined in the `BankWrappers` package and include a service initializer, `BankWrappers/Init.java`, shown in the following code. This initializer will be invoked if you specify `-Dvbroker.orb.dynamicLibs=BankWrappers.Init` on the command-line when starting the client or server with `vbj`. Various typed object wrappers can be installed, based on the command-line properties you supply, which are summarized in Table 25.3.

Code sample 25.8 Service initializer for BankWrappers

```
package BankWrappers;
import java.util.*;
import com.inprise.vbroker.orb.ORB;
import com.inprise.vbroker.properties.PropertyManager;
import com.inprise.vbroker.interceptor.*;
public class Init implements ServiceLoader {
    com.inprise.vbroker.orb.ORB _orb;
    public void init(final org.omg.CORBA.ORB orb) {
        _orb = (ORB) orb;
        PropertyManager pm = _orb.getPropertyManager();
        // install my CachingAccountObjectWrapper
        String val = pm.getString("CachingAccount", this.toString());
        Class c = CachingAccountObjectWrapper.class;
        if (!val.equals(this.toString())) {
```

```

        if( val.equalsIgnoreCase("client") ) {
            Bank.AccountHelper.addClientObjectWrapperClass(orb, c);
        } else if( val.equalsIgnoreCase("server") ) {
            Bank.AccountHelper.addServerObjectWrapperClass(orb, c);
        } else {
            Bank.AccountHelper.addClientObjectWrapperClass(orb, c);
            Bank.AccountHelper.addServerObjectWrapperClass(orb, c);
        }
    }
}
// install my CachingAccountManagerObjectWrapper
val = pm.getString("CachingAccountManager", this.toString());
c = CachingAccountManagerObjectWrapper.class;
if( !val.equals(this.toString()) ) {
    if( val.equalsIgnoreCase("client") ){
        Bank.AccountManagerHelper.addClientObjectWrapperClass(orb, c);
    } else if( val.equalsIgnoreCase("server") ) {
        Bank.AccountManagerHelper.addServerObjectWrapperClass(orb, c);
    } else {
        Bank.AccountManagerHelper.addClientObjectWrapperClass(orb, c);
        Bank.AccountManagerHelper.addServerObjectWrapperClass(orb, c);
    }
}
}
// install my SecureAccountManagerObjectWrapper
val = pm.getString("SecureAccountManager",
    this.toString());
c = SecureAccountManagerObjectWrapper.class;
if( !val.equals(this.toString()) ) {
    if( val.equalsIgnoreCase("client") ){
        Bank.AccountManagerHelper.addClientObjectWrapperClass(orb, c);
    } else if( val.equalsIgnoreCase("server") ) {
        Bank.AccountManagerHelper.addServerObjectWrapperClass(orb, c);
    } else {
        Bank.AccountManagerHelper.addClientObjectWrapperClass(orb, c);
        Bank.AccountManagerHelper.addServerObjectWrapperClass(orb, c);
    }
}
}
}
public void init_complete(org.omg.CORBA.ORB orb) {}
public void shutdown(org.omg.CORBA.ORB orb) {}
}

```

Command-line arguments for un-typed wrappers

The un-typed wrappers may be enabled by specifying the following on the command-line:

1 -Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init

2 Using the one or more of the properties summarized Table 25.4.

Table 25.4 Command-line properties for enabling or disabling UtilityObjectWrappers

UtilityObjectWrappers properties	Description
-DTiming[=<client server>]	Installs an un-typed object wrapper that timing information for a client or a server. If no value for the sub-property is specified, both the client and server wrappers are installed.
-DTracing[=<client server>]	Installs an un-typed object wrapper that tracing information for a client or a server. If no value for the sub-property is specified, both the client and server wrappers are installed.

Initializers for un-typed wrappers

The un-typed wrappers are defined in the `UtilityObjectWrappers` package and include a service initializer, `UtilityObjectWrappers/Init.java`, shown below. This initializer will be invoked if you specify `-Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init` on the command-line when starting the client or server with `vbj`. Various un-typed object wrappers can be installed, based on the command-line properties you supply, which are summarized in Table 25.4.

Code sample 25.9 Service initializer for UtilityObjectWrappers

```
package UtilityObjectWrappers;
import java.util.*;
import com.inprise.vbroker.orb.ORB;
import com.inprise.vbroker.properties.PropertyManager;
import com.inprise.vbroker.interceptor.*;

public class Init implements ServiceLoader {
    com.inprise.vbroker.orb.ORB _orb;
    public void init(final org.omg.CORBA.ORB orb) {
        _orb = (ORB) orb;
        PropertyManager pm= _orb.getPropertyManager();
        try {
            ChainUntypedObjectWrapperFactory factory =
                ChainUntypedObjectWrapperFactoryHelper.narrow(
                    orb.resolve_initial_references("ChainUntypedObjectWrapperFactory"));
            // install my Timing ObjectWrapper
            String val = pm.getString("Timing", this.toString());
            if( !val.equals(this.toString())) {
                UntypedObjectWrapperFactory f= new TimingUntypedObjectWrapperFactory();
                if( val.equalsIgnoreCase("client") ){
                    factory.add(f, Location.CLIENT);
                } else if( val.equalsIgnoreCase("server") ) {
                    factory.add(f, Location.SERVER);
                } else {
                    factory.add(f, Location.BOTH);
                }
            }
        }
    }
}
```

```

// install my Tracing ObjectWrapper
val = pm.getString("Tracing", this.toString());
if( !val.equals(this.toString())) {
    UntypedObjectWrapperFactory f= new TracingUntypedObjectWrapperFactory();
    if( val.equalsIgnoreCase("client") ){
        factory.add(f, Location.CLIENT);
    } else if( val.equalsIgnoreCase("server") ) {
        factory.add(f, Location.SERVER);
    } else {
        factory.add(f, Location.BOTH);
    }
}
} catch( org.omg.CORBA.ORBPackage.InvalidName e ) {
    return;
}
}

public void init_complete(org.omg.CORBA.ORB orb) {}
public void shutdown(org.omg.CORBA.ORB orb) {}
}

```

Executing the sample applications

Before executing the sample applications, make sure that an osagent is running on your network. You can then execute the server application without any tracing or timing object wrappers by using the command

Example `prompt> vbj Server`

Note The server is designed as a co-located application. It implements both the server and a client.

From another window, you can execute the client application without any tracing or timing object wrappers to query the balance in a user's account using the command

Example `prompt> vbj Client John`

You can also execute this command if want a default name to be used.

Example `prompt> vbj Client`

Turning on timing and tracing object wrappers

To execute the client with un-typed timing and tracing object wrappers enabled, use this command:

Example `prompt> vbj -Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init -DTiming=client \
-DTracing=client Client John`

To execute the server with un-typed wrappers for timing and tracing enabled, use this command:

Example `prompt> vbj -Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init -DTiming=server\
-DTracing=server Server`

Turning on caching and security object wrappers

To execute the client with the typed wrappers for caching and security enabled, use this command:

Example

```
prompt> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init -DCachingAccount=client \
-DcachingAccountManager=client\
-DSecureAccountManager=client
Client John
```

To execute the server with typed wrappers for caching and security enabled, use this command:

Example

```
prompt>vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init -DCachingAccount=server \
-DcachingAccountManager=server \
-DSecureAccountManager=server \
Server
```

Turning on typed and un-typed wrappers

To execute the client with all typed and un-typed wrappers enabled, use this command:

Example

```
prompt> vbj -DOvbroker.orb.dynamicLibs=BankWrappers.Init, UtilityObjectWrappers.Init \
-DcachingAccount=client \
-DcachingAccountManager=client\
-DSecureAccountManager=client \
-DTiming=client \
-DTracing=client \
Client John
```

To execute the server with all typed and un-typed wrappers enabled, use this command:

Example

```
prompt>vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init, UtilityObjectWrappers.Init \
-DcachingAccount=server \
-DcachingAccountManager=server\
-DSecureAccountManager=server \
-DTiming=server \
-DTracing=server \
Server
```

Executing a co-located client and server

Specifying the `-runCoLocated` command-line option allows you to execute the client and server within the same process.

Property	Description
<code>-runCoLocated Client</code>	Executes the <code>Server.java</code> and the <code>Client.java</code> within the same process.
<code>-runCoLocated TypedClient</code>	Executes the <code>Server.java</code> and the <code>TypedClient.java</code> within the same process.
<code>-runCoLocated UntypedClient</code>	Executes the <code>Server.java</code> and the <code>UntypedClient.java</code> within the same process.

The following command will execute a co-located server and client with all typed wrappers enabled, the un-typed timing wrapper enables for just the client, and the un-typed tracing wrapper enabled for just the server, use this command,

Example `prompt> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init, UtilityObjectWrappers.Init \
 -DcachingAccount -DsecureAccountManager \
 -DTiming=client -DTracing=server \
 Server -runCoLocated Client`

Using RMI over IIOP

This chapter describes the VisiBroker tools which allow you to use RMI over IIOP.

Overview

RMI (remote method invocation) is a Java mechanism which allows objects to be created and used in a distributed environment. In this sense, RMI is an ORB, which is language-specific (Java) and non-CORBA compliant. The OMG has issued a specification, the Java language to IDL Mapping, which allows Java classes written using RMI to interoperate with CORBA objects using the IIOP encoding.

java2iiop and java2idl tools

VisiBroker has two compilers which allow you to adapt your existing Java classes to work with other objects using the VisiBroker ORB.

- The `java2iiop` compiler lets you adapt your RMI-compliant classes to use IIOP by generating all the proper skeleton, stub, and helper classes.
- The `java2idl` compiler generates IDL from your Java classes, allowing you to implement them in languages other than Java.

Using java2iiop

The `java2iiop` compiler lets you define interfaces and data types in Java, rather than IDL, that can then be used as interfaces and data types in CORBA. The compiler does not read Java source code (*java* files) or IDL, but Java *bytecode* (*class* files). The compiler then generates IIOP-compliant stubs and skeletons needed to do all the marshalling and communication required for CORBA.

Supported interfaces

When you run the `java2iiop` compiler, it generates the same files as if you had written the interface in IDL. All primitive data types like the numeric types (`short`, `int`, `long`, `float`, and `double`), `string`, CORBA objects or interface objects, `Any` objects, `typecode` objects are understood by the `java2iiop` compiler and mapped to the corresponding IDL types.

You can use `java2iiop` on any Java class or interface. For example, if a Java interface adheres to one of the following rules:

- Extends `java.rmi.Remote` and all of its methods throw `java.rmi.RemoteException`
- Extends `org.omg.CORBA.Object`

then, `java2iiop` will translate the interface to a CORBA interface in IDL.

Code sample 26.1 illustrates a Java RMI interface. The code example can be found in `examples/rmi-iiop` folder of the VisiBroker installation.

Code sample 26.1 Extending `java.rmi.Remote`

```
public interface Account extends java.rmi.Remote {
    String name() throws java.rmi.RemoteException;
    float getBalance() throws java.rmi.RemoteException;
    void setBalance(float bal) throws java.rmi.RemoteException;
}
```

Running java2iiop

You must compile your Java classes before you can use the `java2iiop` compiler. Once you have generated bytecode, you may run `java2iiop` to generate to generate client stubs, server skeletons, and the associated auxiliary files. For example, after running `java2iiop` on the `Account.class` file in the `examples/rmi-iiop/Bank` directory, you would have the following files:

- `_Account_Stub`
- `AccountHelper`
- `AccountHolder`
- `AccountPOA`
- `AccountPOATie`
- `AccountOperations`

For more details on these files, see Chapter 4, “Generated interfaces and classes,” in the VisiBroker for Java *Reference*.

Reverse mapping of Java classes to IDL

When mapping IDL interfaces to Java classes, using the `idl2java` compiler, interface name may use any of the generated classes suffixes (for example, `Helper`, `Holder`, `POA`, and so on), and the `idl2java` tool will handle the situation correctly by mangling the interface name (prefixing an underscore to the identifier). For example, if you define both a `Foo` and a `FooHolder` interface in IDL, `idl2java` will generate, amongst others, `Foo.java`, `FooHolder.java`, `_FooHolder.java`, and `_FooHolderHolder.java` files. On the other

hand, when generating IIOP-compliant Java classes from RMI Java classes, using the `java2iiop` compiler, the tool cannot generate the mangled classes. So, when declaring interfaces which use reserved suffixes, you cannot have them in the same package as the interface with the same name, (for example, you could not have a `Foo` and a `FooHolder` class in the same package when using the `java2iiop` compiler).

Completing the development process

After generating the associated files from your interfaces, you need to provide implementations for the interfaces. Follow these steps:

- 1 Create an implementation for the interface classes.
- 2 Compile your server class.
- 3 Write and compile your client code.
- 4 Start the Server program.
- 5 Run the Client program.

Note If you attempt to marshal a non-conforming class, an `org.omg.CORBA.MARSHAL: Cannot marshal non-conforming value of class <class name>` will be thrown. For instance, if you create the following two classes,

```
// This is a conforming class
public class Value implements java.io.Serializable {
    java.lang.Object any;
    ...
}

// This is a non-conforming class
public class Something {
    ...
}
```

and then attempt this,

```
Value val = new Value();
val.any = new Something();
```

You will raise an `org.omg.CORBA.MARSHAL` exception when you attempt to marshal `val`.

RMI-IIOP Bank example

The `Account` interface extends the `java.rmi.Remote` interface and is implemented by the `AccountImpl` class (see Code sample 26.2).

The `Client` class (see Code sample 26.3) first creates all the specified `Account` objects with the appropriate balances by creating `AccountData` objects for each account and passing them to the `AccountManager` to create the accounts. It then confirms that the balance is correct on the created account. The client then queries the `AccountManager` for a list of all the accounts, and proceeds to credit \$10.00 to each account. It then verifies if the new balance on the account is accurate.

Note The code example can be found in `examples/rmi-iiop` folder of the `VisiBroker` installation.

Code sample 26.2 Implementing the Account interface

```

public class AccountImpl extends Bank.AccountPOA {
    public AccountImpl(Bank.AccountData data) {
        _name = data.getName();
        _balance = data.getBalance();
    }
    public String name() throws java.rmi.RemoteException {
        return _name;
    }
    public float getBalance() throws java.rmi.RemoteException {
        return _balance;
    }
    public void setBalance(float balance) throws java.rmi.RemoteException {
        _balance = balance;
    }
    private float _balance;
    private String _name;
}

```

Code sample 26.3 Client class

```

public class Client {
    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // Get the manager Id
            byte[] managerId = "RMIBankManager".getBytes();
            // Locate an account manager. Give the full POA name and the servant ID.
            Bank.AccountManager manager =
                Bank.AccountManagerHelper.bind(orb, "/rmi_bank_poa", managerId);
            // Use any number of argument pairs to indicate name,balance of accounts to create
            if (args.length == 0 || args.length % 2 != 0) {
                args = new String[2];
                args[0] = "Jack B. Quick";
                args[1] = "123.23";
            }
            int i = 0;
            while (i < args.length) {
                String name = args[i++];
                float balance;
                try {
                    balance = new Float(args[i++]).floatValue();
                } catch (NumberFormatException n) {
                    balance = 0;
                }
                Bank.AccountData data = new Bank.AccountData(name, balance);
                Bank.Account account = manager.create(data);
                System.out.println("Created account for " + name
                    + " with opening balance of $" + balance);
            }
            java.util.Hashtable accounts = manager.getAccounts();
            for (java.util.Enumeration e = accounts.elements(); e.hasMoreElements();) {
                Bank.Account account =
                    Bank.AccountHelper.narrow((org.omg.CORBA.Object)e.nextElement());
            }
        }
    }
}

```

```

        String name = account.name();
        float balance = account.getBalance();
        System.out.println("Current balance in " + name + "'s account is $" + balance);
        System.out.println("Crediting $10 to " + name + "'s account.");
        account.setBalance(balance + (float)10.0);
        balance = account.getBalance();
        System.out.println("New balance in " + name + "'s account is $" + balance);
    }
} catch (java.rmi.RemoteException e) {
    System.err.println(e);
}
}
}

```

Supported data types

In addition to all of the Java primitive data types, RMI-IIOP supports a subset of Java classes.

Mapping primitive data types

Client stubs generated by `java2iiop` handle the marshalling of the Java primitive data types that represent an operation request so that they may be transmitted to the object server. When a Java primitive data type is marshalled, it must be converted into an IIOP-compatible format. Table 26.1 summarizes the mapping of Java primitive data types to IDL/IIOP types.

Table 26.1 Mapping Java types to IDL/IIOP

Java type	IDL/IIOP type
<code>void</code>	<code>void</code>
<code>boolean</code>	<code>boolean</code>
<code>byte</code>	<code>octet</code>
<code>char</code>	<code>char</code>
<code>short</code>	<code>short</code>
<code>int</code>	<code>long</code>
<code>long</code>	<code>long long</code>
<code>float</code>	<code>float</code>
<code>double</code>	<code>double</code>
<code>java.lang.String</code>	<code>CORBA::WStringValue</code>
<code>java.lang.Object</code>	<code>any</code>
<code>java.io.Serializable</code>	<code>any</code>
<code>java.io.Externalizable</code>	<code>any</code>

Mapping complex data types

This section discusses interfaces, arrays, and Java classes; it shows how the `java2iiop` compiler can be used to handle complex data types.

Interfaces

Java interfaces are represented in IDL as CORBA interfaces, and they must inherit from the `org.omg.CORBA.Object` interface. When passing objects that implement these interfaces, they are passed by reference.

Arrays

Another complex data type that may be defined in classes is an array. If you have an interface or definitions that use arrays, the arrays map to CORBA boxed sequence of boxed type.

Using the dynamically managed types

This chapter describes the `DynAny` feature of VisiBroker, which allows you to construct and interpret data types at runtime.

Overview

The `DynAny` interface provides a way to dynamically create basic and constructed data types at runtime. It also allows information to be interpreted and extracted from an `Any` object, even if the type it contains was not known to the server at compile-time. The use of the `DynAny` interface enables you to build powerful client and server applications that create and interpret data types at runtime.

Example client and server applications that illustrate the use of `DynAny` are included as part of the VisiBroker distribution. The examples are found in the `dynany` directory. The path is `inprise/vbroker/examples`. These example programs will be used to illustrate `DynAny` concepts in this chapter.

DynAny types

A `DynAny` object has an associated value that may either be a basic data type (such as `boolean`, `int`, or `float`) or a constructed data type. The `DynAny` interface, described in detail in the *VisiBroker for Java Reference*, provides methods for determining the type of the contained data as well as for setting and extracting the value of primitive data types.

Constructed data types are represented by the following interfaces, which are all derived from `DynAny`. Each of these interfaces provides its own set of methods that are appropriate for setting and extracting the values it contains.

Table 27.1 Interfaces derived from `DynAny` that represent constructed data types

Interface	TypeCode	Description
<code>DynArray</code>	<code>_tk_array</code>	An array of values with the same data type that has a fixed number of elements.
<code>DynEnum</code>	<code>_tk_enum</code>	A single enumeration value.
<code>DynFixed</code>	<code>_tk_fixed</code>	Not supported.
<code>DynSequence</code>	<code>_tk_sequence</code>	A sequence of values with the same data type. The number of elements may be increased or decreased.
<code>DynStruct</code>	<code>_tk_struct</code>	A structure.
<code>DynUnion</code>	<code>_tk_union</code>	A union.
<code>DynValue</code>	<code>_tk_value</code>	Not supported.

Usage restrictions

A `DynAny` object may only be used locally by the process which created it. Any attempt to use a `DynAny` object as a parameter on an operation request for a bound object or to externalize it using the `ORB.object_to_string` method will cause a `MARSHAL` exception to be raised.

Furthermore, any attempt to use a `DynAny` object as a parameter on DII request will cause a `NO_IMPLEMENT` exception to be raised.

This version does not support the long double and fixed types as specified in CORBA 2.3.

Creating a DynAny

A `DynAny` object is created by invoking an operation on a `DynAnyFactory` object. First obtain a reference to the `DynAnyFactory` object, and then use that object to create the new `DynAny` object.

```
// Resolve Dynamic Any Factory
DynAnyFactory factory =
    DynAnyFactoryHelper.narrow(orb.resolve_initial_references("DynAnyFactory"));
byte[] oid = "PrinterManager".getBytes();

// Create the printer manager object.
PrinterManagerImpl manager =
    new PrinterManagerImpl((com.inprise.vbroker.CORBA.ORB) orb, factory, serverPoa, oid);

// Export the newly create object.
serverPoa.activate_object_with_id(oid, manager);
System.out.println(manager + " is ready.");
```


Initializing and accessing the value in a DynAny

The `DynAny.insert_<type>` methods allow you to initialize a `DynAny` object with a variety of basic data types, where `<type>` is `boolean`, `octet`, `char`, and so on. Any attempt to insert a type that does not match the `TypeCode` defined for the `DynAny` will cause an `TypeMismatch` exception to be raised.

The `DynAny.get_<type>` methods allow you to access the value contained in a `DynAny` object, where `<type>` is `boolean`, `octet`, `char`, and so on. Any attempt to access a value from a `DynAny` component which does not match the `TypeCode` defined for the `DynAny` will cause a `TypeMismatch` exception to be raised.

The `DynAny` interface also provide methods for copying, assigning, and converting to or from an `Any` object. The sample programs, described later in this chapter, provide examples of how to use some of these methods. The *VisiBroker for Java Reference* provides a complete description of these methods.

Constructed data types

The following types are derived from the `DynAny` interface and are used to represent constructed data types. These interfaces, and the methods they offer, all described in the *VisiBroker for Java Reference*.

Traversing the components in a constructed data type

Several of the interfaces that are derived from `DynAny` actually contain multiple components. The `DynAny` interface provides methods that allow you to iterate through these components. The `DynAny`-derived objects that contain multiple components maintain a pointer to the current component.

DynAny method	Description
<code>rewind</code>	Resets the current component pointer to the first component. Has no effect if the object contains only one component.
<code>next</code>	Advances the pointer to the next component. If there are no more components or if the object contains only one component, <code>false</code> is returned.
<code>current_component</code>	Returns a <code>DynAny</code> object, which may be narrowed to the appropriate type, based on the component's <code>TypeCode</code> .
<code>seek</code>	Sets the current component pointer to the component with the specified, zero-based index. Returns <code>false</code> if there is no component at the specified index. Sets the current component pointer to <code>-1</code> (no component) if specified with a negative index.

DynEnum

This interface represents a single enumeration constant. Methods are provided for setting and obtaining the value as a string or as an integral value.

DynStruct

This interface represents a dynamically constructed `struct` type. The members of the structure can be retrieved or set using a sequence of `NameValuePair` objects. Each `NameValuePair` object contains the member's name and an `Any` containing the member's Type and value.

You may use the `rewind`, `next`, `current_component`, and `seek` methods to traverse the members in the structure. Methods are provided for setting and obtaining the structure's members.

DynUnion

This interface represents a `union` and contains two components. The first component represents the discriminator and the second represents the member value.

You may use the `rewind`, `next`, `current_component`, and `seek` methods to traverse the components. Methods are provided for setting and obtaining the union's discriminator and member value.

DynSequence and DynArray

A `DynSequence` or `DynArray` represents a sequence of basic or constructed data types without the need of generating a separate `DynAny` object for each component in the sequence or array. The number of components in a `DynSequence` may be changed, while the number of components in a `DynArray` is fixed.

You may use the `rewind`, `next`, `current_component`, and `seek` methods to traverse the members in a `DynArray` or `DynSequence`.

Example IDL

The following code sample shows the IDL used in the example client and server applications. The `StructType` structure contains two basic data types and an enumeration value. The `PrinterManager` interface is used to display the contents of an `Any` without any static information about the data type it contains.

Code sample 27.1 IDL for the `DynAny` example clients

```
// Printer.idl
module Printer {
    enum EnumType {first, second, third, fourth};
    struct StructType {
        string str;
        EnumType e;
        float fl;
    };
};
```

```

interface PrinterManager {
    void printAny(in any info);
    oneway void shutdown();
};
};

```

Example client application

Code sample 27.2 shows a client application that can be found in the `dynany` directory of the `examples` directory in the VisiBroker distribution. The path is `inprise/vbroker/examples/dynany`. The client application uses the `DynStruct` interface to dynamically create a `StructType` structure.

The `DynStruct` interface uses a sequence of `NameValuePair` objects to represent the structure members and their corresponding values. Each name-value pair consists of a string containing the structure member's name and an `Any` object containing the structure member's value.

After initializing the ORB in the usual manner and binding to an `PrintManager` object, the client performs these steps:

- 1 Create an empty `DynStruct` with the appropriate type.
- 2 Create a sequence of `NameValuePair` objects that will contain the structure members.
- 3 Create and initialize `Any` objects for each of the structure member's values.
- 4 Initialize each `NameValuePair` with the appropriate member name and value.
- 5 Initialize the `DynStruct` object with the `NameValuePair` sequence.
- 6 Invoke the `PrinterManager.printAny` method, passing the `DynStruct` converted to a regular `Any`.

Note You must use the `DynAny.to_any` method to convert a `DynAny` object, or one of its derived types, to an `Any` before passing it as a parameter on an operation request.

Code sample 27.2 Example client application that uses `DynStruct`

```

// Client.java

import org.omg.DynamicAny.*;

public class Client {

    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);

            DynAnyFactory factory =
                DynAnyFactoryHelper.narrow(orb.resolve_initial_references("DynAnyFactory"));

```

Example server application

```
// Locate a printer manager.
Printer.PrinterManager manager =
    Printer.PrinterManagerHelper.bind(orb, "PrinterManager");

// Create Dynamic struct
DynStruct info =
    DynStructHelper.narrow(factory.create_dyn_any_from_type_code
        (Printer.StructTypeHelper.type()));

// Create our NameValuePair sequence (array)
NameValuePair[] NVPair = new NameValuePair[3];

// Create and initialize Dynamic Struct data as any's
org.omg.CORBA.Any str_any = orb.create_any();
str_any.insert_string("String");
org.omg.CORBA.Any e_any = orb.create_any();
Printer.EnumTypeHelper.insert(e_any, Printer.EnumType.second);
org.omg.CORBA.Any fl_any = orb.create_any();
fl_any.insert_float((float)864.50);

NVPair[0] = new NameValuePair("str", str_any);
NVPair[1] = new NameValuePair("e", e_any);
NVPair[2] = new NameValuePair("fl", fl_any);

// Initialize the Dynamic Struct
info.set_members(NVPair);

manager.printAny(info.to_any());

manager.shutdown();
}
catch (Exception e) {
    e.printStackTrace();
}
}
```

Example server application

The following code sample shows a server application that can be found in the `dynany` directory of the `examples` directory in the VisiBroker distribution. The server application performs these steps.

- 1 Initialize the ORB.
- 2 Create the policies for the POA.
- 3 Create a `PrintManager` object.
- 4 Export the `PrintManager` object.
- 5 Print a message and wait for incoming operation requests.

Code sample 27.3 Example server application

```
// Server.java

import java.util.*;
import org.omg.DynamicAny.*;
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.*;

public class Server {

    public static void main(String[] args) {
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

            // Resolve Root POA
            POA rootPoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootPoa.the_POAManager().activate();

            // Create a BindSupport Policy that makes POA register each servant
            // with osagent
            org.omg.CORBA.Any any = orb.create_any();
            BindSupportPolicyValueHelper.insert(any, BindSupportPolicyValue.BY_INSTANCE);

            org.omg.CORBA.Policy bsPolicy =
                orb.create_policy(BIND_SUPPORT_POLICY_TYPE.value, any);

            // Create policies for our testPOA
            org.omg.CORBA.Policy[] policies = {
                rootPoa.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
                bsPolicy
            };

            // Create managerPOA with the right policies
            POA serverPoa = rootPoa.create_POA( "serverPoa", rootPoa.the_POAManager(), policies );

            // Resolve Dynamic Any Factory
            DynAnyFactory factory =
                DynAnyFactoryHelper.narrow(orb.resolve_initial_references("DynAnyFactory"));

            byte[] oid = "PrinterManager".getBytes();

            // Create the printer manager object.
            PrinterManagerImpl manager =
                new PrinterManagerImpl((com.inprise.vbroker.CORBA.ORB) orb, factory, serverPoa, oid);

            // Export the newly create object.

            serverPoa.activate_object_with_id(oid, manager);

            System.out.println(manager + " is ready.");
        }
    }
}
```

```
// Wait for incoming requests
orb.run();
}
catch (Exception e) {
    e.printStackTrace();
}
}
}
```

The following code sample shows how the `PrinterManager` implementation follows these steps in using a `DynAny` to process the `Any` object, without any compile-time knowledge of the type the `Any` contains.

- 1 Create a `DynAny` object, initializing it with the received `Any`.
- 2 Perform a switch on the `DynAny` object's type.
- 3 If the `DynAny` contains a basic data type, simply print out the value.
- 4 If the `DynAny` contains an `Any` type, create a `DynAny` for it, determine it's contents, and then print out the value.
- 5 If the `DynAny` contains an enum, create a `DynEnum` for it and then print out the string value.
- 6 If the `DynAny` contains a union, create a `DynUnion` for it and then print out the union's discriminator and the member.
- 7 If the `DynAny` contains a struct, array, or sequence, traverse through the contained components and print out each value.

Code sample 27.4 Implementation of `PrinterManager` showing the use of `DynAny` types to process a received `Any` object

```
// PrinterManagerImpl.java

import java.util.*;
import org.omg.DynamicAny.*;
import org.omg.PortableServer.*;

public class PrinterManagerImpl extends Printer.PrinterManagerPOA {
    private com.inprise.vbroker.CORBA.ORB _orb;
    private DynAnyFactory _factory;
    private POA _poa;
    private byte[] _oid;

    public PrinterManagerImpl(com.inprise.vbroker.CORBA.ORB orb,
        DynAnyFactory factory, POA poa, byte[] oid) {
        _orb = orb;
        _factory = factory;
        _poa = poa;
        _oid = oid;
    }

    public synchronized void printAny(org.omg.CORBA.Any info) {
        // Display info with the assumption that we don't have
        // any info statically about the type inside the any
    }
}
```

```

try {
    // Create a DynAny object
    DynAny dynAny = _factory.create_dyn_any(info);
    display(dynAny);
}
catch (Exception e) {
    e.printStackTrace();
}

}

public void shutdown() {
    try {
        _poa.deactivate_object(_oid);
        System.out.println("Server shutting down");
        _orb.shutdown(false);
    }
    catch (Exception e) {
        System.out.println(e);
    }
}

private void display(DynAny value) throws Exception {
    switch(value.type().kind().value()) {
        case org.omg.CORBA.TCKind._tk_null:
        case org.omg.CORBA.TCKind._tk_void: {
            break;
        }
        case org.omg.CORBA.TCKind._tk_short: {
            System.out.println(value.get_short());
            break;
        }
        case org.omg.CORBA.TCKind._tk_ushort: {
            System.out.println(value.get_ushort());
            break;
        }
        case org.omg.CORBA.TCKind._tk_long: {
            System.out.println(value.get_long());
            break;
        }
        case org.omg.CORBA.TCKind._tk_ulong: {
            System.out.println(value.get_ulong());
            break;
        }
        case org.omg.CORBA.TCKind._tk_float: {
            System.out.println(value.get_float());
            break;
        }
        case org.omg.CORBA.TCKind._tk_double: {
            System.out.println(value.get_double());
            break;
        }
        case org.omg.CORBA.TCKind._tk_boolean: {
            System.out.println(value.get_boolean());

```

```
        break;
    }
    case org.omg.CORBA.TCKind._tk_char: {
        System.out.println(value.get_char());
        break;
    }
    case org.omg.CORBA.TCKind._tk_octet: {
        System.out.println(value.get_octet());
        break;
    }
    case org.omg.CORBA.TCKind._tk_string: {
        System.out.println(value.get_string());
        break;
    }
    case org.omg.CORBA.TCKind._tk_any: {
        DynAny dynAny = _factory.create_dyn_any(value.get_any());
        display(dynAny);
        break;
    }
    case org.omg.CORBA.TCKind._tk_TypeCode: {
        System.out.println(value.get_typecode());
        break;
    }
    case org.omg.CORBA.TCKind._tk_objref: {
        System.out.println(value.get_reference());
        break;
    }
    case org.omg.CORBA.TCKind._tk_enum: {
        DynEnum dynEnum = DynEnumHelper.narrow(value);
        System.out.println(dynEnum.get_as_string());
        break;
    }
    case org.omg.CORBA.TCKind._tk_union: {
        DynUnion dynUnion = DynUnionHelper.narrow(value);
        display(dynUnion.get_discriminator());
        display(dynUnion.member());
        break;
    }
    case org.omg.CORBA.TCKind._tk_struct:
    case org.omg.CORBA.TCKind._tk_array:
    case org.omg.CORBA.TCKind._tk_sequence: {
        value.rewind();
        boolean next = true;
        while(next) {
            DynAny d = value.current_component();
            display(d);
            next = value.next();
        }
        break;
    }
}
```



```

        case org.omg.CORBA.TCKind._tk_longlong: {
            System.out.println(value.get_longlong());
            break;
        }
        case org.omg.CORBA.TCKind._tk_ulonglong: {
            System.out.println(value.get_ulonglong());
            break;
        }
        case org.omg.CORBA.TCKind._tk_wstring: {
            System.out.println(value.get_wstring());
            break;
        }
        case org.omg.CORBA.TCKind._tk_wchar: {
            System.out.println(value.get_wchar());
            break;
        }
        default:
            System.out.println("Invalid type");
    }
}
}

```


Using valuetypes

This chapter explains how to use the `valuetype` IDL type in VisiBroker.

Understanding valuetypes

The IDL type `valuetype` is used to pass state data over the wire. A `valuetype` is best thought of as a `struct` with inheritance and methods. Valuetypes differ from normal interfaces in that they contain properties to describe the valuetype's state, and contain implementation details beyond that of an interface. The following IDL code declares a simple valuetype:

IDL sample 28.1 Simple valuetype IDL

```
module Map {  
    valuetype Point {  
        public long x;  
        public long y;  
        private string label;  
        factory create (in long x, in long y, in string z);  
        void print();  
    };  
};
```

Valuetypes are always local. They are not registered with the ORB, and require no identity, as their value is their identity. They can not be called remotely.

Concrete valuetypes

Concrete valuetypes contain state data. They extend the expressive power of IDL structs by allowing:

- Single concrete valuetype derivation and multiple abstract valuetype derivation
- Multiple interface support (one concrete and multiple abstract)
- Arbitrary recursive valuetype definitions
- Null value semantics
- Sharing semantics

Valuetype derivation

You can derive a concrete valuetype from one other concrete valuetype. However, valuetypes can be derived from multiple other abstract valuetypes.

Sharing semantics

Valuetype instances can be shared by other valuetypes across or within other instances. Other IDL data types such as structs, unions, or sequences can not be shared. Valuetypes that are shared are isomorphic between the sending context and the receiving context.

In addition, when the same valuetype is passed into an operation for two or more arguments, the receiving context receives the same valuetype reference for both arguments.

Null semantics

Null valuetypes can be passed over the wire, unlike IDL data types such as structs, unions, and sequences. For instance by boxing a struct as a boxed valuetype, you can pass a null value struct. For more information, see “Boxed valuetypes” on page 28-7.

Factories

Factories are methods that can be declared in valuetypes to create valuetypes in a portable way. For more information on Factories, see “Implementing factories” on page 28-5.

Abstract valuetypes

Abstract valuetypes contain only methods and do not have state. They may not be instantiated. Abstract valuetypes are a bundle of operation signatures with a purely local implementation.

For instance, the following IDL defines an abstract valuetype `Account` that contains no state, but one method, `get_name`:

```
abstract valuetype Account{  
    string get_name();  
}
```

Now, two valuetypes are defined that inherit the `get_name` method from the abstract valuetype:

```
valuetype savingsAccount:Account{
    private long balance;
}
valuetype checkingAccount:Account{
    private long balance;
}
```

These two valuetypes contain a variable `balance`, and they inherit the `get_name` method from the abstract valuetype `Account`.

Implementing valuetypes

To implement valuetypes in an application, do the following:

- 1 Define the valuetypes in an IDL file.
- 2 Compile the IDL file using `idl2java`.
- 3 Implement your valuetypes by inheriting the valuetype base class.
- 4 Implement the Factory class to implement any factory methods defined in IDL
- 5 Implement the `create_for_unmarshal` method.
- 6 If necessary, register your Factory with the ORB.
- 7 Either implement the `_add_ref`, `_remove_ref`, and `_ref_countvalue` methods or derive from `CORBA::DefaultValueRefCountBase`.

Defining your valuetypes

In IDL sample 28.1 on page 28-1, you define a valuetype named `Point` that defines a point on a graph. It contains two public variables, the `x` and `y` coordinates, one private variable that is the label of the point, the valuetype's factory, and a print method to print the point.

Compiling your IDL file

Now that you've defined your IDL, compile it using `idl2java`. This will create the Java source files that you will modify to implement your valuetypes.

If you compile the above IDL, your output will consist of the following files:

- `Point.java`
- `PointDefaultFactory.java`
- `PointHelper.java`
- `PointHolder.java`
- `PointValueFactory.java`

Inheriting the valuetype base class

After compiling your IDL, create your implementation of the valuetype. The implementation class will inherit the base class. This class contains the constructor that is called in your `ValueFactory`, and contains all the variables and methods declared in your IDL.

For example, in `obv\PointImpl.java`, the `PointImpl` class extends the `Point` class which was generated from the IDL:

```
public class PointImpl extends Point {
    public PointImpl() {}
    public PointImpl(int a_x, int a_y, String a_label) {
        x = a_x;
        y = a_y;
        label = a_label;
    }
    public void print () {
        System.out.println("Point is [" + label + ": (" + x + ", " + y + ")");
    }
}
```

Implementing the Factory class

Now that you have created an implementation class, implement the Factory for your valuetype.

In our example, the generated `Point_init` class contains the `create` method declared in your IDL. This class extends `org.omg.CORBA.portable.ValueFactory`. The `PointDefaultFactory` class implements `PointValueFactory`:

```
public class PointDefaultFactory implements PointValueFactory {
    public java.io.Serializable read_value (org.omg.CORBA.portable.InputStream is) {
        java.io.Serializable val = new PointImpl(); // Called the implementation class
        // create and initialize value
        val = ((org.omg.CORBA_2_3.portable.InputStream)is).read_value(val);
        return val;
    }
    // It is up to the user to implement the valuetype however they want:
    public Point create (int x,
        int y,
        java.lang.String z) {
        // IMPLEMENT:
        return null;
    }
}
```

`PointImpl()` is called to create a new valuetype, which is read in from the `InputStream` by `read_value`.

Note You must call `read_value` or your Factory will not work, and you may not call any other method.

Registering your Factory with the ORB

Call `ORB.register_value_factory` to register your Factory with the ORB. This is required only if you do not name your factory `valuetypeNameDefaultFactory`. See “Registering valuetypes” on page 28-6 for more information on registering Factories.

Implementing factories

When the ORB receives a valuetype, it must first be demarshaled, and then the appropriate factory for that type must be found in order to create a new instance of that type. Once the instance has been created, the value data is unmarshalled into the instance. The type is identified by the RepositoryID that is passed as part of the invocation. The mapping between the type and the factory is language specific.

The following code, using JDK 1.2, contains a sample implementation of the factory of the Point valuetype:

Code sample 28.1 Factory for Point valuetype

```
public class PointDefaultFactory implements PointValueFactory {
    public java.io.Serializable read_value (org.omg.CORBA.portable.InputStream is) {
        java.io.Serializable val = new PointImpl();
        // create and initialize value
        // It is very important that this call is made.
        val = ((org.omg.CORBA_2_3.portable.InputStream)is).read_value(val);
        return val;
    }
    public Point create (int x, int y, java.lang.String z) {
        // IMPLEMENT:
        return NO_IMPLEMENT;
    }
}
```

VisiBroker 4.5 will generate the correct signatures for either the JDK 1.2 or JDK 1.3 default value factory method. Existing (4.0) generated code is not designed to run under JDK 1.3, unless you modify the default value factory method signature as shown in the below. If you use your existing code with JDK 1.3 and do not modify default value factory, the code will not compile or will throw a `NO_IMPLEMENT` exception. Consequently, we recommend that you regenerate your code to generate the correct signatures.

The following code sample shows how you should modify the default value factory method signature to make sure that it compiles under JDK 1.3:

Code sample 28.2 Factory code showing the method signature for JDK 1.3 code generation

```
public class PointDefaultFactory implements PointValueFactory {
    public java.io.Serializable read_value (org.omg.CORBA_2_3.portable.InputStream is) {
        java.io.Serializable val = new PointImpl();
        // create and initialize value
        // It is very important that this call is made.
        val = ((org.omg.CORBA_2_3.portable.InputStream)is).read_value(val);
        return val;
    }
    public Point create (int x, int y, java.lang.String z) {
        // IMPLEMENT:
        return NO_IMPLEMENT;
    }
}
```

Factories and valuetypes

When the ORB receives a valuetype, it will look for that type's factory. It will look for a factory named *valuetypeDefaultFactory*. For instance, the Point valuetype's factory is called *PointDefaultFactory*. If the correct factory doesn't conform to this naming schema (*valuetypeDefaultFactory*), you must register the correct factory so the ORB can create an instance of the valuetype.

If the ORB cannot find the correct factory for a given valuetype, a MARSHAL exception is raised, with an identified minor code.

Registering valuetypes

Each language mapping specifies how and when registration occurs. If you created a factory with the *valuetypeDefaultFactory* naming convention, this is considered implicitly registering that factory, and you do not need to explicitly register your factory with the ORB.

To register a factory that doesn't conform to the *valuetypeDefaultFactory* naming convention, call *register_value_factory*. To unregister a factory, call *unregister_value_factory* on the ORB. You can also lookup a registered valuetype factory by calling *lookup_value_factory* on the ORB.

Boxed valuetypes

Boxed valuetypes allow you to wrap non-value IDL data types as valuetypes. For example, the following IDL boxed valuetype declaration,

```
valuetype Label string;
```

is equivalent to this IDL valuetype declaration:

```
valuetype Label{
    public string name;
}
```

By boxing other data types as valuetypes, it allows you to use valuetype's null semantics and sharing semantics.

Valueboxes are implemented purely with generated code. No user code is required.

Abstract interfaces

Abstract interfaces allow you to choose at runtime whether the object will be passed by value or by reference.

They differ from IDL interfaces in the following ways:

- The actual parameter type determines whether the object is passed by reference or a valuetype is passed. The parameter type is determined based on two rules. It is treated as an object reference if it is a regular interface type or sub-type, the interface type is a sub-type of the signature abstract interface type, and the object is already registered with the ORB. It is treated as a value if it can not be passed as an object reference, but can be passed as a value. If it fails to pass as a value, a `BAD_PARAM` exception is raised.
- Abstract interfaces do not implicitly derive from `org.omg.CORBA.Object` because they can represent either object references or valuetypes. Valuetypes do not necessarily support common object reference operations. If the abstract interface can be successfully narrowed to an object reference type, you can invoke the operations of `org.omg.CORBA.Object`.
- Abstract interfaces may only inherit from other abstract interfaces.
- valuetypes can support one or more abstract interfaces.

For example, examine the following abstract interface.

IDL sample 28.2 Abstract interface IDL

```
abstract interface ai{
};
interface itp : ai{
};
valuetype vtp supports ai{
};

interface x {
    void m(ai aitp);
};
valuetype y {
    void op(ai aitp);
};
```

For the argument to method `m`:

- `itp` is always passed as an object reference.
- `vtp` is passed as a value.

Custom valuetypes

By declaring a custom valuetype in IDL, you bypass the default marshalling and unmarshalling model and are responsible for encoding and decoding.

IDL sample 28.3 Custom valuetype IDL

```
custom valuetype customPoint{
    public long x;
    public long y;
    private string label;
    factory create(in long x, in long y, in string z);
};
```

You must implement the `marshal` and `unmarshal` methods from the `CustomMarshal` interface.

When you declare a custom valuetype, the valuetype extends `org.omg.CORBA.portable.CustomValue`, as opposed to `org.omg.CORBA.portable.StreamableValue`, as in a regular valuetype. The compiler doesn't generate read or write methods for your valuetype.

You must implement your own read and write methods by using `org.omg.CORBA.portable.DataInputStream` and `org.omg.CORBA.portable.DataOutputStream` to read and write the values, respectively. For more information on these classes, see the *VisiBroker for Java Reference*.

Truncatable valuetypes

Truncatable valuetypes allow you to treat an inherited valuetype as its parent.

The following IDL defines a valuetype `checkingAccount` that is inherited from the base type `Account` and can be truncated an the receiving object.

```
valuetype checkingAccount: truncatable Account{  
    private long balance;  
}
```

This is useful if the receiving context doesn't need the new data members or methods in the derived valuetype, and if the receiving context isn't aware of the derived valuetype. However, any state data from the derived valuetype that isn't in the parent data type will be lost when the valuetype is passed to the receiving context.

Note You cannot make a custom valuetype truncatable.

Using URL naming

This chapter explains how to use the URL Naming Service which allows you to associate a URL (Uniform Resource Locator) with an object's IOR (Interoperable Object Reference). Once a URL has been bound to an object, client applications can obtain a reference to the object by specifying the URL as a string instead of the object's name. If you want client applications to locate objects without using the osagent or a CORBA Naming Service, specifying a URL is an alternative.

URL Naming Service

The URL Naming Service is a simple mechanism that lets a server object associate its IOR with a URL in the form of a string in a file. Client programs can then locate the object using the URL pointing to the file containing the stringified URL on the web server. The URL Naming Service supports the `http` URL scheme for registering objects and any URL scheme that your Java runtime supports, such as `http:`, `ftp:`, or `file:` for locating an object by the URL.

This URL name service provides a way to locate objects without using the Smart Agent or a CORBA Naming Service. It enables client applications to locate objects provided by any vendor. The IDL specification for this service is shown in IDL sample 29.1.

Note VisiBroker's URL Naming supports whatever form of URL handling that your Java environment supports.

IDL sample 29.1 WebNaming module

```
// WebNaming.idl
#pragma prefix "inprise.com"
module URLNaming {
    exception InvalidURL(string reason);;
    exception CommFailure(string reason);;
    exception ReqFailure(string reason);;
    exception AlreadyExists(string reason);;
    abstract interface Resolver {
        // Read Operations
        Object locate(in string url_s)
            raises (InvalidURL, CommFailure, ReqFailure);
        // Write Operations
        void force_register_url(in string url_s, in Object obj)
            raises (InvalidURL, CommFailure, ReqFailure);
        void register_url(in string url_s, in Object obj)
            raises (InvalidURL, CommFailure, ReqFailure, AlreadyExists);
    };
};
```

Registering objects

Object servers register objects by binding to the `Resolver` and then using the `register_url` or the `force_register_url` method to associate a URL with an object's IOR. `register_url` is used to associate a URL with an object's IOR if no prior association exists. Using the `force_register_url` method associates a URL with an object's IOR regardless of whether a URL has already been bound to that object. If you use the `register_url` method under the same circumstances, an `AlreadyExists` exception is raised. For information about all of the available methods, see the *VisiBroker for Java Reference*.

For an example illustrating the server-side use of this feature, see Code sample 29.1. This example uses `force_register_url`. For `force_register_url` to be successful, the web server must be allowed to issue HTTP PUT commands. The code for the examples in this chapter is provided in the `bank_URL` directory within the `java_examples` directory where the VisiBroker for Java product was installed.

Note To get a reference to the `Resolver`, use the ORB's `resolve_initial_references` method, as shown in the example.

Code sample 29.1 Associating a URL with an Object's IOR

```

...
public class Server {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: vbj Server <URL string>");
            return;
        }
        String url = args[0];
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // get a reference to the root POA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // Create the servant
            AccountManagerImpl managerServant = new AccountManagerImpl();
            // Decide on the ID for the servant
            byte[] managerId = "BankManager".getBytes();
            // Activate the servant with the ID on myPOA
            rootPOA.activate_object_with_id(managerId, managerServant);

            // Activate the POA manager
            rootPOA.the_POAManager().activate();
            // Create the object reference
            org.omg.CORBA.Object manager =
                rootPOA.servant_to_reference(managerServant);
            // Obtain the URLNaming Resolver
            Resolver resolver = ResolverHelper.narrow(
                orb.resolve_initial_references("URLNamingResolver"));
            // Register the object reference (overwrite if exists)
            resolver.force_register_url(url, manager);
            System.out.println(manager + " is ready.");
            // Wait for incoming requests
            orb.run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

In this code sample `args[0]` is of the form

Example `http://<host_name>:<http_server_port>/<ior_file_path>/<ior_file_name>`

The `ior_file_name` is the user-specified file name where the stringified object reference is stored. The suffix of the `ior_file_name` must be `.ior` if the Gatekeeper will be used instead of an HTTP server. An example using the Gatekeeper and its default port number is

Example `http://mars:15000/URLNaming/Bank_Manager.ior`

Locating an object by URL

Client applications do not need to bind to the `Resolver`, they simply specify the URL when they call the `bind` method, as shown in Code sample 29.2. The `bind` accepts the URL as the object name. If the URL is invalid, an `InvalidURL` exception is raised. The `bind` method transparently calls `locate()` for you. For an example of how to use `locate()`, see Code sample 29.3 on page 29-5.

Code sample 29.2 Obtaining an object reference, given a URL

```
// ResolverClient.java
import com.inprise.vbroker.URLNaming.*;
public class ResolverClient {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: vbj Client <URL string> [Account name]");
            return;
        }
        String url = args[0];
        try {
            // Initialize the ORB.
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            // Obtain the URLNaming Resolver
            Resolver resolver = ResolverHelper.narrow(
                orb.resolve_initial_references("URLNamingResolver"));
            // Locate the object
            Bank.AccountManager manager =
                Bank.AccountManagerHelper.narrow(resolver.locate(url));
            // use args[0] as the account name, or a default.
            String name = args.length > 1 ? args[1] : "Jack B. Quick";
            // Request the account manager to open a named account.
            Bank.Account account = manager.open(name);
            // Get the balance of the account.
            float balance = account.balance();
            // Print out the balance.
            System.out.println("The balance in " + name + "'s account is $" + balance);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


Code sample 29.3 Obtaining an object reference using the Resolver.locate method

```
// Client.java
public class Client {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: vbj Client <URL string> [Account name]");
            return;
        }
        String url = args[0];
        // Initialize the ORB.
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // Locate the object
        Bank.AccountManager manager = Bank.AccountManagerHelper.bind(orb, url);
        // use args[0] as the account name, or a default.
        String name = args.length > 1 ? args[1] : "Jack B. Quick";
        // Request the account manager to open a named account.
        Bank.Account account = manager.open(name);
        // Get the balance of the account.
        float balance = account.balance();
        // Print out the balance.
        System.out.println("The balance in " + name + "'s account is $" + balance);
    }
}
```


Bidirectional Communication

This chapter explains how to establish bidirectional connections in VisiBroker without using the Gatekeeper. Information about bidirectional communications when using Gatekeeper is available in the VisiBroker for Java *Gatekeeper Guide*.

Note: Before enabling bidirectional IIOP, please read about “Security considerations” on page 30-11.

Using bidirectional IIOP

Most clients and servers that exchange information via the Internet are typically protected by corporate firewalls. In systems where requests are initiated only by the clients, the presence of firewalls is usually transparent to the clients. However, there are cases where clients need information asynchronously, that is, information must arrive that is not in response to a request. Client-side firewalls prevent servers from initiating connections back to clients. Therefore, if a client is to receive asynchronous information, it usually requires additional configuration.

In earlier versions of GIOP and VisiBroker, the only way to make it possible for a server to send asynchronous information to a client was to use a client-side Gatekeeper to handle the callbacks from the server.

If you use bidirectional IIOP, rather than having servers open separate connections to clients when asynchronous information needs to be transmitted back to clients (these would be rejected by client-side firewalls anyway), servers use the client-initiated connections to transmit information to clients. The CORBA specification also adds a new policy to portably control this feature.

Because bidirectional IIOP allows callbacks to be set up without a Gatekeeper, it greatly facilitates deployment of clients.

Bidirectional ORB properties

Three properties provide bidirectional support:

```
vbroker.orb.enableBiDir=client|server|both|none
vbroker.se.<sename>.scm.<scmname>.manager.exportBiDir=true|false
vbroker.se.<sename>.scm.<scmname>.manager.importBiDir=true|false
```

vbroker.orb.enableBiDir property

The `vbroker.orb.enableBiDir` property can be used on both the server and the client to enable bidirectional communication. This property allows you to change an existing unidirectional application into a bidirectional one without changing any code. The `vbroker.orb.enableBiDir` property may be set to the following values:

Value	Description
client	Enables bidirectional IIOP for all POAs and for all outgoing connections. This setting is equivalent to creating all POAs with a setting of the <code>BiDirectional</code> policy to <code>both</code> and setting the policy override for the <code>BiDirectional</code> policy to <code>both</code> on the ORB level. Furthermore, all created SCMs will permit bidirectional connections, as if the <code>exportBiDir</code> property had been set to true for every SCM.
server	Causes the server to accept and use connections that are bidirectional. This is equivalent to setting the <code>importBiDir</code> property on all SCMs to <code>true</code> .
both	Sets the property to both <code>client</code> and <code>server</code> .
none	Disables bidirectional GIOP altogether. This is the default value.

vbroker.se.<sename>.scm.<scmname>.manager.exportBiDir property

The `vbroker.se.<sename>.scm.<scmname>.manager.exportBiDir` property is a client-side property. By default, it is not set to anything by the ORB. Setting it to `true` enables creation of a bidirectional callback POA on the specified server engine. Setting it to `false` disables creation of a bidirectional POA on the specified server engine.

vbroker.se.<sename>.scm.<scmname>.manager.importBiDir property

The `vbroker.se.<se-name>.scm.<scm-name>.manager.importBiDir` property is a server-side property. By default, it is not set to anything by the ORB. Setting it to `true` allows the server-side to reuse the connection already established by the client for sending requests to the client. Setting it to `false` prevents reuse of connections in this fashion.

NOTE: These properties are evaluated only once -- when the SCMs are created. In all cases, the `exportBiDir` and `importBiDir` properties on the SCMs govern the `enableBiDir` property. In other words, if both properties are set to conflicting values, the SCM-specific properties will take effect. This allows you to set the `enableBiDir` property globally and specifically turn off `BiDir` in individual SCMs.

About the examples

Examples demonstrating use of this feature are located in subdirectories of `examples/bidir-iop` in the VisiBroker for Java installation directory.

All the examples are based on a simple stock quote callback application:

- 1 The client creates a CORBA object that processes stock quote updates;
- 2 The client sends the object reference of this CORBA object to the server;
- 3 The server invokes this callback object to periodically update stock quotes.

In the sections that follow, these examples are used to explain different aspects of the bidirectional IIOP feature.

Enabling bidirectional IIOP for existing applications

You can enable bidirectional communication in existing VisiBroker for Java applications without modifying any source code. A simple callback application that does not use Bidirectional IIOP at all is stored in the `examples/bidir-iop/basic/` directory.

To enable bidirectional IIOP for this application, you set the `vbroker.orb.enableBiDir` property:

- 1 Make sure the osagent is running.
- 2 Start the server.

Unix: `prompt> vbj -Dvbroker.orb.enableBiDir=server Server &`

Windows: `prompt>start vbj -Dvbroker.orb.enableBiDir=server Server`

- 3 Start the client:

`prompt> vbj -Dvbroker.orb.enableBiDir=client RegularClient`

The existing callback application now uses bidirectional IIOP and works through a client-side firewall.

Explicitly enabling bidirectional IIOP

The Client in directory `examples/bidir-iop/basic` is derived from the `RegularClient` described above, except that this client enables bidirectional IIOP programmatically.

The changes required are in the client code only. To convert the unidirectional client into a bidirectional client, all you need to do is:

- 1 Include the `BiDirectional` policy in the list of policies for the callback POA and
- 2 Add the `BiDirectional` policy to the list of overrides for the object reference that refers to the server for which we want to enable bidirectional IIOP.

3 Set the `exportBiDir` property to `true` in the client.

In the following code snippet, the code that implements bidirectional IIOP is displayed in bold:

```

public static void main (String[] args) {
    try {
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
        org.omg.PortableServer.POA rootPOA = org.omg.PortableServer.POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
        org.omg.CORBA.Any bidirPolicy = orb.create_any();
        bidirPolicy.insert_short(BOTH.value);
        org.omg.CORBA.Policy[] policies = {
            //set bidir policy
            orb.create_policy(BIDIRECTIONAL_POLICY_TYPE.value, bidirPolicy)
        };
        org.omg.PortableServer.POA callbackPOA =
            rootPOA.create_POA("bidir", rootPOA.the_POAManager(), policies);
        QuoteConsumerImpl c = new QuoteConsumerImpl();
        callbackPOA.activate_object(c);
        callbackPOA.the_POAManager().activate();
        QuoteServer serv = QuoteServerHelper.bind(orb, "/QuoteServer_poa",
            "QuoteServer".getBytes());
        serv=QuoteServerHelper.narrow(serv._set_policy_override(
            policies, org.omg.CORBA.SetOverrideType.ADD_OVERRIDE));

        serv.registerConsumer(QuoteConsumerHelper.narrow(callbackPOA.servant_to_reference(c)));
        System.out.println("Client: consumer registered");
        //sleeping for 60 seconds, receiving message
        try{
            Thread.currentThread().sleep(60*1000);
        }
        catch(java.lang.InterruptedException e){ }

        serv.unregisterConsumer(QuoteConsumerHelper.narrow(callbackPOA.servant_to_reference(c)));
        System.out.println("Client: consumer unregistered. Good bye.");
        orb.shutdown(true);

        ...
    }
}

```

Note: Please see the chapter on QOS framework in the Visibroker for Java Programmers Guide for information about how to set policies to tune your application.

A client connection can be either unidirectional or bidirectional. A server can use a bidirectional connection to call back the client without opening a new connection.

Otherwise, the connection is considered unidirectional.

The POA on which the callback object is hosted must enable bidirectional IIOP by setting the `BiDirectional` policy to `BOTH`. This POA must be created on an SCM which has been enabled for bidirectional support by setting the `vbroker.<sename>.scm.<scmname>.manager.exportBiDir` property on the SCM manager. Otherwise, the POA will not be able to receive requests from the server over a client-initiated connection.

If a POA does not specify the `BiDirectional` Policy, it must not be exposed in outgoing connections. To satisfy this requirement, a POA which does not have the

BiDirectional policy set cannot be created on a Server Engine which has even one SCM whose `exportBiDir` property is set. If an attempt is made to create a POA on a unidirectional SE, an `InvalidPolicy` exception is raised, with the `ServerEnginePolicy` in error.

NOTE: Different objects using the same client connection may set conflicting overrides for the BiDirectional policy. Nevertheless, once a connection is made bidirectional, it always remains bidirectional, regardless of the policy effective at a later time.

Once we have full control over the bidirectional configuration, we enable bidirectional IIOP on the `iiop_tp` SCM only:

```
prompt> vbj -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.exportBiDir=true Client
```

Security considerations

Use of bidirectional IIOP may raise significant security issues. In the absence of other security mechanisms, a malicious client may claim that its connection is bidirectional for use with any host and port it chooses. In particular, a client may specify the host and port of security-sensitive objects not even resident on its host. In the absence of other security mechanisms, a server that has accepted an incoming connection has no way to discover the identity or verify the integrity of the client that initiated the connection. Further, the server might gain access to other objects accessible through the bidirectional connection. This is why use of a separate, bidirectional SCM for callback objects is encouraged. If there are any doubts as to the integrity of the client, it is recommended that bidirectional IIOP not be used.

For security reasons, a server running VisiBroker for Java will not use bidirectional IIOP unless explicitly configured to do so. The property `vbroker.<se>.<sename>.scm.<scmname>.manager.importBiDir` gives you control of bidirectionality on a per-SCM basis. For example, you might choose to enable bidirectional IIOP only on a server engine that uses SSL to authenticate the client, and to not make other, regular IIOP connections available for bidirectional use. (See “Bidirectional ORB properties” on page 30-8 for more information about how to do this.) In addition, on the client-side, you might want to enable bidirectional connections only to those servers that do callbacks outside of the client firewall. To establish a high degree of security between the client and server, you should use SSL with mutual authentication (set `vbroker.security.peerAuthenticationMode` to `REQUIRE_AND_TRUST` on both the client and server).

Backward compatibility

This part of the VisiBroker for Java *Programmer's Guide* includes these chapters.

- Chapter 31 “Using the BOA with VisiBroker 4.x”
- Chapter 32 “Migrating VisiBroker code”
- Chapter 33 “Using object activators”

Using the BOA with VisiBroker 4.x

This chapter describes how to use the BOA with VisiBroker 4.x.

Compiling your BOA code with VisiBroker 4.x

If you have existing BOA code that you developed with a previous version of VisiBroker, you can continue to use them with the current version as long as you keep the following points in mind.

- To generate the necessary BOA base code, you must use the “-boa” option with the `idl2java` tool. For more information on using `idl2java` to generate the code, see Chapter 2, “Programmer tools,” in the *VisiBroker for Java Reference*.
- Because the `BOA_init()` is no longer available under `org.omg.CORBA.ORB`, you must cast the ORB to `com.inprise.vbroker.CORBA.ORB`.
- Because the `BOA` class is no longer available in the `org.omg.CORBA` package, you must now refer to it in the `com.inprise.vbroker.CORBA` package. For more information on the ORB package, see Chapter 5, “Core interfaces and classes,” of the *VisiBroker for Java Reference*.

Supporting BOA options

All OA command line options supported by VisiBroker 3.x are still supported.

Limitations in using the BOA

Two features are not supported with VisiBroker 4.x BOA:

- Persistent DSI objects are not supported
- `_boa()` on DSI objects is not supported

Using object activators

BOA object activators are supported with VisiBroker 4.x. However, these activators can be used only with BOA, not POA. The POA uses servant activators and servant locators in place of object activators.

In this release of VisiBroker, the Portable Object Adaptor (POA) supports the features that were provided by the BOA in VisiBroker 3.x releases. For backward compatibility reasons, you may still use the object activators with your code. For more information on how to use the object activators with this release, see Chapter 33, “Using object activators.”

Naming objects under the BOA

Though the BOA is deprecated in VisiBroker 4.x, you may still use it in conjunction with the Smart Agent to specify a name for your server objects which may be bound to in your client programs.

Object names

When creating an object, a server must specify an object name if the object is to be made available to client applications through the osagent. When the server calls the `BOA.obj_is_ready` method, the object's interface name will only be registered with the VisiBroker osagent if the object is named. Objects that are given an object name when they are created return *persistent* object references, while objects which are not given object names are created as *transient*.

Note If you pass an empty string for the object name to the object constructor in VisiBroker for Java, a transient object is created (an object which is not registered with the Smart Agent). If you pass a null reference to the constructor, a transient object is created.

The use of an object name by your client application is required if it plans to bind to more than one instance of an object at a time. The object name distinguishes between multiple instances of an interface. If an object name is not specified when the `bind` method is called, the osagent will return any suitable object with the specified interface.

Note In VisiBroker 3.x, it was possible to have a server process that provided different interfaces, all of which had the same object name, but in VisiBroker 4.x different interfaces may not have the string equivalent names.

Migrating VisiBroker code

This chapter describes how to migrate your VisiBroker code from previous versions of VisiBroker to VisiBroker 4.5. There are two ways of migrating Java code from VisiBroker 3.x to VisiBroker 4.5: the migrator, a command-line utility that attempts to automate a significant part of the migration process, and manual migration. It is recommended that, whenever possible, VisiBroker 3.x code be manually migrated to VisiBroker 4.x. There are many advantages to using native VisiBroker 4.x calls rather than upgrading VisiBroker 3.x.

However, a migrator is provided to help migrate the code written for VisiBroker 3.x to VisiBroker 4.x. The migrator attempts to make its changes automatically, without any interaction with the user, although this is not always possible.

This chapter begins with information about how to use the migrator. It also provides information regarding:

- How to use the BOA with VisiBroker 4.5, how to change your BOA code to POA, and how to use servant activators.
- List of changes to package names, class names, and API calls in VisiBroker 4.5.

Migrator

When the migrator parses the original Java source files, it changes:

- Package name prefixes
- Class names
- API calls

Please see the following tables for more information on these migration changes.

Changes to package name prefixes

The migrator changes package names as follows:

Table 32.1 Changes to package name prefixes

VisiBroker 3.x package name prefix	VisiBroker 4.5 package name prefix
com.visigenic.vbroker	com.inprise.vbroker
com.visigenic.vbroker.services.CosEvent	com.inprise.vbroker.CosEvent
com.visigenic.vbroker.services.CosNaming	com.inprise.vbroker.naming

Changes to class names

The migrator changes some class names to those used in VisiBroker 4.5. If it cannot determine a compatible class, it substitutes a class from VisiBroker 3.x.

Table 32.2 Changes to class names

VisiBroker 3.x class name	VisiBroker 4.5 class name
org.omg.CORBA.BOA	com.inprise.vbroker.CORBA.BOA
org.omg.CORBA.DynamicImplementation	com.inprise.vbroker.CORBA.migration. DynamicImplementation
com.visigenic.vbroker.interceptor.BindInterceptor	com.inprise.vbroker.interceptor.migration. BindInterceptorDelegate
com.visigenic.vbroker.interceptor.ChainBindInterceptor	com.inprise.vbroker.interceptor.migration. ChainBindDelegateFactory
com.visigenic.vbroker.interceptor.ChainBindInterceptorHelper	com.inprise.vbroker.interceptor.migration. ChainBindDelegateFactoryHelper
com.visigenic.vbroker.interceptor.ClientInterceptor	com.inprise.vbroker.interceptor.migration. ClientInterceptorDelegate
com.visigenic.vbroker.interceptor.ClientInterceptorFactory	com.inprise.vbroker.interceptor.migration. ClientInterceptorFactory
com.visigenic.vbroker.interceptor.ChainClientInterceptorFactory	com.inprise.vbroker.interceptor.migration. ChainClientDelegateFactory
com.visigenic.vbroker.interceptor.ChainClientInterceptorFactoryHelper	com.inprise.vbroker.interceptor.migration. ChainClientDelegateFactoryHelper
com.visigenic.vbroker.interceptor.ServerInterceptor	com.inprise.vbroker.interceptor.migration. ServerInterceptorDelegate
com.visigenic.vbroker.interceptor.ServerInterceptorFactory	com.inprise.vbroker.interceptor.migration. ServerInterceptorDelegateFactory
com.visigenic.vbroker.interceptor.ChainServerInterceptorFactory	com.inprise.vbroker.interceptor.migration. ChainServerDelegateFactory
com.visigenic.vbroker.interceptor.ChainServerInterceptorFactoryHelper	com.inprise.vbroker.interceptor.migration. ChainServerDelegateFactoryHelper
com.visigenic.vbroker.orb.ServiceInit	com.inprise.vbroker.interceptor.migration. ServiceInit

Changes to API calls

The migrator changes some API calls to those used in VisiBroker 4.5.

Table 32.3 Changes to API calls

VisiBroker 3.x call	VisiBroker 4.5 call
<code>BOA_init()</code> on instance of <code>org.omg.CORBA.ORB</code>	<code>BOA_init()</code> on instance of <code>com.inprise.vbroker.orb.ORB</code>
<code>bind(repId, objectName, hostName, bindOptions)</code> on instance of <code>org.omg.CORBA.ORB</code>	<code>bind(repId, objectName, hostName, bindOptions)</code> on instance of <code>com.inprise.vbroker.orb.ORB</code>
<code>create_channel(boa, name, debug, maxQueueLength)</code> on instance of <code>com.visigenic.vbroker.services.CosEvent.EventLibrary</code>	<code>create_channel(name, debug, maxQueueLength)</code> on instance of <code>com.inprise.vbroker.CosEvent.EventLibrary</code>
<code>create_channel(boa, name, debug)</code> on instance of <code>com.visigenic.vbroker.services.CosEvent.EventLibrary</code>	<code>create_channel(name, debug)</code> on instance of <code>com.inprise.vbroker.CosEvent.EventLibrary</code>
<code>create_channel(boa, name)</code> on instance of <code>com.visigenic.vbroker.services.CosEvent.EventLibrary</code>	<code>create_channel(name)</code> on instance of <code>com.inprise.vbroker.CosEvent.EventLibrary</code>
<code>create_channel(boa)</code> on instance of <code>com.visigenic.vbroker.services.CosEvent.EventLibrary</code>	<code>create_channel()</code> on instance of <code>com.inprise.vbroker.CosEvent.EventLibrary</code>

Changes from BOA to POA

The migrator does not change code. For this reason, it does not convert VisiBroker 3.x code that was written for BOA to use the POA. Because the BOA is no longer standard in the CORBA ORB, the migrator casts calls that involve the BOA to the Inprise ORB. For information about manually changing code that uses BOA to POA, please see “Manually Migrating BOA to POA” on page 32-5.

Changes in use of interceptors

In some cases there is no way to automatically migrate code and provide functionality similar to that of VisiBroker 3.x. This is the case, for example, with interceptors. In order to be able to use the old code, a special set of interceptors provides at least signature compatibility to the old interceptors.

Note In cases where the migrator cannot determine how to migrate code, it uses migration packages that attempt to wrap the new API using the old semantics. These cases usually arise where the new API differs completely (in logic and behavior) from the old API. In these cases, it is possible that some of the migrated code might not work (the migrated method might not get called) or it might work differently than it did under VisiBroker 3.x.

Invoking the migrator

To run the migrator, type:

```
migrator [options]
```

Migrator options

Table 32.4 Migrator options

Option	Description
-o <file>	Name of output file, or “-” for stdout
-src_dir <path>	Directory in which generated source files are to be placed
-src_suffix <string>	Source filename suffix (.cc)
-list_files	List files written during code generation
-h, -help, -usage, -?	Print this usage information
-version	Display software version numbers
file1 [file2] ...	One or more files to process, or “-” for stdin

Driver options

Table 32.5 Driver options

Option	Description
-J<java option>	Pass the option to JVM directly
-VBJversion	Print VBJ version
-VBJdebug	Print VBJ debug information
-VBJclasspath	Specify classpath, precedes CLASSPATH env variable
-VBJprop <name>[=<value>]	Pass name/value pair to Java VM
-VBJjavavm <jvmpath>	Specify Java VM path
-VBJaddJar <jarfile>	Append jarfile to the CLASSPATH before execing VM

Using migrated code

In order to use the code which is migrated with the migrator, the user needs to add migration.jar to the classpath. This jar file contains the migration classes that are specific to migration for the Visi Broker 4.x API.

Manually Migrating BOA to POA

Class names have changed from previous versions of VisiBroker. Be sure to update your source files to point to the most recent class names. The following tables illustrate these name changes using an example class name.

Table 32.6 Class name changes

Old class name	New class name
<code>_st_Account</code>	<code>_AccountStub</code>
<code>_st_AccountManager</code>	<code>_AccountManagerStub</code>
<code>_AccountImplBase</code>	<code>AccountPOA</code>
<code>_AccountManagerImplBase</code>	<code>AccountManagerPOA</code>
<code>_tie_Account</code>	<code>AccountPOATie</code>
<code>_tie_AccountManager</code>	<code>AccountManagerPOATie</code>

Looking at an example

The `examples/boa/boa2poa` directory contains an example that shows how to update your BOA to the equivalent POA code.

In this example, the BOA code in `Server.java` was updated to POA by:

- Obtaining a reference to the root POA instead of initializing the BOA
- Setting the appropriate POA policies to mimic the BOA characteristics
- Defining the servant (the POA has a different definition of a servant than the BOA)
- Activating the POA manager (no equivalent step for the BOA)
- Waiting for incoming requests through `orb.run()` instead of `boa.impl_is_ready()`

Obtaining a reference to the root POA

When using the BOA, a reference to the BOA was obtained through `orb.BOA_init()`. With the POA, however, you obtain a reference to the root POA. You do this by using `orb.resolve_initial_references("RootPOA").resolve_initial_references` returns a value of type `CORBA.Object` which you then narrow to the desired type.

Code sample 32.1 Obtaining a reference to the rootPOA

```
POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

Setting the POA policies

The characteristics of a POA are defined by the policies set for that POA. Each POA has its own set of policies; POAs can not inherit policies from other POAs.

In this example, persistent objects are used. With the BOA, persistent objects are those which have a specific instance name and are registered with the Smart Agent. A single BOA can support both persistent and transient objects. Under the POA, a persistent object is one that lives past the process that creates them. A single POA can support either persistent object or transient objects, not both. The supported object

type is set by the POA policy. Since the root POA supports transient objects (by default), a new POA must be created to support persistent objects.

Note You can not change the policies of a POA once it is created.

To support persistent objects, set the Lifespan policy to `PERSISTENT`. This example also sets the Bind Support policy (a VisiBroker-specific policy) to `BY_INSTANCE`. This policy registers all active objects with the Smart Agent instead of just the POAs (the default).

Once the appropriate policies have been set, a new POA can be created with `create_POA()`.

Code sample 32.2 Setting the POA policies

```
org.omg.CORBA.Any any = orb.create_any();
BindSupportPolicyValueHelper.insert(any, BindSupportPolicyValue.BY_INSTANCE);
org.omg.CORBA.Policy bsPolicy =
    orb.create_policy(com.inprise.vbroker.PortableServerExt.BIND_SUPPORT_POLICY_TYPE,
        value, any);

org.omg.CORBA.Policy[] policies = {
    rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT), bsPolicy};
// Create myPOA with the right policies
POA myPOA = rootPOA.create_POA( "bank_agent_poa", rootPOA.the_POAManager(),
    policies );
```

Defining the servant

With the BOA, a servant is a CORBA object. In this example, the account manager object is created and then exported with `obj_is_ready()`.

With the POA, a servant is a programming object that provides the implementation of an abstract object. A servant is not a CORBA object. Under the POA scenario, the servant is created and then activated with a specific ID. You can use this ID to obtain the object reference.

Code sample 32.3 Defining and activating a servant

```
// Create the servant
AccountManagerImpl managerServant = new AccountManagerImpl();
// Decide on the ID for the servant
byte[] managerId = "BankManager".getBytes();
// Activate the servant with the ID on myPOA
myPOA.activate_object_with_id(managerId, managerServant);
```

Activating the POA manager

A POA Manager is an object that controls how a POA processes requests. By default, POA Managers are created in a holding state. In this state, all requests are routed to a holding queue and are not processed. To allow requests to be dispatched, the POA Manager associated with the POA must be changed from the holding state to an active state.

This is a new step required for the POA. There is no equivalent step for the BOA.

Code sample 32.4 Activating the POA manager

```
rootPOA.the_POAManager().activate();
```

Waiting for incoming requests

With the BOA, `impl_is_ready()` is called in order to wait for requests from clients. With the POA, use `orb.run()`.

Code sample 32.5 Waiting for incoming requests

```
orb.run();
```

Looking at the other files

The `AccountImpl` and `AccountManagerImpl` class changes are much simpler. Most of the changes simply involve pointing to the new classes.

Mapping BOA types to POA policies

The following table shows how to set your POA policies to mimic BOA behavior.

Table 32.7 Mapping BOA types to POA policies

	Transient BOA	Persistent BOA
TPOOL	Server Engine policy with TPOOL dispatcher LifeCycle property set to TRANSIENT	Server Engine policy with TPOOL dispatcher LifeCycle property set to PERSISTENT IDAssignment policy set to USER_ID BindSupport policy set to BY_INSTANCE
TSESSION	Server Engine policy with TSESSION dispatcher LifeCycle property set to TRANSIENT	Server Engine policy with TSESSION dispatcher LifeCycle property set to PERSISTENT IDAssignment policy set to USER_ID BindSupport policy set to BY_INSTANCE
Service-activated objects	LifeCycle property set to TRANSIENT Request Processing policy to USE_SERVANT_MANAGER Implicit Activation policy set to IMPLICIT_ACTIVATION	LifeCycle property set to PERSISTENT Request Processing policy to USE_SERVANT_MANAGER Implicit Activation policy set to IMPLICIT_ACTIVATION

Migrating to new package names

The following table shows how VisiBroker 3.x package name prefixes map to the latest release.

VisiBroker 3.x package names	VisiBroker 4.5 package names
<code>com.visigenic.vbroker</code>	<code>com.inprise.vbroker</code>
<code>com.visigenic.vbroker.services.CosEvent</code>	<code>com.inprise.vbroker.CosEvent</code>
<code>com.visigenic.vbroker.services.CosNaming</code>	<code>com.inprise.vbroker.naming</code>

Migrating to new class names

The following table shows how VisiBroker 3.x class names map to the latest release.

VisiBroker 3.x class names	VisiBroker 4.5 class names
org.omg.CORBA.BOA	com.inprise.vbroker.CORBA.BOA
org.omg.CORBA.DynamicImplementation	com.inprise.vbroker.CORBA.migration. DynamicImplementation
com.visigenic.vbroker.interceptor. BindInterceptor	com.inprise.vbroker.interceptor.migration. BindInterceptorDelegate
com.visigenic.vbroker.interceptor. ChainBindInterceptor	com.inprise.vbroker.interceptor. ChainBindDelegateFactory
com.visigenic.vbroker.interceptor. ChainBindInterceptorHelper	com.inprise.vbroker.interceptor.migration. ChainBindDelegateFactoryHelper
com.visigenic.vbroker.interceptor. ClientInterceptor	com.inprise.vbroker.interceptor.migration. ClientInterceptorDelegate
com.visigenic.vbroker.interceptor. ClientInterceptorFactory	com.inprise.vbroker.interceptor.migration. ClientInterceptorFactory
com.visigenic.vbroker.interceptor. ChainClientInterceptorFactory	com.inprise.vbroker.interceptor.migration. ChainClientDelegateFactory
com.visigenic.vbroker.interceptor. ChainClientInterceptorFactoryHelper	com.inprise.vbroker.interceptor.migration. ChainClientDelegateFactoryHelper
com.visigenic.vbroker.interceptor. ServerInterceptor	com.inprise.vbroker.interceptor.migration. ServerInterceptorDelegate
com.visigenic.vbroker.interceptor. ServerInterceptorFactory	com.inprise.vbroker.interceptor.migration. ServerInterceptorDelegateFactory
com.visigenic.vbroker.interceptor. ChainServerInterceptorFactory	com.inprise.vbroker.interceptor.migration. ChainServerDelegateFactory
com.visigenic.vbroker.interceptor. ChainServerInterceptorFactoryHelper	com.inprise.vbroker.interceptor.migration. ChainServerDelegateFactory Helper
com.visigenic.vbroker.orb.ServiceInit	com.inprise.vbroker.interceptor.migration. ServiceInit

Migrating to new API calls

The following table shows how VisiBroker 3.x API calls map to the latest release.

VisiBroker 3.x API calls	VisiBroker 4.5 API calls
BOA_init() on instance of org.omg.CORBA.ORB	BOA_init() on instance of com.inprise.vbroker.orb.ORB
bind(repId, objectName, hostName, bindOptions) on instance of org.omg.CORBA.ORB	bind(repId, objectName, hostName, bindOptions) on instance of com.inprise.vbroker.orb.ORB

VisiBroker 3.x API calls	VisiBroker 4.5 API calls
<code>create_channel(boa, name, debug, maxQueueLength)</code> on instance of <code>com.visigenic.vbroker.services.CosEvent.EventLibrary</code>	<code>create_channel(name, debug, maxQueueLength)</code> on instance of <code>com.inprise.vbroker.CosEvent.EventLibrary</code>
<code>create_channel(boa, name, debug)</code> on instance of <code>com.visigenic.vbroker.services.CosEvent.EventLibrary</code>	<code>create_channel(name, debug)</code> on instance of <code>com.inprise.vbroker.CosEvent.EventLibrary</code>
<code>create_channel(boa, name)</code> on instance of <code>com.visigenic.vbroker.services.CosEvent.EventLibrary</code>	<code>create_channel(name)</code> on instance of <code>com.inprise.vbroker.CosEvent.EventLibrary</code>
<code>create_channel(boa)</code> on instance of <code>com.visigenic.vbroker.services.CosEvent.EventLibrary</code>	<code>create_channel()</code> on instance of <code>com.inprise.vbroker.CosEvent.EventLibrary</code>

For more information on the new package names, classes, and API calls, see the VisiBroker for Java *Reference*.

Migrating interceptors

The preferred method for migrating interceptors to VisiBroker 4.5 is to use the new VisiBroker 4.5 interceptors.

Note: Although VisiBroker 4.5 does provide wrappers that allow you to migrate your old interceptor code virtually unchanged (described below), the VisiBroker 4.5 wrappers for 3.x code do not provide functionality comparable to that of VisiBroker 4.5 interceptors.

Using VisiBroker 3.x interceptors

Although VisiBroker 4.5 ensures that method signatures of VisiBroker 3.x interceptors need not change, installation and initialization procedures for old-style interceptors are changed.

Installing VisiBroker 3.x interceptors

In order to use old-style interceptors with VisiBroker 4.5:

- 1 When the migration tool migrates the interceptors, it adds the following statement to the imports declaration in the Java file:

```
import com.inprise.vbroker.interceptor.migration.*;
```

If you are migrating the interceptors manually, you need to add this statement to the imports declaration yourself.

2 Rename the following interceptors as shown in the following table:

Table 32.9 Renaming interceptors

Old style name	New style name
<code>com.(inprise borland visigenic).vbroker.interceptor.BindInterceptor</code>	<code>com.inprise.vbroker.interceptor.migration.BindInterceptorDelegate</code>
<code>com.(inprise borland visigenic).vbroker.interceptor.ChainBindInterceptor</code>	<code>com.inprise.vbroker.interceptor.migration.ChainBindDelegateFactory</code>
<code>com.(inprise borland visigenic).vbroker.interceptor.ClientInterceptor</code>	<code>com.inprise.vbroker.interceptor.migration.ClientInterceptorDelegate</code>
<code>com.(inprise borland visigenic).vbroker.interceptor.ServerInterceptor</code>	<code>com.inprise.vbroker.interceptor.migration.ServerInterceptorDelegate</code>
<code>com.(inprise borland visigenic).vbroker.interceptor.ClientInterceptorFactory</code>	<code>com.inprise.vbroker.interceptor.migration.ClientInterceptorFactory</code>
<code>com.(inprise borland visigenic).vbroker.interceptor.ServerInterceptorFactory</code>	<code>com.inprise.vbroker.interceptor.migration.ServerInterceptorFactory</code>

Migrating BindInterceptors

The VisiBroker 4.5, wrappers simulate the real BindInterceptor.

In previous versions, to add a BindInterceptor, you would:

- 1 Get the reference to the ChainBindInterceptor by calling `ORB.resolve_initial_references("ChainBindInterceptor")`.
- 2 Add the new interceptor to the chain.

To use your VisiBroker 3.x bind interceptor code in VisiBroker 4.5, you should instead:

- 1 Get the reference to `interceptor_migration.ChainBindDelegateFactory` by calling `ORB.resolve_initial_references("ChainBindInterceptor")`.
- 2 Then, create and add your `interceptor_migration.BindInterceptorDelegate` (rather than the `interceptor.BindInterceptor` that you used in VisiBroker 3.x) to the chain.

Migrating client- and server-side interceptors

In previous versions, to add a ClientInterceptor or a ServerInterceptor, you would:

First, implement the interface `Interceptor.ClientInterceptorFactory` or `Interceptor.ServerInterceptorFactory`. This interface provides methods for creating user-implemented ClientInterceptors and ServerInterceptors. You could then obtain a reference to the `ChainClientDelegateFactory` or the `ChainServerDelegateFactory` and, using these, you can add your own interceptors to the chain.

Under VisiBroker 4.5, you should instead:

Implement `interceptor_migration.ClientInterceptorDelegate` or `interceptor_migration.ServerInterceptorDelegate`. Then, obtain a reference to the `interceptor_migration.ClientInterceptorFactory` or the `interceptor_migration.ServerInterceptorFactory`. These methods return the instance of the appropriate `InterceptorDelegate`.

After you have access to the client factory, server factory or both, you can install your client- or server-side interceptors into the appropriate factory chain. To do so, call `ORB.resolve_initial_references("ChainClientDelegateFactory")` or `ORB.resolve_initial_references("ChainServerDelegateFactory")` Once you have the references, you can use the `Add()` method to add to the chain. (This procedure is unchanged from VisiBroker 3.x.)

Using object activators

This chapter describes using the VisiBroker object activators.

In this release of VisiBroker, the Portable Object Adaptor (POA) supports the features that were provided by the BOA in VisiBroker 3.x releases. For backward compatibility reasons, you may still use the object activators as described in this chapter with your code. For more information on how to use the BOA activators with this release, see Chapter 31, “Using the BOA with VisiBroker 4.x,” and Chapter 32, “Migrating VisiBroker code.”

Deferring object activation

You can defer activation of multiple object implementations, using service activation, with a single `Activator` when a server needs to provide implementations for a large number of objects.

Activator interface

You can derive your own interface from the `Activator` interface. This allows you to implement the `activate` and `deactivate` methods that the ORB will use for the `DBObjectImpl` object. You can then delay the instantiation of the `AccountImpl` object until the BOA receives a request for that object. It also allows you to provide clean-up processing when the BOA deactivates the object.

Code sample 33.1 shows the `Activator` interface, which provides methods invoked by the BOA to activate and deactivate an ORB object.

Code sample 33.1 Activator interface

```
package com.inprise.vbroker.extension;
public interface Activator {
    public abstract org.omg.CORBA.Object activate(ImplementationDef impl);
    public abstract void deactivate(org.omg.CORBA.Object obj, ImplementationDef impl);
}
```

Code sample 33.2 shows how to create an `Activator` for the `DBObjectImpl` interface.

Code sample 33.2 Deriving an `DBActivator` interface, implementing the activate and deactivate methods

```
// Server.java
import com.inprise.vbroker.extension.*;
...
class DBActivator implements Activator {
    private static int _count;
    private com.inprise.vbroker.CORBA.BOA _boa;

    public DBActivator(com.inprise.vbroker.CORBA.BOA boa) {
        _boa = boa;
    }
    public org.omg.CORBA.Object activate(
        com.inprise.vbroker.extension.ImplementationDef impl) {
        System.out.println("Activator called " + ++_count + " times");
        byte[] ref_data = ((ActivationImplDef) impl).id();
        DBObjImpl obj = new DBObjImpl(new String(ref_data));
        _boa.obj_is_ready(obj);
        return obj;
    }
    public void deactivate(org.omg.CORBA.Object obj,
        com.inprise.vbroker.extension.ImplementationDef impl) {
        // nothing to do here...
    }
}
...
```

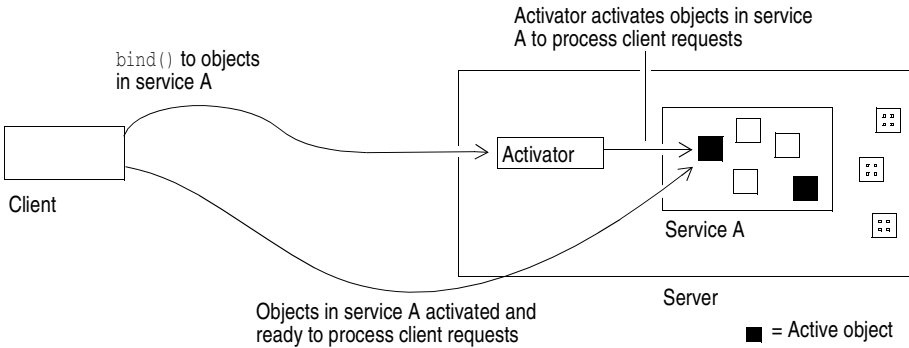
Using the service activation approach

Service activation can be used when a server needs to provide implementations for a large number of objects (commonly thousands of objects, possibly millions) but only a small number of implementations need to be active at any specific time. The server can supply a single `Activator` which is notified whenever any of these subsidiary objects are needed. The server can also deactivate these objects when they are not in use.

For example, you might use service activation for a server that loads objects implementations whose states are stored in a database. The `Activator` is responsible for loading all objects of a given type or logical distinction. When ORB requests are made on the references to these objects, the `Activator` is notified and creates a new implementation whose state is loaded from the database. When the `Activator`

determines that the object should no longer be in memory and if the object had been modified, it writes the object's state to the database and releases the implementation.

Figure 33.1 Diagram showing the process of deferring activation for a service



Deferring object activation using service activators

Assuming the objects that will make up the service have already been created, the following steps are required to implement a server that uses service activation:

- 1 Define a service name that describes all objects activated and deactivated by the Activator.
- 2 Provide implementations for the interface which are service objects, rather than persistent objects. This is done when the object constructs itself as an activatable part of a service.
- 3 Implement the `Activator` which creates the object implementations on demand. In the implementation, you derive an `Activator` interface from `extension::Activator`, overriding the `activate` and `deactivate` methods.
- 4 Register the service name and the `Activator` interface with the BOA.

Example of deferred object activation for a service

The following sections describe the `odb` example for service activation which is located in the `examples/boa/odb` directory of your VisiBroker installation. The examples directory contains the following files:

Table 33.1 Files in the `odb` example for service activation

Name	Description
<code>odb.idl</code>	IDL for DB and DBObject interfaces.
<code>Server.java</code>	Creates objects using service activators, returns IORs for the objects, and deactivates the objects.
<code>Creator.java</code>	Calls the DB interface to create 100 objects and stores the resulting stringified object references in a file (<code>objref.out</code>).

Table 33.1 Files in the odb example for service activation (continued)

Name	Description
Client.java	Reads the stringified object references to the objects from a file and makes calls on them, causing the activators in the server to create the objects.
Makefile	When <code>make</code> or <code>nmake</code> (on Windows) is invoked in the <code>odb</code> subdirectory, builds the following client and server programs: <div>Server Creator Client</div>

The `odb` example shows how an arbitrary number of objects can be created by a single service. The service alone is registered with the BOA, instead of each individual object, with the reference data for each object stored as part of the IOR. This facilitates object-oriented database (OODB) integration, since you can store object keys as part of an object reference. When a client calls for an object that has not yet been created, the BOA calls a user-defined `Activator`. The application can then load the appropriate object from persistent storage.

In this example, an `Activator` is created that is responsible for activating and deactivating objects for the service named “`DBService`.” References to objects created by this `Activator` contain enough information for the ORB to relocate the `Activator` for the `DBService` service, and for the `Activator` to recreate these objects on demand.

The `DBService` service is responsible for objects that implement the `DBObject` interface. An interface (contained in `odb.idl`) is provided to enable manual creation of these objects.

odb.idl interface

The `odb.idl` interface enables manual creation of objects that implement the `DBObject` odb interface.

IDL sample 33.1 db.idl interface

```
interface DBObject {
    string get_name();
};

typedef sequence<DBObject> DBObjectSequence;

interface DB {
    DBObject create_object(in string name);
};
```

The `DBObject` interface represents an object created by the `DB` interface, and can be treated as a service object.

`DBObjectSequence` is a sequence of `DBObject`s. The server uses this sequence to keep track of currently active objects.

The `DB` interface creates one or more `DBObject`s using the `create_object` operation. The objects created by the `DB` interface can be grouped together as a service.

Implementing a service activator

Normally, an object is activated when a server instantiates the Java classes implementing the object, and then calls `obj_is_ready` followed by `impl_is_ready`. To defer activation of objects, it is necessary to gain control of the `activate` method that the BOA invokes during object activation. You obtain this control by deriving a new class from `com.inprise.vbroker.extension.Activator` and overriding the `activate` method, using the overridden `activate` method to instantiate Java classes specific to the object.

In the `odb` example, the `DBActivator` class derives from `com.inprise.vbroker.extension.Activator`, and overrides the `activate` and `deactivate` methods. The `DBObject` is constructed in the `activate` method.

Code sample 33.3 Example of overriding activate and deactivate

```
// Server.java
class DBActivator implements Activator {
    private static int _count;
    private com.inprise.vbroker.CORBA.BOA _boa;
    public DBActivator(com.inprise.vbroker.CORBA.BOA boa) {
        _boa = boa;
    }
    public org.omg.CORBA.Object activate(
        com.inprise.vbroker.extension.ImplementationDef impl) {
        System.out.println("Activator called " + ++_count + " times");
        byte[] ref_data = ((ActivationImplDef) impl).id();
        DBObjectImpl obj = new DBObjectImpl(new String(ref_data));
        _boa.obj_is_ready(obj);
        return obj;
    }
    public void deactivate(org.omg.CORBA.Object obj, ImplementationDef impl) {
        // nothing to do here...
    }
}
```

As shown in Code sample 33.4, the `DBActivator` class creates an object based on its `ReferenceData` parameter. When the BOA receives a client request for an object under the responsibility of the Activator, the BOA invokes the `activate` method on the Activator. When calling this method, the BOA uniquely identifies the activated object implementation by passing the Activator an `ImplementationDef` parameter—from which the implementation can obtain `ReferenceData`, the requested object's unique identifier.

Code sample 33.4 Example of implementing a service activator

```
public org.omg.CORBA.Object activate(ImplementationDef impl) {
    System.out.println("Activator called " + ++_count + " times");
    byte[] ref_data = ((ActivationImplDef) impl).id();
    DBObjectImpl obj = new DBObjectImpl(new String(ref_data));
    _boa.obj_is_ready(obj);
    return obj;
}
```

Instantiating the service activator

As shown in Code sample 33.5 below, the `DBActivator` service activator is created and registered with the BOA using the `impl_is_ready` call in the main server program. The `DBActivator` service activator is responsible for all objects that belong to the `DBService` service. All requests for objects of the `DBService` service are directed through the `DBActivator` service activator. All objects activated by this service activator have references that inform the ORB that they belong to the `DBService` service.

Code sample 33.5 Example of instantiating the service activator

```
public static void main(String[] args) {
    org.omg.CORBA.ORB orb = ORB.init(args, null);
    com.inprise.vbroker.CORBA.BOA boa = ((com.inprise.vbroker.ORB )orb).BOA_init();
    DB db = new DBImpl("Database Manager");
    boa.obj_is_ready(db);
    boa.impl_is_ready("DBService", new DBActivator(boa));
}
```

Note that the call to `impl_is_ready` is a variation on the usual call to `impl_is_ready`—it takes two arguments:

- Service name.
- Instance of an `Activator` interface that will be used by the BOA to activate objects belonging to the service.

Using a service activator to activate an object

Whenever an object is constructed, `obj_is_ready` must be explicitly invoked in `activate`. There are two calls to `obj_is_ready` in the server program. One call occurs when the server creates a service object and returns an IOR to the creator program.

Code sample 33.6 First server call to `obj_is_ready`

```
public DBObject create_object(String name) {
    System.out.println("Creating: " + name);
    DBObject dbObject = new DBObjectImpl(name);
    _boa().obj_is_ready(dbObject, "DBService", name.getBytes());
    return dbObject;
}
```

The second occurrence of `obj_is_ready` is in `activate`, and this needs to be explicitly called. Refer to Code sample 33.4 on page 33-5 to see this second call in context.



CORBA exceptions

This appendix provides information about CORBA exceptions that can be thrown by the VisiBroker ORB, and explains possible causes for VisiBroker throwing them.

The following table lists CORBA exceptions, and explains reasons why the VisiBroker ORB might throw them.

Table A.1 CORBA exceptions and possible causes

Exception	Explanation	Possible causes
<code>CORBA::BAD_CONTEXT</code>	An invalid context has been passed to the server.	An operation may raise this exception if a client invokes the operation but the passed context does not contain the context values required by the operation.
<code>CORBA::BAD_INV_ORDER</code>	The necessary prerequisite operations have not been called prior to the offending operation request.	<p>An attempt to call the <code>CORBA::Request::get_response()</code> or <code>CORBA::Request::poll_response()</code> methods may have occurred prior to actually sending the request.</p> <p>An attempt to call the <code>exception::get_client_info()</code> method may have occurred outside of the implementation of a remote method invocation. This function is only valid within the implementation of a remote invocation.</p> <p>An operation was called on a n ORB that was already shut down.</p>
<code>CORBA::BAD_OPERATION</code>	An invalid operation has been performed.	<p>A server throws this exception if a request is received for an operation that is not defined on that implementation's interface. Ensure that the client and server were compiled from the same IDL.</p> <p>The <code>CORBA::Request::return_value()</code> method throws this exception if the request was not set to have a return value. If a return value is expected when making a DII call, be sure to set the return value type by calling the <code>CORBA::Request::set_return_type()</code> method.</p>

Table A.1 CORBA exceptions and possible causes (continued)

Exception	Explanation	Possible causes
CORBA::BAD_PARAM	A parameter passed to the ORB is invalid.	<p>Sequences throw <code>CORBA::BAD_PARAM</code> if an access is attempted to an invalid index. Make sure you use the <code>length()</code> method to set the length of the sequence before storing or retrieving elements of the sequence.</p> <p>ORB throws this exception if <code>null</code> reference is passed).</p> <p>An attempt was made to send a value that is out of range for an enumerated data type.</p> <p>An attempt may have been made to construct a <code>TypeCode</code> with an invalid <code>kind</code> value.</p> <p>An attempt may have been made to insert a <code>null</code> object reference into an <code>Any</code>.</p> <p>Using the DII and one way method invocations, an <code>OUT</code> argument may have been specified. An interface repository thrown this exception if an argument passed into an IR object's operation conflicts with its existing settings. See the compiler errors for more information.</p>
CORBA::BAD_TYPECODE	The ORB has encountered a malformed type code.	
CORBA::CODESET_INCOMPATIBLE	Communication between client and server native code sets fails because the code sets are incompatible.	The code sets used by the client and server cannot work together. For instance, the client uses ISO 8859-1 and the server uses the Japanese code set.
CORBA::COMM_FAILURE	Communication is lost while an operation is in progress, after the request was sent by the client but before the reply has been returned.	This exception is raised if communication is lost while an operation is in progress, after the request was sent by the client, but before the reply from the server has been returned to the client.
CORBA::DATA_CONVERSION	The ORB cannot convert the representation of marshaled data into its native representation or vice-versa.	An attempt to marshal Unicode characters with <code>Output.write_char()</code> or <code>Output.write_string</code> fails.
CORBA::IMP_LIMIT	An implementation limit was exceeded in the ORB run time.	<p>The ORB may have reached the maximum number of references it can hold simultaneously in an address space.</p> <p>The size of the parameter may have exceeded the allowed maximum.</p> <p>The maximum number of running clients and servers has been exceeded.</p>

Table A.1 CORBA exceptions and possible causes (continued)

Exception	Explanation	Possible causes
<code>CORBA::INITIALIZE</code>	A necessary initialization has not been performed.	The <code>ORB_init()</code> method may not have been called. All clients must call the <code>ORB_init()</code> method prior to performing any ORB-related operations. This call is typically made immediately upon program startup at the top of the main routine.
<code>CORBA::INTERNAL</code>	An internal ORB error has occurred.	An internal ORB error may have occurred. For instance, the internal data structures of the ORB may have been corrupted.
<code>CORBA::INTF_REPOS</code>	An instance of the Interface Repository could not be located.	If an object implementation cannot locate an interface repository during an invocation of the <code>get_interface()</code> method, this exception will be thrown to the client. Ensure that an Interface Repository is running, and that the requested object's interface definition has been loaded into the Interface Repository.
<code>CORBA::INV_FLAG</code>	An invalid flag was passed to an operation.	A Dynamic Invocation Interface request was created with an invalid flag.
<code>CORBA::INV_IDENT</code>	An IDL identifier is syntactically invalid.	An identifier passed to the interface repository is not well formed. An illegal operation name is used with the Dynamic Invocation Interface.
<code>CORBA::INV_OBJREF</code>	An invalid object reference has been encountered.	The ORB will throw this exception if an object reference is obtained that contains no usable profiles. The <code>ORB::string_to_object()</code> method will throw this exception if the stringified object reference does not begin with the characters "IOR:".
<code>CORBA::INV_POLICY</code>	An invalid policy override has been encountered.	This exception can be thrown from any invocation. It can be raised when an invocation cannot be made due to an incompatibility between policy overrides that apply to the particular invocation.
<code>CORBA::INVALID_TRANSACTION</code>	A request carried an invalid transaction context.	See your transaction service documentation for more information on this exception.
<code>CORBA::MARSHAL</code>	Error marshalling parameter or result.	A request or reply from the network is structurally invalid. This error typically indicates a bug in either the client-side or server-side run time. For example, if a reply from the server indicates that the message contains 1000 bytes, but the actual message is shorter or longer than 1000 bytes, the ORB raises this exception. A <code>MARSHAL</code> exception can also be caused by using the DII or DSI incorrectly. For example, if the type of the actual parameters sent does not agree with IDL signature of an operation.

Table A.1 CORBA exceptions and possible causes (continued)

Exception	Explanation	Possible causes
CORBA::NO_IMPLEMENT	The requested object could not be located.	Indicates that even though the operation that was invoked exists (it has an IDL definition), no implementation for that operation exists. For example, a NO_IMPLEMENTATION is raised when a server doesn't exist or is not running when a client initiates a bind.
CORBA::NO_MEMORY	The ORB runtime has run out of memory.	
CORBA::NO_PERMISSION	The caller has insufficient privileges to complete an invocation.	
CORBA::NO_RESOURCES	A necessary resource could not be acquired.	If a new thread cannot be created, this exception will be thrown. A server will throw this exception when a remote client attempts to establish a connection if the server cannot create a socket—for example, if the server runs out of file descriptors. The minor code contains the system error number obtained after the server's failed <code>::socket()</code> or <code>::accept()</code> call. A client will similarly throw this exception if a <code>::connect()</code> call fails due to running out of file descriptors.
CORBA::NO_RESPONSE	A client attempts to retrieve the result of a deferred synchronous call, but the response for the request is not yet available.	
CORBA::OBJ_ADAPTER	An administrative mismatch has occurred.	A server has attempted to register itself with an implementation repository under a name that already is in use, or is unknown to the repository. The POA has raised an OBJ_ADAPTER error due to problems with the application's servant managers.
CORBA::OBJECT_NOT_EXIST	The requested object does not exist.	A server throws this exception if an attempt is made to perform an operation on an implementation that does not exist within that server. This will be seen by the client when attempting to invoke operations on deactivated implementations. For instance, if an attempt to bind to an object fails, or an auto-rebind fails, OBJECT_NOT_EXIST will be raised.
CORBA::PERSIST_STORE	A persistent storage failure has occurred.	Attempts to establish a connection to a database has failed, or the database is corrupt.

Table A.1 CORBA exceptions and possible causes (continued)

Exception	Explanation	Possible causes
<code>CORBA::REBIND</code>	The client has received an IOR which conflicts with QOS policies.	Thrown anytime the client gets an IOR which will conflict with the QOS policies that have been set. If the <code>RebindPolicy</code> has a value of <code>NO_REBIND</code> , <code>NO_CONNECT</code> , or <code>VB_NOTIFY_REBIND</code> and an invocation on a bound object reference results in an object forward or a location forward message.
<code>CORBA::TRANSACTION_REQUIRED</code>	The request carried a null transaction context, but an active transaction is required.	See your transaction service documentation for more information on this exception.
<code>CORBA::TRANSACTION_ROLLEDBACK</code>	The transaction associated with a request has already been rolled back, or marked for roll back.	See your transaction service documentation for more information on this exception.
<code>CORBA::TRANSIENT</code>	An error has occurred, but the ORB believes it is possible to retry the operation.	A communications failure may have occurred and the ORB is signalling that an attempt should be made to rebind to the server with which communications have failed. This exception will not occur if the <code>BindOptions</code> are set to false with the <code>enable_rebind()</code> method, or the <code>RebindPolicy</code> is properly set.
<code>CORBA::UNKNOWN</code>	The ORB could not determine the thrown exception.	<p>The server throws something other than a correct exception, such as a Java runtime exception.</p> <p>There is an IDL mismatch between the server and the client, and the exception is not defined in the client program.</p> <p>In DII, if the server throws an exception not known to the client at the time of compilation and the client did not specify an exception list for the <code>CORBA::Request</code>. Set the property <code>vbroker.orb.warn=2</code> on the server to see which runtime exception caused the problem.</p>

Table A.2 CORBA exception minor codes

System exception	Minor code	Explanation
<code>BAD_PARAM</code>	1	Failure to register, unregister, or lookup the value factory
	2	RID already defined in the interface repository
	3	Name already used in the context in the interface repository
	4	Target is not a valid container
	5	Name clash in inherited context
	6	Incorrect type for abstract interface
<code>MARSHAL</code>	1	Unable to locate value factory
<code>NO_IMPLEMENT</code>	1	Missing local value implementation
	2	Incompatible value implementation version

Table A.2 CORBA exception minor codes (continued)

System exception	Minor code	Explanation
BAD_INV_ORDER	1	Dependency exists in the interface repository preventing the destruction of the object
	2	Attempt to destroy indestructible objects in the interface repository
	3	Operation would deadlock
	4	ORB has shut down
OBJECT_NOT_EXIST	1	Attempt to pass an unactivated (unregistered) value as an object reference

Glossary

This is a glossary of terms used in Inprise products.

activation

Process of preparing an object to receive requests.

API (application program interface)

A set of operations which allows a (client) program to access functionality contained in a library or another program, possibly a server.

attribute

An attribute is a property of an object. For example, a Point object might have two coordinate attributes, X and Y.

applet

A small, platform-independent program designed to be downloaded and executed by a Java-enabled web browser. An applet is downloaded from a host at runtime and is not “trusted”—because of security restrictions placed on the applet by the web browser, it cannot open local files or establish network connections.

application

A computer program designed to help people perform a certain type of work. Depending on the work for which it was designed, an application can manipulate text, numbers, graphics, or a combination of these elements.

bind (NamingService)

The process of associating a Name with a remote object in a server application, so that a client application can resolve the Name and obtain a reference to the remote object.

bind (VisiBroker)

The process of establishing a connection to a server hosting an object we are interested in.

class

A class is a data type which declares what attributes and operations an instantiated object will have.

client/server

A programming strategy in which two programs cooperate with one another using some common and conventional protocol. For example, on the worldwide web, the browser is the client software, the web server is the server software, and HTTP is the protocol. Clients send requests to servers, and servers send replies to clients.

component

A chunk or object of a distributed application.

CORBA (common object request broker architecture)

An open, object-oriented, standard architecture developed by the OMG for the interoperability of distributed objects on different platforms, under different operating systems and implemented in different programming languages.

distributed application

An application whose components are distributed across multiple computers on a network but which seem to be running on the user's computer.

distributed objects

Software modules that are designed to work together but reside in multiple computer systems throughout the organization. A program in one machine sends a message to an object in a remote machine to perform some processing. The results are sent back to the calling machine.

Dynamic Invocation Interface (DII)

An API that allows a client to make dynamic invocations on remote CORBA objects. It is used if at compile time a client does not have knowledge about an object it wants to invoke. Once an object is discovered, the client program can obtain a definition of it, issue a parameterized call to it, and receive a reply from it, all without having a type-specific client stub for the remote object.

Dynamic Skeleton Interface (DSI)

An API that provides a way to deliver requests from an ORB to an object implementation when the type of the object implementation is not known at compile time. DSI, which is the server side analog to the client side DII, makes it possible for the application programmer to inspect the parameters of an incoming request to determine a target object and method.

failover

Having more than one system which may be used as backup in case one of the systems fail.

HTML (hypertext markup language)

An SGML application used to specify the structure of a hypertext (web) document.

HTTP (hypertext transport protocol)

A protocol used by worldwide web client/server applications to connect and transfer HTML documents.

IDL (interface definition language)

A high-level, programming language independent, declarative language for defining the interface of a distributed object.

IDL compiler

A compiler which translates an IDL specification into programming language specific stub and skeleton files which are used to implement distributed objects.

IDL file

A plain text file which declares modules and interfaces in IDL.

IIOP (Internet Inter-ORB protocol)

A TCP/IP-based protocol developed by the OMG. The IIOP enables two or more ORBs to work in conjunction to provide requests to objects.

interface

The set of public attributes and operations (or signature) which a (server) object exposes to a (client) object.

interface repository

A service that contains all the registered component interfaces, the methods they support, and the parameters they require. The IFR stores, updates, and manages object interface definitions. Programs may use the IFR APIs to access and update this information.

JNDI (Java naming and directory interface)

The Java Naming and Directory Interface (JNDI) is a standard extension to the Java platform, providing Java-enabled applications with a unified interface to multiple naming and directory services in the enterprise.

master/slave

The Interoperable Naming service runs master and slave naming service for a failover purposes. The master is the primary service and the slave is the fallback service in general.

method

An operation of an object (the server) which when called by another object (the client) performs some declared behavior.

multithreading

A programming technique whereby an application can be divided into more than one asynchronous time-slice (or thread of execution).

name

A name is a predefined name, an alias, or a convenient handle which is associated with a server object. To bind a name to an object, you use the bind method. To resolve a name (that is, to retrieve an object reference) use the resolve method.

namespace

A collection of names, no two of which are identical.

naming service

A CORBA service that allows CORBA objects to be named by means of binding a name to an object reference. The name binding may be stored in the naming service, and a client may supply the name to obtain the desired object reference.

n-tier

A programming strategy in which *n* programs cooperate with one another using some common and conventional protocol. For example, a client/server application can also be described as a two-tier application.

object

A programming entity which is defined by its properties (attributes) and behaviors (operations). Objects have unique identities and can be distinguished from one another. An object is an instance of a particular class.

object adapter

The ORB component which provides object reference, activation, and state related services to an object implementation.

object implementation

A server process that offers one or more objects which client applications may use.

object reference

A handle to an object, used by a client application to invoke methods on the object.

OMG (Object Management Group)

A consortium of software companies which is charged with the development of the CORBA specification: (see <http://www.omg.org/>).

operation

The method of an object (the server) which when called by another object (the client) performs some declared behavior.

ORB (object request broker)

The ORB allows clients to make and receive requests and responses.

package

A logical collection of Java classes that provide similar or related features.

protocol

A language which defines the requests and replies of client/server objects or applications.

RMI (remote method invocation)

A Java API which allows objects to be instantiated and used in a distributed application.

RPC (remote procedure call)

A strategy which allows procedures to be called from outside the currently running program's memory. RPC allows two or more different programs to interoperate with one another.

scalability

The degree to which a system or application can handle increasing or decreasing demand on system resources without significant performance degradation.

servant

An instance of an object implementation for an IDL interface. The servant object is registered with the ORB so that the ORB knows where to send invocations. It is the servant that performs the services requested when a CORBA object's method is invoked.

server

An object or application which performs a service for other objects or applications (the clients). A server replies to a client's request using a protocol.

service

The functionality of a given server.

SGML (standard generalized markup language)

Abbreviation of Standard Generalized Markup Language, a system for organizing and tagging elements of a document. SGML was developed and standardized by the International Organization for Standards (ISO). SGML itself does not specify any particular formatting; rather, it specifies the rules for tagging elements. These tags can then be interpreted to format elements in different ways.

signature

The set of parameters and their names of a given operation which uniquely identify the operation.

skeleton (file)

An older construct (used prior to VisiBroker 4.0): a serverside file generated from IDL which is to be implemented by the object implementor.

stringification

Converting an object reference to a character string format. Used when an object reference needs to be made persistent to a text file or stored in a database or sent to a client program.

stub (file)

The portion of a client or server program that executes the data marshalling and network transportation routines.

TCP/IP (transport control protocol / internet protocol)

TCP is one of the main protocols in TCP/IP networks. Whereas the IP protocol deals only with packets, TCP enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent.

thread

A thread is a stream of execution within a process. In a multithreaded environment, multiple tasks can execute concurrently within the same application.

transaction server

A server which supports transactional semantics, (for example, commit or rollback).

XML (extensible markup language)

Extensible Markup Language. A specification developed by the World Wide Web Consortium (W3C). XML is a subset of the SGML document language, designed especially for Web documents.

Index

Symbols

... ellipsis 1-5
:: scope resolution operator 23-5
[] brackets 1-5
| vertical bar 1-5

A

abstract
 interfaces 28-7
 valuetypes 28-2
accessing
 Implementation Repository 12-7
 Interface Repository 12-10
 Location Service 12-2
 Naming Service 12-4
 value of a DynAny 27-3
account.idl, files produced from
 account_c.cc 4-4
activate() method 32-1
activating
 the POA manager 31-6
activating objects
 changing characteristics dynamically 20-10
 deferring 32-1, 32-3
 deferring object activation with service
 activators 32-3
 OAD, arguments passed by 20-12
 registering with OAD
 oadutil, using 20-6
activation 2-3
 service activation 32-2
activation policies
 setting using CreationImplDef 20-9
Activator class
 deactivating an ORB object 32-2
 deferring object activation with 32-2, 32-3
ActiveObjectLifeCycleInterceptor 24-4
adding fields to user exceptions 5-7
additional information
 where to find 1-6
administration commands
 oadutil list 20-5
 oadutil unreg 20-12
 osfind 16-12
agent
 reporting 16-12
agentaddr file
 specifying IP addresses 16-9
 specifying Smart Agent IP addresses in 16-5
AliasDef object 21-7

Any 23-7
 class 22-10
Any class 19-2
APIs
 migrating to new calls 31-8
application development costs, reducing 2-1
Application Server Console
 starting 11-2
applications
 defining object interfaces 4-3
 thread pooling 8-4
 thread-per-session 8-7
applications, running 4-8
 client program, starting 4-9
 server object, starting 4-9
 Smart Agent, starting 4-8
arguments
 -export 32-3
ArrayDef object 21-7
at runtime 18-2
AttributeDef object 21-7
attributes, interface 15-7

B

backing store 18-14
backingStoreType 18-16
BAD_CONTEXT exception 5-1
BAD_INV_ORDER exception 5-1
BAD_OPERATION exception 5-1
 raised when operation is not found 23-7
BAD_PARAM exception 5-1
BAD_TYPECODE exception 5-1
bank example 4-2
bind
 generic object references 22-5
 nsutil 18-8
 process 10-2
bind_context
 nsutil 18-8
bind_new_context
 nsutil 18-8
binding
 multiple names 18-2
 names to objects 18-2
 ORB's tasks 16-12
binding to the EventChannel 19-8
binding, to objects
 actions performed by _bind() 10-2
 connection established by ORB 10-2
 proxy object created 10-2

- BindInterceptor 24-2
 - migrating 31-10
- BindInterceptorDelegate 31-10
- BindInterceptorManager 31-10
- BOA
 - binding 16-12
 - class moved 30-1
 - compiling BOA code 30-1
 - defining the servant 31-6
 - incoming requests 31-7
 - limitations in using 30-2
 - mapping to POA 31-7
 - migrating to POA 31-1
 - object activators 30-2
 - objects
 - naming 30-2
 - supported options 30-1
 - using with VisiBroker 4.0 30-1
- BOA initialization option
 - OAConnectionMax 4-12
 - OAConnectionMax 4-13
 - OAConnectionMaxIdle 4-12
 - OAid—TPool or TSession 4-13
 - OAipAddr 4-13
 - OAThreadMax 4-13
 - OAThreadMaxIdle 4-13
 - OAThreadMin 4-13
- BOA initialization
 - optionOAConnectionMaxIdle 4-13
- BOA::obj_is_ready() method 23-2
- BOA_init
 - change to package 30-1
- BOA_init options
 - OAConnectionMax 4-12, 4-13
 - OAThreadMax 4-13
- bootstrapping the naming service 18-7, 18-9
- bound objects
 - determining location and state 10-4
- boxed valuetypes 28-7
- broadcast address 16-7
- broadcast messages 16-1
- browser
 - Implementation Repository 12-7
 - Interface Repository 12-8, 12-10, 12-11
 - Location Service 12-1, 12-2
 - Naming Service 12-3, 12-4
 - Naming Service clusters 12-5
 - Naming Service federations 12-6
 - refreshing active object list 12-3
 - viewing Interface Repositories 12-9
- browsing
 - Interface Repository 12-11
- buffer size
 - setting 4-12
- building code 4-8

- make 4-8
- nmake 4-8

C

- callback object
 - triggers, using for 17-6
- calls
 - migrating to new API calls 31-8
- catching exceptions
 - modifying object to 5-6
 - system exceptions 5-4
 - user exceptions 5-6
- ChainClientInterceptorFactory 31-11
- ChainServerInterceptorFactory 31-11
- class
 - PullSupplierPOA 19-9
- classes
 - _tie
 - how it works 9-1
 - Any 22-10
 - ConstantDef 21-7
 - CORBA::DynamicImplementation 23-2
 - CreationImplDef 20-9
 - activation policy property 20-9
 - args property 20-9
 - env property 20-9
 - path_name property 20-9
 - DynamicImplementation 23-3
 - example of deriving from 23-3
 - ExceptionDef 21-7
 - migrating to new class names 31-8
 - NamedValue 22-9
 - NVList 23-7
 - ARG_IN parameter 23-7
 - ARG_INOUT parameter 23-7
 - ARG_OUT parameter 23-7
 - Repository 21-8
 - Request 22-6
 - ServerRequest 23-6
 - TriggerHandler 17-6
 - TypeCode 22-10
- CLASSPATH 3-2
- client
 - implementing 4-5
 - locating objects via URL 29-1
 - specifying a URL 29-1
 - using thread pooling 8-4
 - using thread-per-session 8-7
- client and server
 - running 4-8
- client application
 - DynAny example 27-5
- client stubs
 - generating 4-3
- ClientInterceptor 31-10

- ClientInterceptorDelegate 31-10
- ClientInterceptorFactory 31-10
- ClientRequestInterceptor 24-3
- client-side interceptors
 - migrating 31-10
- Closure objects
 - working with interceptors 24-16
- clusters 18-20
 - browsing 12-5
- code
 - compiling BOA code 30-1
- code generation 4-4
- COMM_FAILURE exception 5-1
- command_line
 - setting properties 14-3
- Common Object Request Broker *See* CORBA
- compilers
 - IDL, feature summary 2-4
 - make 4-8
 - nmake 4-8
- compiling
 - BOA code 30-1
 - the idl file 28-3
- completion status 5-2
 - system exceptions, obtaining for 5-2
- complex name 18-5
- configuring
 - Console 11-2
 - setting Console preferences 11-3
 - setting preferences 11-3
- configuring the naming service 18-6
- connect_pull_consumer 19-20
- connect_pull_supplier 19-7, 19-20
- connect_push_consumer 19-8, 19-20
- connect_push_supplier 19-7
- connecting
 - client applications with objects 2-1
 - point-to-point communications 16-8
 - Smart Agents on different local networks 16-5
- Connecting Suppliers to an EventChannel 19-7
- connection management 2-3
 - idle time 4-13
 - maximum number of connections 4-13
- connections
 - idle time 4-12
 - managing 4-12
 - managing, feature summary 2-3
 - point-to-point
 - IP addresses
 - agentaddr file, using 16-9
 - OSAGENT_ADDR, using 16-9
 - OSAGENT_ADDR_FILE, using 16-9
 - specifying at runtime 16-8
- Console 11-1, 12-1, 12-3, 12-8
 - browsers 12-1
 - changing property settings 13-8
 - configuring 11-2
 - Content pane 11-6
 - contents 11-1
 - elements 11-5
 - Gatekeeper 11-7
 - Implementation Repository 11-7
 - Interface Repositories (IREP) 11-7
 - invoking methods 13-6
 - Location Service 11-6
 - Naming Service 11-6
 - navigating 11-5
 - Navigation pane 11-6
 - ORB Services 11-6
 - preferences 11-3
 - Server Manager 11-7, 13-1
 - Server Manager contents 13-4
 - Server Manager security 13-4
 - setting preferences 11-3
 - setting properties 13-7
 - starting 11-2
 - starting from Application Server 11-2
 - starting from JBuilder 11-2
 - system information 11-4
- ConsoleImplementation Repositories 12-7
- ConstantDef object 21-7
- constructed
 - data types 27-3
- constructed data types 27-3
- consumer
 - adding to the EventChannel 19-8
- consumer proxy
 - getting a 19-7
- ConsumerAdmin
 - getting a 19-8
 - interface 19-18
 - methods
 - obtain_pull_supplier 19-18
 - obtain_push_supplier 19-18
- contacting
 - Borland 1-6
 - Inprise 1-6
 - Inprise Sales 1-6
 - technical support 1-6
- Contained object 21-8
 - defined_in() method 21-8
 - describe() method 21-8
 - name() method 21-8
- container
 - top level 13-2
- Container object 21-8
- Content pane
 - Console 11-6
- contents
 - highlights 1-4

- conventions
 - documentation 1-5
 - platform 1-5
 - typographic 1-5
- converting object references to string 10-3
- CORBA
 - Common Object Request Broker
 - Architecture 2-1
 - defined 2-1
 - description of 2-1
 - VisiBroker compliance 2-6
- CORBA exceptions A-1
- CORBA::DynamicImplementation class 23-2
- CORBA::ORB::create_operation_list() method 23-7
- corbaloc URL 18-10
- corbaname URL 18-10
- CosNaming
 - calling from the command line 18-7
- creating
 - a DII request 22-6
 - DynAnys 27-2
 - software components 2-1
- CreationImplDef class 20-9
 - activation_policy property 20-9
 - args property 20-9
 - env property 20-9
 - path_name property 20-9
- CreationImplDef struct
 - activating an object 20-10
- custom valuetypes 28-8

D

- data transfer to EventChannel 19-7
- data types
 - constructed 27-3
 - DynArray 27-4
 - DynEnum 27-3
 - DynSequence 27-4
 - DynStruct 27-4
 - DynUnion 27-4
 - traversing the components 27-3
- DATA_CONVERSION exception 5-1
- DataExpress (DX) adaptor 1-3
- DataExpress adaptor 18-14
- deactivate() method 32-1
- debugging interceptors, using 24-1
- def_kind() method 21-8
- default factories 28-6
- default naming context 18-12
- deferring
 - object activation 32-3
- deferring object activation
 - service activation
 - example of 32-3
 - ways to defer 32-1

- defined_in() method 21-8
- defining interface names 15-2
- deploying
 - applications 4-9
- deployment
 - description 4-9
- describe() method 21-8
- DescSeq all_instance_descs() method 17-4
- DescSeq all_replica_descs() method 17-5
- destroy
 - nsutil 18-8
- developer support 1-6
- developing
 - an example application 4-1
- development
 - defining object interfaces 4-3
- DII 2-4
 - creating a DII request 22-6
 - creating a request 22-6
 - feature summary 2-4
 - generic object reference 22-5
 - Interface Repository 21-1, 22-16
 - receiving multiple requests 22-15
 - receiving results 22-13
 - requests
 - asynchronous 22-14
 - sending a request 22-13
 - sending and receiving multiple requests 22-15
 - setting request arguments 22-8
 - using idl2java compiler 22-5, 23-2
- disabling
 - Smart Agent 16-3
- disconnect_push_consumer 19-21
- distributed applications
 - development process for distributed applications 4-1
- domains, running multiple 16-4
- Dynamic Invocation Interface *See* DII
- Dynamic Skeleton Interface 2-4, 23-3
 - activating objects 23-8
 - compiling object servers 23-2
 - deriving classes 23-3
 - DynamicImplementation class, deriving from 23-3
 - examples
 - invoke() method, implementing 23-3
 - location of 23-2
 - feature summary 2-4
 - implementation
 - derive from
 - CORBA::DynamicImplementation class 23-2
 - implement invoke() method 23-2
 - register with BOA using BOA::obj_is_ready() method 23-2

- responsible for 23-1
 - steps for creating 23-2
- input parameters 23-7
- inter-protocol bridging 23-1
- overview 23-1
- protocol bridging 23-1
- return values 23-7
- server object, implementing 23-6
- ServerRequest class 23-6
- DynamicImplementation class 23-3
 - example of deriving from 23-3
- DynAny
 - access and initializing 27-3
 - constructed data types 27-3
 - creating 27-2
 - current_component method 27-3
 - example application 27-4
 - example client application 27-5
 - example IDL 27-4
 - example server application 27-6
 - next method 27-3
 - overview 27-1
 - rewind method 27-3
 - seek method 27-3
 - to_any method 27-5
 - types 27-1
 - usage restrictions 27-2
- DynArray 27-2
 - data type 27-4
- DynEnum 27-2
 - data type 27-3
- DynFixed 27-2
- DynSequence 27-2
 - data type 27-4
- DynStruct 27-2
 - data type 27-4
- DynUnion 27-2
 - data type 27-4

E

- effective policies 10-6
- enabling rebinds
 - with the Smart Agent 16-10
- EnumDef object 21-7
- environment variables
 - CLASSPATH 3-2
 - for OAD 20-3
 - OSAGENT_ADDR 16-9
 - OSAGENT_ADDR_FILE 16-5
 - OSAGENT_LOCAL_FILE 16-7
 - OSAGENT_PORT
 - port number, choosing for Smart Agent 16-4
 - setting 3-3
 - overriding the Windows registry 3-2, 3-3
 - PATH, setting 3-1

- VBROKER_ADM
 - setting 3-2
 - setting path to agentaddr file 16-5
- equivalent implementations, checking for 10-4
- error log files 3-3
 - location of 3-4
- VBROKER_ADM, setting to log directory 3-4
- event channel 19-2
 - in-process implementation 19-15
 - using 19-6
- event service
 - communication models 19-4
 - overview 19-1
 - pull models 19-5
 - push models 19-5
 - setting queue length 19-14
 - starting 19-14
- EventChannel 19-5
 - interface 19-17
 - methods
 - destroy 19-17
 - for_consumers 19-17
 - for_suppliers 19-17
- EventChannelFactory
 - interface 19-17
 - methods
 - create 19-17, 19-18
- EventLibrary 19-15, 19-16
 - for Java 19-16
- example application
 - compiling the example 4-8
 - defining object interfaces 4-3
 - deploying the application 4-9
 - development process 4-1
 - generating client stubs 4-3
 - implementing the client 4-5
 - implementing the server 4-6
 - running the example 4-8
 - server servants 4-3
 - starting the server 4-8
 - with VisiBroker 4-1
 - writing the account interface in IDL 4-3
- example program, bank example 4-2
- examples
 - _tie class
 - location of code sample 9-2
 - activating objects 32-5
 - overriding activate() and deactivate() methods 32-2
 - service activator
 - instantiating 32-6
- activation
 - deferring with Activator service 32-3

- Dynamic Implementation class, deriving from 23-3
 - Dynamic Skeleton Interface
 - activating objects 23-8
 - Dynamic Implementation class, deriving from 23-3
 - invoke() method, implementing 23-3
 - location of sample code 23-2
 - exceptions
 - catching, modifying object to 5-6
 - fields, adding to user exceptions 5-7
 - narrowing to a system exception 5-4
 - printing an exception 5-3
 - throwing, modifying object to 5-6
 - user, defining 5-5
 - IDL
 - example specification 15-2
 - interface inheritance, specifying 15-8
 - oneway methods, defining 15-7
 - Interface Repository
 - interface, looking up 21-9
 - Location Service
 - finding all instances of an interface 17-7
 - finding all known by Smart Agents 17-8
 - trigger handler
 - implementing 17-10
 - OSAGENT_PORT environment variable, setting 16-4
 - push consumer 19-8
 - push supplier 19-8
 - quick start
 - building example 4-8
 - make 4-8
 - makefile sample 4-8
 - nmake 4-8
 - client
 - balance, obtaining 4-6
 - binding to Account object 4-5
 - implementing 4-5
 - compiling, files produced 4-4
 - IDL, writing account interface in 4-3
 - running example 4-8
 - Account server, starting 4-9
 - client program, starting 4-9
 - Smart Agent, starting 4-8
 - server
 - implementing Account 4-6
 - server 13-2
 - Smart Agent localaddr file 16-7
 - examplesactivating objects 32-5
 - ExceptionDef object 21-7
 - exceptions
 - adding fields to user exceptions 5-7
 - BAD_OPERATION 23-7
 - catching user exceptions 5-6
 - completion status for exceptions 5-2
 - CORBA, overview 5-1
 - CORBA-defined system exceptions 5-1
 - handling 5-3
 - narrowing to system exceptions 5-3
 - system
 - BAD_CONTEXT 5-1
 - BAD_INV_ORDER 5-1
 - BAD_OPERATION 5-1
 - BAD_PARAM 5-1
 - BAD_TYPECODE 5-1
 - COMM_FAILURE 5-1
 - completion status, obtaining 5-2
 - CompletionStatus values 5-2
 - DATA_CONVERSION 5-1
 - FREE_MEM 5-2
 - handling 5-3
 - IMP_LIMIT 5-2
 - INITIALIZE 5-2
 - INTERNAL 5-2
 - INTF_REPOS 5-2
 - INV_FLAG 5-2
 - INV_INDENT 5-2
 - INV_OBJREF 5-2
 - MARSHAL 5-2
 - narrowing exceptions to 5-3
 - NO_IMPLEMENT 5-2
 - NO_MEMORY 5-2
 - NO_PERMISSION 5-2
 - NO_RESOURCES 5-2
 - NO_RESPONSE 5-2
 - OBJ_ADAPTOR 5-2
 - OBJECT_NOT_EXIST 5-2
 - PERSIST_STORE 5-2
 - SystemException class 5-1
 - TRANSIENT 5-2
 - types, catching specific 5-4
 - UNKNOWN 5-2
 - UserException class 5-5
 - throwing 5-6
 - user
 - catching exceptions, modifying object to 5-6
 - defining 5-5
 - fields, adding to 5-7
 - throwing exceptions, modifying object to 5-6
 - exclusive connection 1-1
 - export argument 32-3
- ## F
-
- factories
 - default 28-6
 - valuetypes 28-5
 - factory
 - migration 31-11

- factory_name 18-8
 - setting 18-7
- failover 18-23
- fault tolerance 2-3, 18-24
 - object implementation 16-10
 - providing for objects 16-10
 - replicating objects registered with the OAD 16-10
- features of VisiBroker 2-3
 - activating objects and implementations 2-3
 - compilers, IDL 2-4
 - connection management 2-3
 - dynamic invocation 2-4
 - IDL compilers 2-4
 - IDL interface to Smart Agent 2-3
 - implementation activation 2-3
 - implementation repository 2-4
 - interface repository 2-4
 - Location Service 2-3
 - multithreading 2-3
 - object activation 2-3
 - object database integration 2-5
 - Smart Agent architecture 2-3
 - thread management 2-3
- federations
 - browsing 12-6
- file extensions 4-4
- files
 - agentaddr 16-5
 - compiling, produced by 4-4
 - impl_rep 20-2
 - localaddr 16-7
- for_consumers 19-8
 - method 19-17
- for_suppliers 19-7
 - method 19-17
- FREE_MEM exception 5-2

G

- Gatekeeper
 - Console 11-7
- generating
 - client stubs and server servants 4-3
- _get_policy 10-6
- globally scoped objects
 - Smart Agent, registration with 16-1
- Glossary G-1

H

- handling system exceptions 5-3
- hash value, obtaining for an object reference 10-4
- HostnameSeq all_agent_locations() method 17-4
- HTML
 - setting properties 14-3

I

- id field
 - NameComponent 18-4
- IDL
 - compiler 4-3
 - feature summary 2-4
 - generating stubs and skeletons with 4-3
 - constructs, represented in Interface Repository 21-2
 - DynAny example 27-4
 - example specification 15-2
 - idl2cpp compiler
 - how it generates code 15-2
 - interface inheritance, specifying 15-8
 - methods for attributes, generated by 15-7
 - oneway methods, defining 15-7
 - interface inheritance, specifying 15-8
 - Interface Repository, information contained in 21-1
 - mapping to Java 2-9, 4-3
 - OAD interface 20-14
 - oneway methods, defining 15-7
 - specifying objects in 4-3
- idl2cpp compiler 4-3
 - attribute methods 15-7
 - export 32-3
 - how it generates code 15-2
 - interface inheritance, specifying 15-8
 - methods for attributes, generated by 15-7
 - oneway methods, defining 15-7
- idl2ir compiler 21-5
 - command info 2-6
 - description 2-6
- idl2java compiler
 - generating stub code using DII 22-5, 23-2
 - portable flag 22-5, 23-2
- IMP_LIMIT exception 5-2
- impl_is_down() method 17-6
- impl_is_ready() method
 - Location Service, with 17-6
- impl_rep file for Implementation Repository
 - data 20-2
- implementation
 - activating
 - changing characteristics dynamically 20-10
 - activation 2-3
 - deferred
 - Service Activator 32-5
 - activation policies
 - setting using CreationImplDef 20-9
 - connections with Smart Agents 16-1
 - deferring
 - example of 32-3
 - services 32-3

- ways to defer 32-1
- dynamic creation with DSI, steps for 23-2
 - implement invoke() method 23-2
 - register with BOA using `boa.obj_is_ready()` method 23-2
- equivalent, checking for 10-4
- fault tolerance, providing 16-10
- migrating
 - between hosts 16-11
 - instantiated objects 16-11
 - OAD, registered with 16-11
 - objects with state 16-11
- multiple instances, distinguishing between 20-9
- OAD, arguments passed by 20-12
- registering
 - multiple instances, defining 20-9
- replicating 16-10
- state, invoking methods on 16-10
- stateless, invoking methods on 16-10
- support 2-3
- unregistering with the OAD 20-12
 - oadutil, using 20-12
- Implementation Repository 2-4
 - accessing 12-7
 - browser 12-7
 - Console 11-7
 - contents of, displaying 20-14
 - feature summary 2-4
 - for OAD 20-5
 - impl_rep file 20-2
 - registration information stored in 20-2
 - removed when unregistered with the OAD 20-12
 - specifying directory with OAD 20-3
 - unregistering objects 20-12
 - using OAD 20-3
- implementations
 - binding 16-12
 - inheritance
 - allowing 9-1
 - listing 20-5
 - registered with OAD 20-5
 - reporting 16-12
 - thread pooling 8-3
 - unregistering with OAD 20-12, 20-13
 - using thread-per-session 8-7
- implementing
 - a list of NamedValue objects 22-8
 - push suppliers 19-8
 - the client 4-5
 - the server 4-6
- implementing factories 28-5
- implementing the Factory class 28-4
- implementing valuetypes 28-3
- import statements 18-24
- importBiDir 11
- information, where to find 1-6
- inheritance
 - from implementations, allowing 9-1
 - implementations
 - generated skeleton, not from 23-1
 - interface 15-8
- inheritance of interfaces
 - specifying 15-8
- inheriting
 - valuetype base classes 28-4
- INITIALIZE exception 5-2
- initializing
 - value of a DynAny 27-3
- Initializing the naming service (Java only) 18-12
- In-memory adaptor 18-14
- Inprise
 - contacting 1-6
- in-process event channel 19-15, 19-16
- input parameters, processing in DSI 23-7
- input/output arguments
 - for method invocation requests 22-9
- installation support 1-6
- installing the naming service 18-6
- instances
 - determining for object reference 10-4
 - distinguishing between 20-9
 - finding with `all_instances` method 17-4
 - finding with Location Service 17-1
 - like-named, finding using Location Service 17-5
- interceptor
 - default interceptor classes 24-6
- Interceptor interface
 - example 24-7
 - interfaces 24-2
 - managers 24-2
 - message, supported 24-1
 - overview 24-1
 - registering with the ORB 24-6
 - request, supported 24-1
- interceptors
 - ActiveObjectLifecycleInterceptor 24-4
 - BindInterceptor 24-2
 - client 24-2
 - ClientRequestInterceptor 24-3
 - creating interceptor objects 24-7
 - customizing the ORB 2-5
 - example program 24-7
 - interfaces 24-2
 - IORCreationInterceptor 24-5
 - loading 24-7
 - managers 24-2
 - passing data between 24-16

- POALifeCycleInterceptor 24-4
- registering interceptors with the ORB 24-6
- server 24-4
- ServerRequestInterceptor 24-4
- *_interface_name() method 10-4
- interface
 - attributes 15-7
 - IDL, defining in 4-3
 - inheritance 15-8
 - looking up 21-9
 - TriggerHandler 17-6
- Interface Definition Language *See* IDL
- interface name
 - obtaining 10-4
 - unregistering objects with OAD 20-12
- Interface Repository 2-4, 12-8
 - _get_interface() method 21-2
 - accessing 12-10
 - accessing object information 21-8
 - browser 12-8, 12-10, 12-11
 - browsing 12-11
 - console 11-7
 - contents of 21-2, 21-7
 - def_kind 21-6
 - definition types 12-9
 - description 21-1
 - examples 21-9
 - feature summary 2-4
 - how many? 21-2
 - id, identifying an IRObjct with 21-6
 - identifying objects within 21-6
 - def_kind 21-6
 - id 21-6
 - name 21-6
 - inherited interfaces 21-8
 - Contained 21-8
 - defined_in() method 21-8
 - describe() method 21-8
 - name() method 21-8
 - Container 21-8
 - IRObjct 21-8
 - def_kind() method 21-8
 - irep 12-8
 - name, specifying for IR objects 21-6
 - populating with idl2ir 2-6
 - starting 12-8
 - structure 21-6
 - structure of 21-5
 - types of objects stored in 21-7
 - AliasDef 21-7
 - ArrayDef 21-7
 - AttributeDef 21-7
 - ConstantDef 21-7
 - EnumDef 21-7
 - ExceptionDef 21-7
 - InterfaceDef 21-7
 - ModuleDef 21-7
 - OperationDef 21-7
 - PrimitiveDef 21-7
 - Repository 21-7
 - SequenceDef 21-7
 - StringDef 21-7
 - StructDef 21-7
 - UnionDef 21-7
 - updating contents with idl2ir 21-5
 - viewing 12-9
 - viewing contents of 21-4
 - what is? 21-1
- InterfaceDef object 21-7
 - in Interface Repository 21-2
- interfaces
 - abstract 28-7
 - accessible, finding all 17-4
 - descriptions of in Interface Repository 21-1
 - inheritance, specifying 15-8
 - Quality of Service 10-6
 - reporting 16-12
 - using java2iio 26-6
- INTERNAL exception 5-2
- interoperability 2-8
 - with other ORB products 2-9
 - with VisiBroker for C++ 2-8
- INTF_REPOS exception 5-2
- INV_FLAG exception 5-2
- INV_INDENT exception 5-2
- INV_OBJREF exception 5-2
- invocation feature summary 2-4
- invoke() method 23-1, 23-2
 - example of implementing 23-3
- IORCreationInterceptor 24-5
- IP address 4-12
- IP subnet mask
 - broadcast messages, specifying scope of 16-5
 - localaddr file, contained within 16-7
- IR *See* Interface Repository 2-4
- IR *See* Interface Repository
- IREP
 - Console 11-7
- irep tool
 - creating an interface repository with this tool 21-3
 - creating Interface Repository with 21-3
 - viewing Interface Repository with 21-4
- IRObjct object 21-8
 - def_kind() method 21-8
 - _is_a() method 10-4
 - _is_bound() method 10-4
 - _is_local() method 10-4
 - _is_remote() method 10-4

J

Java

- Java Development Kit (JDK) 2-7
- runtime environment 2-7
- java2iio
 - mapping complex data types 26-6
 - mapping primitive types 26-5
- JDataStore JDBC driver 1-3
- JDBC adaptor 1-3, 18-14
- JDBC Adaptor properties 18-16
- jdbcDriver 18-16

K

kind field

- NameComponent 18-4

L

limitations

- in using BOA 30-2

list

- nsutil 18-8

listing, contents of implementation

- repository 20-14

load balancing

- Location Service, used for 17-3
- migrating objects between hosts 16-11

localaddr file, specifying interface usage 16-7

Location Service 12-1

- accessing 12-2

Agent, accessible through 17-3

- DescSeq all_instance_descs() method 17-4
- DescSeq all_replica_descs() method 17-5
- HostnameSeq all_agent_locations()
 - method 17-4
- impl_is_down() method 17-6
- impl_is_ready() method 17-6
- ObjSeq all_instances() method 17-4
- ObjSeq all_replica() method 17-5
- reg_trigger() method 17-5
- RepositoryIDSeq all_repository_ids()
 - method 17-4

browser 12-1, 12-2

components of Location Service Agent 17-3

Console 11-6

enhanced object discovery 2-3

examples

- finding all instances of an interface 17-7
- finding all known by Smart Agents 17-8
- trigger handler
 - implementing 17-10

feature summary 2-3

filtering 12-1

instances, finding 17-4

like-named 17-5

refreshing 12-1

refreshing active object list 12-3

repository ID, used to identify interfaces 17-4

Smart Agents

- cooperation with 17-1

- finding hosts running 17-4

starting 12-1

trigger

- creating 17-6

- first instance only, looking at 17-6

- what is? 17-5

TriggerHandler 17-6

- what is a location service? 17-1

location, determining for an object reference 10-4

log files 3-3

- location of 3-4

- VBROKER_ADM, setting to log directory 3-4

logging output 3-3

loginPwd 18-16

lookup() method 21-8

M

make, compiling with 4-8

makefile, sample for Solaris 4-8

managing

Implementation Repositories

- starting 12-7

Interface Repository 12-8

Location Service 12-1

Naming Service 12-3

manual

- conventions 1-5

mapping

- IDL to Java 2-9

MARSHAL exception 5-2

marshalling 28-8

- using java2iio 26-5

maxQueueLength 19-14

message interceptors, supported 24-1

method 10-6

- for_consumers 19-17

methods

- *_interface_name() 10-4

- *_object_name() 10-4

- *_repository_id() 10-4

- *object_to_string() 10-3

- _is_a() 10-4

- _is_bound() 10-4

- _is_local() 10-4

- _is_remote() 10-4

- _name()

- discovering name of IR object with 21-8

activate() 32-1

boa.obj_is_ready() 23-2

- CORBA::ORB::create_operation_list() 23-7
- deactivate() 32-1
- def_kind() 21-8
- defined_in() 21-8
- describe() 21-8
- DescSeq all_instance_descs() 17-4
- DescSeq all_replica_descs() 17-5
- for_suppliers 19-17
- _get_policy 10-6
- HostnameSeq all_agent_locations() 17-4
- impl_is_down() 17-6
- impl_is_ready() 17-6
- invoke() 23-1, 23-2
 - example of implementing 23-3
- invoking in the Console 13-6
- lookup() 21-8
- ObjSeq all_instances() 17-4
- ObjSeq all_replica() 17-5
- oneway, defining 15-7
- open() 23-7
- reg_trigger() 17-5
- RepositoryIDSeq all_repository_ids() 17-4
- _set_policy_override method 10-6
- state
 - objects with, invoking on 16-10
 - stateless objects, invoking on 16-10
- string_to_object() 10-3
- unreg_trigger() 17-5
- migrating
 - activating the POA manager 31-6
 - BOA to POA 31-1
 - code 31-1
 - defining the servant 31-6
 - incoming requests 31-7
 - instantiated objects 16-11
 - mapping BOA types to POA 31-7
 - objects 16-11
 - objects between hosts 16-11
 - objects registered with OAD 16-11
 - objects with state 16-11
 - setting POA policies 31-5
 - to new API calls 31-8
 - to new class names 31-8
 - to new package names 31-7
- Migrating BindInterceptor 31-10
- ModuleDef object 21-7
 - in Interface Repository 21-2
- multihomed hosts
 - described 16-6
 - interface usage, specifying 16-7
- multithreading
 - feature summary 2-3

- binding names to objects 18-1
- _name() method 21-8
- name
 - complex 18-5
 - defined 18-4
 - object
 - qualifying binding with 10-2
 - resolution 18-4
 - simple 18-5
 - stringified 18-5
- Name Resolution 18-5
- NameComponent
 - defined 18-4
 - id field 18-4
 - kind field 18-4
- NamedValue
 - class 22-9
 - objects 22-8
 - pair 22-9
- nameserve 18-7
- namespace 18-1
- NameValuePair 27-5
- Naming Contexts
 - class 18-11
 - default 18-12
 - defined 18-3
 - root 18-4
 - use by client applications 18-3
 - use by object implementations 18-3
- Naming Service 12-3
 - accessing 12-4
 - bootstrapping 18-7, 18-9
 - browser 12-3, 12-4
 - browsing 12-5
 - clusters 12-5, 18-20
 - configuring 18-6
 - Console 11-6
 - failover 18-23
 - fault tolerance 18-24
 - federations 12-6
 - hierarchical namespace 18-2
 - installing 18-6
 - overview 18-1
 - pluggable backing store 18-14
 - configuration 18-15
 - properties file 18-15
 - types 18-14
 - properties 18-13
 - sample programs 18-25
 - starting 12-3, 18-6
- Naming Service Utility 18-7
- Naming Services Manager
 - introduction 12-3, 12-8
- NamingContext
 - bootstrapping 18-4

N

Name

- factories 18-4
- NamingContextExt 18-12
- narrowing
 - exceptions to system exception 5-3
 - object reference 10-5
- navigating
 - the Console 11-5
- Navigation pane
 - Console 11-6
- network
 - reporting objects and services 16-12
- new_context
 - nsutil 18-8
- nmake, compiling with 4-8
- NO_IMPLEMENT exception 5-2
- NO_MEMORY exception 5-2
- NO_PERMISSION exception 5-2
- NO_RESOURCES exception 5-2
- NO_RESPONSE exception 5-2
- nsutil 18-7
- NT services
 - console mode 16-3
 - osagent 16-3
- null semantics 28-7
- NVList class 23-7
 - ARG_IN parameter 23-7
 - ARG_INOUT parameter 23-7
 - ARG_OUT parameter 23-7
 - implementing a list of arguments with 22-8

O

- OAD
 - arguments passed by 20-12
 - IDL interface to 20-14
 - impl_rep file 20-2
 - implementation repository 20-2
 - listing objects 20-5
 - migrating objects registered with 16-11
 - programming interface 20-14
 - registering objects 20-10
 - registration information stored in
 - Implementation Repository 20-2
 - replicating objects registered with 16-10
 - setting the activation policy 20-11
 - specifying time-out 20-3
 - starting 20-2 to 20-3
 - storing registration info 20-5
 - unregistering objects 20-12
 - oadutil, using 20-12
 - removing from Implementation Repository 20-12
 - removing from Smart Agent 20-12
- OAD command
 - setting environment variables for 20-3
- oadj
 - reporting 16-12
- oadutil
 - listing objects registered with OAD 20-5
 - unregistering implementations 20-12
- oadutil tool
 - displaying contents of Implementation Repository 20-14
 - registering object implementations with 20-1
- OBJ_ADAPTOR exception 5-2
- *object_to_string() method 10-3
- object
 - accessible, finding all 17-4
 - accessing information from Interface Repository 21-8
 - activating
 - changing characteristics dynamically 20-10
 - deferring
 - services 32-3
 - OAD, arguments passed by 20-12
 - activation
 - deferred service Activator 32-5
 - activation policies
 - setting using CreationImplDef 20-9
 - active connection 10-5
 - connecting to with the OAD 16-2
 - connections with Smart Agents 16-1
 - deferring
 - ways to defer 32-1
 - dynamic creation with DSI, steps for 23-2
 - implement invoke() method 23-2
 - register with BOA using boa.obj_is_ready() method 23-2
 - Dynamic Skeleton Interface
 - server object, implementing 23-6
 - exceptions
 - catching, modifying to 5-6
 - throwing, modifying to 5-6
 - fault tolerance, providing 16-10
 - finding with Location Service 17-1
 - getting object name 10-4
 - IDL, specifying in 4-3
 - implementing a specific interface 10-4
 - implementing on remote host 10-5
 - in local address space 10-5
 - inheritance
 - from implementations 9-1
 - generated skeletons, not from 23-1
 - instances
 - finding using Location Service 17-4
 - like-named, finding using Location Service 17-5
 - listing 20-5
 - migrating
 - between hosts 16-11
 - instantiated objects 16-11

- OAD, registered with 16-11
- objects with state 16-11
- multiple instances, distinguishing
 - between 20-9
- referring to same interface implementation 10-4
- registering
 - multiple instances, defining 20-9
 - with OAD 20-10
 - with Smart Agent, automatic 16-3
- replicating 16-10
- reporting objects on a network 16-12
- setting the activation policy 20-11
- setting the path 20-11
- state, invoking methods on 16-10
- stateless, invoking methods on 16-10
- unregistering
 - from OAD 20-12
 - with OAD 20-12
- unregistering with the OAD
 - oadutil, using 20-12
 - using CreationImplDef struct 20-10
- object activation 2-3
 - deferred 32-6
 - example of deferred method 32-3
 - service activation 32-2
 - support 2-3
- Object Activation Daemon *See* OAD
- Object Database Activator
 - feature summary 2-5
- object discovery
 - enhanced with the Location Service 2-3
- object implementation
 - changing dynamically 20-10
 - fault tolerance 16-10
 - implementations that maintain state 16-10
- Object Management Group 2-1
- object migration 16-11
- object names
 - obtaining 10-4
 - qualifying binding with 10-2
- object reference
 - converting to string 10-3
 - converting type 10-5
 - determining the locations and state 10-4
 - equivalent implementations, checking for 10-4
 - hash value, obtaining 10-4
 - instance of type, determining 10-4
 - instances, finding 17-4
 - instances, finding like-named 17-5
 - interface name, obtaining 10-4
 - location, determining 10-4
 - narrowing 10-5
 - object name, obtaining 10-4
 - obtaining object and interface names 10-4
 - operations on 10-3
 - repository id, obtaining 10-4
 - state, determining 10-4
 - string, converting to 10-3
 - sub-type, determining if is 10-4
 - super-type, converting to 10-5
 - type, determining 10-4
 - type, using the `_is_a()` method 10-4
 - widening 10-5
- object references
 - persistent 30-2
- Object Request Broker *See* ORB
- object wrappers
 - adding un-typed 25-6
 - co-located client and server 25-11
 - customizing the ORB 2-5
 - deriving a typed wrapper 25-11
 - described 25-1
 - example programs 25-2
 - installing un-typed 25-5
 - post_method 25-4
 - pre_method 25-4
 - removing typed wrappers 25-14
 - removing un-typed factories 25-8
 - typed 25-2, 25-8
 - order of invocation 25-10
 - un-typed 25-2
 - implementing 25-4
 - using 25-4
 - using both typed and un-typed wrappers 25-14
- OBJECT_NOT_EXIST exception 5-2
- object-oriented approach
 - software component creation 2-1
- objects
 - associating a URL 29-1
 - binding 16-12
 - executable's path 20-11
 - locating via URL 29-1
- ObjectWrapper 25-11
- ObjSeq `all_instances()` method 17-4
- ObjSeq `all_replica()` method 17-5
- obtain_pull_consumer 19-7
- obtain_push_consumer
 - method 19-7
- obtain_push_supplier 19-8
- obtaining
 - object and interface names 10-4
- OMG 2-1
 - Common Object Services Specification 19-3
 - Event Service 19-1
 - Notification Service 19-1
- oneway methods, defining 15-7
- open() method 23-7
- OpenFusion Notification Service 19-1
- OperationDef object 21-7
 - in Interface Repository 21-2

- operator
 - scope resolution (::) 23-5
- options
 - BOA options 30-1
- OptJDBC adaptor 1-3
- ORB
 - creating proxy 16-12
 - customizing with interceptors and object wrappers 2-5
 - definition 16-12
 - domains 16-4
 - function of 2-1
 - interoperability 2-8
 - object implementations 20-5
 - resolve_initial_references 18-9
- ORB browsers
 - Console 12-1
- ORB Console services 11-6
- ORB initialization option
 - ORBAgentaddr 4-12
 - ORBAgentport 4-12
 - ORBmbufsize 4-12
 - ORBTcpNoDelay 4-12
- ORBDefaultInitRef property 18-10
- ORBInitRef 18-7
- ORBInitRef property 18-9
- OSAgent
 - checking client existing (heartbeat) 16-3
 - detecting other Agents 16-6
 - disabling 16-3
 - ensuring availability 16-3
 - locating objects 16-2
 - Smart Agent 16-1
 - starting 16-2
 - verbose output 16-3
- osagent
 - binding 16-12
 - object name 30-2
 - reporting 16-12
 - starting Smart Agents with 4-8
- OSAgent (Smart Agent)
 - VisiBroker architecture 2-3
- OSAGENT_ADDR environment variable 16-9
- OSAGENT_ADDR_FILE environment variable 16-5
- OSAGENT_LOCAL_FILE environment variable 16-7
- OSAGENT_PORT environment variable 16-4
 - setting 3-3
- osfind
 - command info 16-12
- output, logging 3-3
- overrides
 - policy 10-6
- overview

- event service 19-1
- Naming Service 18-1

P

- package names
 - migrating to new packages 31-7
- parameters, passing
 - input, processing in DSI 23-7
- PATH, setting 3-1
- PERSIST_STORE exception 5-2
- persistent objects
 - ODA, feature summary 2-5
- ping
 - nsutil 18-8
- platform conventions 1-5
- platform designation with icons 1-5
- pluggable backing store 18-14
 - configuration 18-15
 - properties file 18-15
 - types 18-14
- POA
 - activating objects 7-12
 - activating the manager 31-6
 - creating 7-3, 7-6
 - defined 7-1
 - defining the servant 31-6
 - dispatching properties 7-20
 - incoming requests 31-7
 - listener port property 7-22
 - listening properties 7-20
 - managing POAs 7-17
 - mapping from BOA types 31-7
 - migrating from BOA 31-1
 - obtaining a root reference 31-5
 - POA manager 7-17
 - policies 7-3
 - processing requests 7-25
 - servant managers 7-12
 - ServantLocators 7-15
 - servants
 - using 7-12
 - setting policies 31-5
- POALifeCycleInterceptor 24-4
- point-to-point communication 16-8
 - IP addresses
 - agentaddr file, using 16-9
 - OSAGENT_ADDR, using 16-9
 - OSAGENT_ADDR_FILE, using 16-9
 - specifying at runtime 16-8
- policies 10-6
 - effective 10-6
 - mapping BOA to POA 31-7
 - POA 7-3
 - setting for POA 31-5
- policy overrides 10-6

- poolSize 18-16
- port number 4-12
 - listener 7-22
 - specifying for Smart Agent 16-4
- portability
 - server-side 2-5
- Portable object adaptor
 - policies 7-3
- preferences
 - setting for Console 11-3
 - setting in the Console 11-3
- PrimitiveDef object 21-7
- process
 - quick start example
 - building example 4-8
 - makefile sample 4-8
 - client
 - balance, obtaining 4-6
 - binding to Account object 4-5
 - implementing 4-5
 - compiling, files produced 4-4
 - IDL, writing account interface in 4-3
 - running example 4-8
 - Account server, starting 4-9
 - client program, starting 4-9
 - Smart Agent, starting 4-8
 - server
 - implementing Account 4-6
- properties 13-10
 - data types 14-4
 - DataExpress adaptor 18-18
 - grouping properties example 14-4
 - Java 14-6
 - JDBC adaptor 18-16
 - JNDI adaptor 18-18
 - Naming Service 18-13, 18-15
 - ORBDefaultInitRef 18-10
 - ORBInitRef 18-9
 - order of precedence 14-1
 - properties file 14-4
 - property file 14-5
 - reload from server 13-11
 - save settings to file 13-10
 - setting 14-1
 - setting boolean values 14-5
 - setting in the Console 13-7
 - setting null values 14-5
 - setting through command-line 14-3
 - setting through HTML 14-3
 - specify for the ORB 14-1
 - specifying a property file 14-5
 - storage file 13-8
 - SVCnameroot 18-9
 - using 14-5
 - using ORB.init 14-1
 - VisiBroker 14-1
- property settings
 - changing 13-8
- property types
 - used in the Server Manager 13-7
- proxy consumer 19-2
- proxy object
 - binding process, created during 10-2
- proxy objects
 - binding 16-12
- proxy supplier 19-2
- ProxyPullConsumer 19-5
 - interface 19-19
- ProxyPullSupplier 19-5
 - interface 19-20
 - methods
 - connect_pull_consumer 19-20
 - connect_push_consumer 19-20
- ProxyPushConsumer 19-5
 - interface 19-20
 - methods
 - connect_pull_supplier 19-20
- ProxyPushSupplier 19-5
- pull 19-7, 19-8
- pull model 19-5
- PullConsume 19-12
- PullConsume.java 19-9
- PullConsumer
 - interface 19-21
 - methods
 - disconnect_push_consumer 19-21
- PullModel 19-8
- PullSupplier
 - interface 19-22, 19-23
 - methods
 - disconnect_pull_supplier 19-22
 - disconnect_push_supplier 19-23
 - pull 19-22
 - try_pull 19-22
- PullSupplierPOA
 - class 19-9
- PullSupply 19-9
- PullSupply.java 19-9
- push 19-7
- push model 19-5
- push supplier
 - example 19-8
- PushConsumer
 - example 19-8
 - interface 19-21
- PushModel 19-8
- PushSupplier
 - implementing 19-8

Q

- QoS 10-6
- Quality of Service 10-6

- interfaces 10-6
- queue length
- setting 19-14

R

- rebind
 - nsutil 18-8
- rebind_context
 - nsutil 18-8
- receiving multiple requests 22-15
- reducing application development costs 2-1
- ref_data parameter 20-9
- reference data 20-9
- reg_trigger() 17-5
 - method 17-5
- registering
 - valuetypes 28-6
 - your Factory with the ORB 28-5
- registering objects 20-9
 - oadutil, using 20-6
 - unregistering
 - with OAD 20-12
 - unregistering with the OAD
 - oadutil, using 20-12
- registration
 - OAD, information stored in Implementation Repository 20-2
 - Smart Agents, with 16-1
- RelativeConnectionTimeoutPolicy 10-8
- replicating objects registered with the OAD 16-10
- *_repository_id() method 10-4
- Repository class 21-8
- repository id
 - obtaining 10-4, 20-4
- Repository object 21-7
- RepositoryIDSeq all_repository_ids() method 17-4
- Request class 22-6
- request interceptors, supported 24-1
- requests
 - waiting for incoming requests with BOA 31-7
 - waiting for incoming requests with POA 31-7
- REQUIRE_AND_TRUST 11
- resolve
 - nsutil 18-8
- Resolver interface
 - associating a URL with an object 29-2
- restrictions
 - usage of DynAnys 27-2
- root
 - POA 31-5
- root NamingContext 18-4
- running applications 4-8
 - client program, starting 4-9
 - server object, starting 4-9
 - Smart Agent, starting 4-8

S

- sample programs
 - Naming Service 18-25
- scope resolution operator (::) 23-5
- sending
 - a DII request 22-13
 - multiple requests 22-15
- SequenceDef object 21-7
- servant
 - defining 31-6
- server
 - enabling 13-5
 - implementing 4-6
 - reload properties 13-11
 - restart 13-10
 - update 13-10
- server application
 - DynAny example 27-6
- Server Manager
 - changing property settings 13-8
 - Console 11-7, 13-1
 - enabling a server 13-5
 - example server 13-2
 - GUI 13-2
 - invoking methods 13-6
 - property types 13-7
 - reload properties 13-11
 - restart server 13-10
 - setting properties 13-7
 - setting security 13-4
 - specifying property storage 13-8
 - top level container 13-2
 - update server 13-10
 - update server and save properties to file 13-10
 - viewing contents 13-4
- server servants
 - generating 4-3
- ServerInterceptor 31-10
- ServerInterceptorDelegate 31-10
- ServerInterceptorFactory 31-10
- ServerRequest class 23-6
- ServerRequestInterceptor 24-4
- servers
 - example of tie mechanism 9-2
 - setting the activation policy 20-11
 - setting the path 20-11
 - threading considerations 8-10
- server-side
 - portability 2-5
- server-side interceptors
 - migrating 31-10
- service activation
 - deferred, implementing 32-3
 - example of 32-3

- deferring object activation 32-3
- example of 32-3
- implementing a service Activator 32-5
- instantiating service activator 32-6
- service activator
 - implementing 32-5
 - instantiating 32-6
- ServiceInit class 24-8
- ServiceLoader interface 24-7, 24-8
- services
 - in the Console 11-6
 - reporting services on a network 16-12
- _set_policy_override method 10-6
- setting
 - Console preferences 11-3
 - properties 13-7
- sharing semantics 28-7
- shutdown
 - nsutil 18-8
- simple name 18-5
- skeletons 4-3
- Smart Agent
 - agentaddr file
 - specifying IP addresses in 16-5
 - availability, ensuring 16-3
 - binding 16-12
 - checking client existing (heartbeat) 16-3
 - communication 16-1
 - connecting networks, different local 16-5
 - connecting to objects with the OAD 16-2
 - cooperation with other agents 16-2
 - detecting other Agents 16-6
 - disabling 16-3
 - domains, running under multiple 16-4
 - ensuring availability 16-3
 - fault tolerance, providing for objects 16-10
 - feature summary 2-3
 - flat namespace 18-2
 - hosts, finding all running Smart Agents 17-4
 - IP addresses
 - agentaddr file, using 16-9
 - OSAGENT_ADDR, using 16-9
 - OSAGENT_ADDR_FILE, using 16-9
 - specifying at runtime 16-8
 - localaddr file, specifying interface usage 16-7
 - locating 16-1
 - Location Service, cooperation with 17-1
 - multihomed hosts
 - interface usage, specifying 16-7
 - using 16-6
 - OAD, cooperating with 16-2
 - object name 30-2
 - objects removed from when unregistered with
 - OAD 20-12
 - OSAgent 16-1
 - OSAGENT_ADDR environment variable 16-9
 - OSAGENT_ADDR_FILE environment variable 16-5
 - OSAGENT_LOCAL_FILE file 16-7
 - OSAGENT_PORT environment variable 16-4
 - point-to-point
 - communication 16-8
 - IP addresses
 - agentaddr file, using 16-9
 - OSAGENT_ADDR, using 16-9
 - OSAGENT_ADDR_FILE, using 16-9
 - specifying at runtime 16-8
 - port numbers, specifying 16-4
 - reregistration of objects automatically 16-3
 - starting 16-2
 - starting multiple instances 16-2
 - VBROKER_ADM environment variable 16-5
 - verbose output 16-3
 - versus the Naming Service 18-2
 - what is? 16-1
 - Smart Agent (OSAgent)
 - architecture 2-3
 - sockets
 - batching requests 4-12
 - specifying
 - IP addresses 16-9
 - starting
 - Application Server Console 11-2
 - Console 11-2
 - OAD 20-2 to 20-3
 - starting the naming service 18-6
 - state
 - determining for an object reference 10-4
 - state, objects with, invoking methods on 16-10
 - stateless objects, invoking methods on 16-10
 - status, completion
 - system exceptions, obtaining for 5-2
 - string
 - converting to object references 10-3
 - string_to_object() method 10-3
 - StringDef object 21-7
 - stringification
 - using object_to_string() method 10-3
 - stringified names 18-5
 - StructDef object 21-7
 - subnet mask 16-5, 16-7
 - sub-type, determining 10-4
 - supplier
 - adding to the EventChannel. 19-7
 - supplier proxy
 - getting a 19-8
 - SupplierAdmin
 - getting 19-7
 - interface 19-19
 - methods

- obtain_pull_consumer 19-19
 - obtain_push_consumer 19-19
- supplier-consumer communication model 19-1
- suppliers
 - connecting to an EventChannel 19-7
- support
 - implementation and object activation 2-3
- support options 1-6
- SVCnameroot 18-7
- SVCnameroot property 18-9
- system exceptions
 - BAD_CONTEXT 5-1
 - BAD_INV_ORDER 5-1
 - BAD_OPERATION 5-1
 - BAD_PARAM 5-1
 - BAD_TYPECODE 5-1
 - catching 5-4
 - COMM_FAILURE 5-1
 - completion status, obtaining 5-2
 - CompletionStatus values 5-2
 - CORBA-defined 5-1
 - DATA_CONVERSION 5-1
 - FREE_MEM 5-2
 - handling 5-3
 - IMP_LIMIT 5-2
 - INITIALIZE 5-2
 - INTERNAL 5-2
 - INTF_REPOS 5-2
 - INV_FLAG 5-2
 - INV_INDENT 5-2
 - INV_OBJREF 5-2
 - MARSHAL 5-2
 - narrowing exceptions to 5-3
 - NO_IMPLEMENT 5-2
 - NO_MEMORY 5-2
 - NO_PERMISSION 5-2
 - NO_RESOURCES 5-2
 - NO_RESPONSE 5-2
 - OBJ_ADAPTOR 5-2
 - OBJECT_NOT_EXIST 5-2
 - PERSIST_STORE 5-2
 - SystemException class 5-1
 - TRANSIENT 5-2
- type
 - catching specific 5-4
- UNKNOWN 5-2

T

- technical support 1-6
 - contacting 1-6
- thread management 2-3
 - idle time 4-13
 - maximum number of threads 4-13
- TPool 4-13
- TSession 4-13

- threading
 - thread policies 8-2
 - thread pooling policy 8-2
 - thread-per-session policy 8-7
 - using synchronized block 8-10
 - worker threads 8-2, 8-7
- threads
 - multithreading, feature summary 2-3
- throwing exceptions, modifying object to 5-6
- throwing user exceptions 5-6
- _tie class
 - delegator implementation 9-1
 - how it works 9-1
- tools
 - administration 2-7
 - CORBA services 2-7
 - idl2cpp 4-3
 - idl2ir 2-6
 - oadutil unreg 20-12
 - oadutil, registering objects with 20-6
 - osfind 16-12
 - programming 2-6
 - vregedit 3-2, 3-3
- TPool 4-13
- tracing code
 - interceptors, using 24-1
- TRANSIENT exception 5-2
- TriggerHandler
 - interface 17-6
- TriggerHandler class 17-6
- triggers
 - creating 17-6
 - first instance only, looking at 17-6
 - trigger handler
 - implementing 17-10
 - what are triggers? 17-3
- truncatable valuetypes 28-9
- try_pull 19-7, 19-8
- TSession 4-13
- type
 - Any 23-7
 - contained in the Interface Repository 12-9
 - descriptions of in Interface Repository 21-1
 - determining for an object reference 10-4
 - exceptions
 - catching specific 5-4
 - instance, determining if is 10-4
 - sub-type, determining if is 10-4
- TypeCode class 22-10
- typecodes, Interface Repository, represented in 21-2
- types
 - DynAny 27-1
- typographic conventions 1-5

U

- UDP protocol 16-1
- unbind
 - nsutil 18-8
- UnionDef object 21-7
- UNKNOWN exception 5-2
- unmarshalling 28-8
- unreg_trigger()
 - method 17-5
- unregistering objects
 - OAD 20-12
 - oadutil, using 20-12
- URL
 - naming service 29-1
- url 18-17
- user exceptions
 - adding fields to 5-7
 - catching exceptions, modifying object to 5-6
 - defining 5-5
 - fields, adding to 5-7
 - throwing exceptions, modifying object to 5-6
 - UserException class 5-5
- utilities
 - idl2cpp compiler
 - how it generates code 15-2
 - interface inheritance, specifying 15-8
 - methods for attributes, generated by 15-7
 - oneway methods, defining 15-7
 - idl2ir 21-5
 - irep 21-3
 - oadutil
 - displaying contents of Implementation Repository 20-14
 - registering object implementations with 20-1
 - osagent 4-8

V

- valuetypes 28-1
 - abstract 28-2
 - abstract interfaces 28-7
 - boxed 28-7
 - compiling the IDL file 28-3
 - concrete 28-2
 - custom 28-8
 - defining 28-3
 - derivation 28-2
 - factories 28-2, 28-6
 - implementing 28-3
 - implementing factories 28-5
 - implementing the Factory class 28-4
 - inheriting valuetype base classes 28-4
 - overview 28-1
 - registering 28-6
 - registering your Factory with the ORB 28-5
 - truncatable 28-9
- variables, environment
 - OSAGENT_PORT, setting 3-3
 - PATH, setting 3-1
 - VBROKER_ADM, setting 3-2 to 3-3
- vbroker.naming.backingStore 18-16
- vbroker.naming.jdbcDriver 18-16
- vbroker.naming.loginName 18-16
- vbroker.naming.loginPwd 18-16
- vbroker.naming.poolSize 18-16
- vbroker.naming.url 18-17
- vbroker.security.peerAuthenticationMode 11
- VBROKER_ADM environment variable 16-5
 - log directory, specifying with 3-4
 - setting 3-2 to 3-3
- VBROKER_JAVAVM 1-3
- VBROKER_TAG 1-3
- version of product 2-6
- viewing
 - Console System Information 11-4
 - Interface Repositories 12-9
- VisiBroker
 - Console
 - using 11-1
 - CORBA compliance 2-6
 - described 2-2
 - features of 2-3
 - activating objects and implementations 2-3
 - compilers, IDL 2-4
 - connection management 2-3
 - dynamic invocation 2-4
 - IDL compilers 2-4
 - IDL interface to Smart Agent 2-3
 - implementation activation 2-3
 - implementation repository 2-4
 - interface repository 2-4
 - Location Service 2-3
 - multithreading 2-3
 - object activation 2-3
 - object database integration 2-5
 - Smart Agent architecture 2-3
 - thread management 2-3
 - introduction 1-1
 - new in this guide 1-4
 - new in this release 1-1
- VisiBroker for C++
 - additional information 1-6
- VISObjectWrapper::ChainUntypedObjectWrapper
 - adding factories 25-6
 - removing factories 25-8
- VISObjectWrapper::UntypedObjectWrapper
 - post_method 25-5
- vregedit tool 3-2, 3-3

W

ways to defer 32-1

web naming

 associating a URL with an object 29-1

web sites

 CORBA specification 1-6, 2-6

What is CORBA? 9-1, 19-1, 25-1, 27-1

What's new in 4.1 1-2, 1-3

widening object references 10-5