



O B J E C T S P A C E
VOYAGER

ORB 3.1

Developer Guide

O B J E C T S P A C E

© 1997-1999 ObjectSpace, Inc. All rights reserved.

ObjectSpace, Inc., has used its best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. ObjectSpace, Inc., makes no warranties of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. ObjectSpace, Inc., shall not be liable in any event for incidental or consequential damages in connection with, or arising from, the furnishing, performance, or use of these programs.

Voyager, Space, Dynamic XML, and Dynamic Aggregation are trademarks of ObjectSpace, Inc.

Java is a trademark of Sun Microsystems.

All other brand or product names are trademarks or registered trademarks of their respective holders.

RESTRICTED RIGHTS LEGEND

Voyager ORB and Voyager ORB Professional are furnished under a license and may not be used, copied, disclosed, and/or distributed except in accordance with the terms of said license. No classes, hierarchies, methods, binaries, or code may be copied for implementation in other systems.

This document and all online system documentation are copyrighted 1997-1999 by ObjectSpace, Inc. All rights reserved. No portion of this document may be copied, photocopied, reproduced, transcribed, translated, or reduced into any language, in any form, or by any means, without the prior written consent of ObjectSpace, Inc.

This document is subject to change without notice.

Software Version 3.1

Second Edition

Printed in the United States of America

Preface

Voyager™ is a family of products that includes a high performance, state-of-the-art Object Request Broker (ORB). This guide assumes a basic working knowledge of Java and explains the power and simplicity of Voyager for developing network Java applications.

This preface covers the following topics:

- ◆ reviewing Voyager requirements
- ◆ accessing Voyager documents
- ◆ setting your CLASSPATH for the Voyager utilities
- ◆ obtaining technical support
- ◆ submitting problem reports and suggestions
- ◆ requesting information about product updates

Reviewing Voyager Requirements

Before you install Voyager, ensure that you are set up to use the product. You need:

- ◆ A Java developer kit (JDK) installed on your computer. Voyager requires JDK 1.1 or later. You can download the latest release of JDK from www.javasoft.com at no charge.
- ◆ A working knowledge of the Java programming language.
- ◆ Approximately 2.5MB of available space on your hard drive to install Voyager.

Note: If you are running Sun Solaris, you must use JDK 1.1.7 or better to avoid a potential problem with earlier JDK versions that used non-native threads.

Accessing Voyager Documentation

The directory structure of Voyager follows:

voyager	
\bin	Voyager utilities
\doc	Documentation files
\examples	Example files
\lib	voyager.jar file (Voyager .class files)

You can access the following documentation by opening a browser on `voyager\doc\index.html`:

- ◆ *Voyager API Guide* – An online manual containing a complete API listing of Voyager public classes and methods.
- ◆ *Voyager ORB Developer Guide* – This manual is in .pdf format. You need Adobe Acrobat Reader 3.0 to view this document. If you are a Voyager ORB Professional user, a hardcopy of this manual is included with the shipped product.

You can also access these documents and others from the ObjectSpace web site at www.objectspace.com.

Setting your CLASSPATH for the Voyager Utilities

When executed, the Voyager utilities search for both your source code and your object code via CLASSPATH. The search is successful when your source files and object files reside in the same directory. If your source files and object files reside in different directories, ensure that the directory structure leading to source files mirrors the directory structure leading to object files. Add the root of each path to CLASSPATH.

For example, if the source code and object code for the `com.foobar` package is organized as follows, the CLASSPATH must include both `\root` and `\root\src`:

```
\root
  \com
    \foobar          .class files (object code)
  \src
    \com
      \foobar        .java files (source code)
```

In addition, when using JDK 1.2 or a non-Sun VM such as the Microsoft VM, the location of the Java classes must be set in the CLASSPATH. For example, if you have installed JDK 1.2 in `c:\jdk1.2`, the CLASSPATH should include `"c:\jdk1.2\jre\lib\rt.jar"`.

Obtaining Technical Support

The ObjectSpace web site contains information about ObjectSpace products. Visit www.objectspace.com and select the appropriate product to view answers to frequently asked questions (FAQ), read about known problems, review technical white papers, or download new versions of software. Several forums, such as discussion lists, news groups, and notification mailing lists, are available for selected products.

Submitting Problems and Suggestions

ObjectSpace welcomes problem reports and suggestions for improving Voyager. Send your valuable feedback to voyager@objectspace.com.

Requesting Product Updates

To receive automatic e-mail notification of new Voyager releases and other Voyager-related news items, join the Voyager discussion list. you automatically receive e-mail notification of new Voyager releases and other Voyager-related news items when you download Voyager from the ObjectSpace web site, unless you explicitly decline automatic notification. You can join the Voyager discussion list at any time by visiting www.objectspace.com.

Contents

Preface	iii
Reviewing Voyager Requirements	iv
Accessing Voyager Documentation	iv
Setting your CLASSPATH for the Voyager Utilities	v
Obtaining Technical Support	v
Submitting Problems and Suggestions	vi
Requesting Product Updates	vi
1 Overview	1
Voyager ORB Benefits	2
Voyager ORB Features	7
2 Basics	11
Starting and Stopping a Voyager Program	12
Starting a Voyager Server from the Command Line	13
Using Interfaces for Distributed Computing	14
Creating a Remote Object	15
Sending Messages and Handling Exceptions	16
Logging Information to the Console	17
Understanding Distributed Garbage Collection	18
Using Naming Services	19
Working with Proxies	19

Exporting Objects	21
Loading Classes	22
Serving Classes	23
Remote-Enabling a Class that has No Interface	23
Using Threads	24
Persistence	25
Multihome Support	26
3 Dynamic Aggregation™	27
Working with Dynamic Aggregation	28
Accessing and Adding Facets	29
Selecting a Facet Implementation	31
Packaging Facets	32
Creating Facet-Aware Classes	32
4 Advanced Messaging	33
Invoking Messages Dynamically	34
Retrieving Remote Results by Reference	38
Using Multicast and Publish/Subscribe	39
5 Mobility and Agents	45
Moving an Object to a New Location	46
Obtaining Move Notification	48
Understanding the Uses for Mobile Agents	49
Creating Mobile Agents	50
Code Mobility	51
6 Naming Service	53
Using a Namespace	54
Working with Federated Directory Services	55
Using the Default Name Service	57
Using JNDI	58
Using PersistentDirectory	59

7	Activation	61
	Enabling an Object for Activation	62
	Activating an Object	64
	Writing an Activator	65
8	Security	67
	Installing a Security Manager	68
	Identifying Object Authority	69
9	Ultra-Light Client, Applets and Servlets	71
	Implementing Voyager Ultra-Light Client	72
	Understanding Ultra-Light Client Limitations	73
	Packaging Voyager Ultra-Light Client	73
	Using Voyager with Servlets	74
	Using Voyager with Applets	74
10	CORBA	77
	Building an Application	79
	Struct	90
	Holders	92
	Typedef	94
	Const	95
	Constant Expressions	96
	Arrays and Sequences	98
	Modules and Interfaces	99
	Inheritance	102
	Scoping	103
	Enum	105
	Union	107
	TypeCode and Any	110
	Attributes	111
	User Exceptions	112
	System Exceptions	113
	Standard Object Methods	114

	Narrowing.....	115
	Java to IDL	115
	Prefixes, Versions, and Repository IDs	119
	Repackaging	120
	Naming Service	124
	Transactions	125
	CORBA Wide Character and Wide String Support	125
11	RMI	127
	Using Voyager as an RMI Client	128
	Using Voyager as an RMI Server.....	128
	Using Universal Directory Integration	129
12	DCOM	131
	Communicating from Voyager Client to COM Component.....	132
	Communicating from COM Client to Voyager Object.....	138
	Troubleshooting	147
13	Dynamic XML	151
	Understanding XML	152
	Manipulating XML Using DOM.....	154
	Manipulating XML using DXML	157
	How DXML Maps DTDs to Java	162
14	Timers	169
	Clocking Time Intervals.....	170
	Using Timers and TimerEvents.....	171
15	Replication	175
	Understanding Replication	176
	Using Replication	177
	Adding Directories	179
	Using a Load Balancing DNS Server.....	180

16	Load Balancing	181
	Balancing Application Load with Voyager	182
	Creating a Load Balanced Directory Server	187
	Working with Load-Balanced Directories	187
17	Configuration and Management	193
	Understanding Voyager Properties	194
	Specifying a Properties File	197
	Specifying Multiple Values	197
	Using the Voyager Management Framework	198
	Working with the ObjectSpace Workshop Framework	199
	Using the Voyager Management Console	200
18	Policy Management	203
	Understanding IPolicy	204
	Understanding Policy Events	205
19	TCP Connection Management	209
	Understanding Connection Management Policies	210
	Understanding Case Policies	211
	Establishing Case Policies	212
	Defining Policy Listeners	213
	Examples	213
	Using the TCP Connection Management GUI	215
A	Utilities	223
	voyager	224
	igen	227
	cgen	229
	pgen	231
B	Examples	233
	Running the Examples	234
	Basics	235
	Dynamic Aggregation	242

Advanced Messaging	257
Multicast and Publish/Subscribe	272
Mobility	284
Agents.....	290
Naming Service	294
Activation	302
Security.....	312
Applets and Servlets.....	317
CORBA	325
RMI	337
DCOM	346
Ultra-Light Client	353
Timers.....	357
Replication Examples.....	365
Load Balancing Examples.....	369
Configuration and Management.....	380
Configuration	399
Universal Gateway	407
Dynamic XML	418

Overview 1

Voyager is ObjectSpace's product family for distributed computing that simplifies and unifies the most common industry standards. Voyager helps organizations produce high-impact, distributed systems quickly. Voyager includes the following products:

- ◆ Voyager ORB is a high-performance, full-featured object request broker that simultaneously supports CORBA, RMI, DCOM. Its innovative dynamic proxy generation removes the need for stub generators. Voyager ORB includes a universal naming service, DCOM support, activation framework, publish/subscribe and mobile agent technology.
- ◆ Voyager ORB Professional builds on the Voyager ORB foundation with a graphical management console, configuration framework, JNDI integration, persistent directory, CORBA naming service, and support for ultra-light clients.
- ◆ Voyager Security includes a flexible security framework, lightweight security implementation, support for secure network communications via SSL adapters, and firewall tunneling using the SOCKS protocol.
- ◆ Voyager Transactions delivers full OTS-compliant distributed transactions support, including two-phase commit and a one-phase commit JDBC adapter.
- ◆ Voyager Application Server offers a true EJB development environment that decouples application logic from systems programming logic.

Voyager ORB Benefits

The Voyager ORB offers many benefits, including:

Universality

The Voyager ORB simplifies and unifies access to the most common industry standards. There are several aspects of Voyager that are universal:

- ◆ Communications

The universal communications architecture allows Voyager programs to be both a universal client and a universal server by supporting simultaneous bi-directional communication with other CORBA, RMI, and DCOM programs.

- ◆ Messaging

The universal messaging layer allows different types of messages such as synchronous, oneway, and futures to be sent to an object regardless of its location or object model.

- ◆ Naming

The universal naming service allows access to the many commercially available naming services through a single API.

- ◆ Directory

The universal directory is a single directory that can be accessed and shared by all clients. for example, an RMI server can bind an object into a universal directory using the native RMI registry API and a CORBA client can lookup up the same object using the CORBA naming service API.

- ◆ Gateway (Voyager ORB Professional Only)

The universal gateway allows voyager to automatically bridge protocols between clients and servers that are not written using Voyager. For example, this allows a native RMI client to send messages to an object in a native CORBA server, even though RMI does not support IIOP.

Ease of Development

Voyager greatly simplifies the creation of distributed systems. For example, Java classes do not have to be modified in order to be remote enabled. In addition, Voyager generates the glue needed for distributed computing at runtime, removing the need for stub generators, skeletons, and helper classes.

Architectural Flexibility

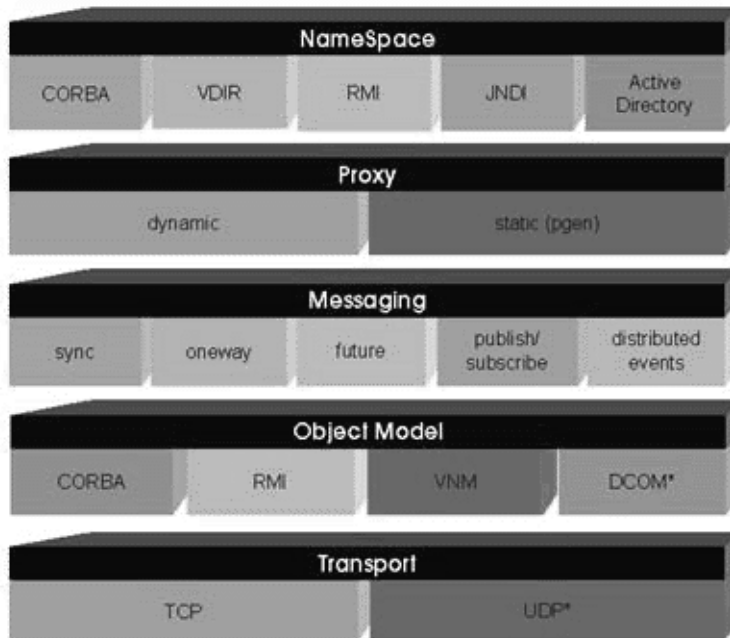
Since it is difficult to anticipate each customer's unique requirements, Voyager has been built from components that can be extended or replaced to integrate into a customer's existing computing infrastructure. For example, some of our customers modify the transport layer so they can use an internally developed network.

The following diagrams illustrate these features:

Universal Architecture

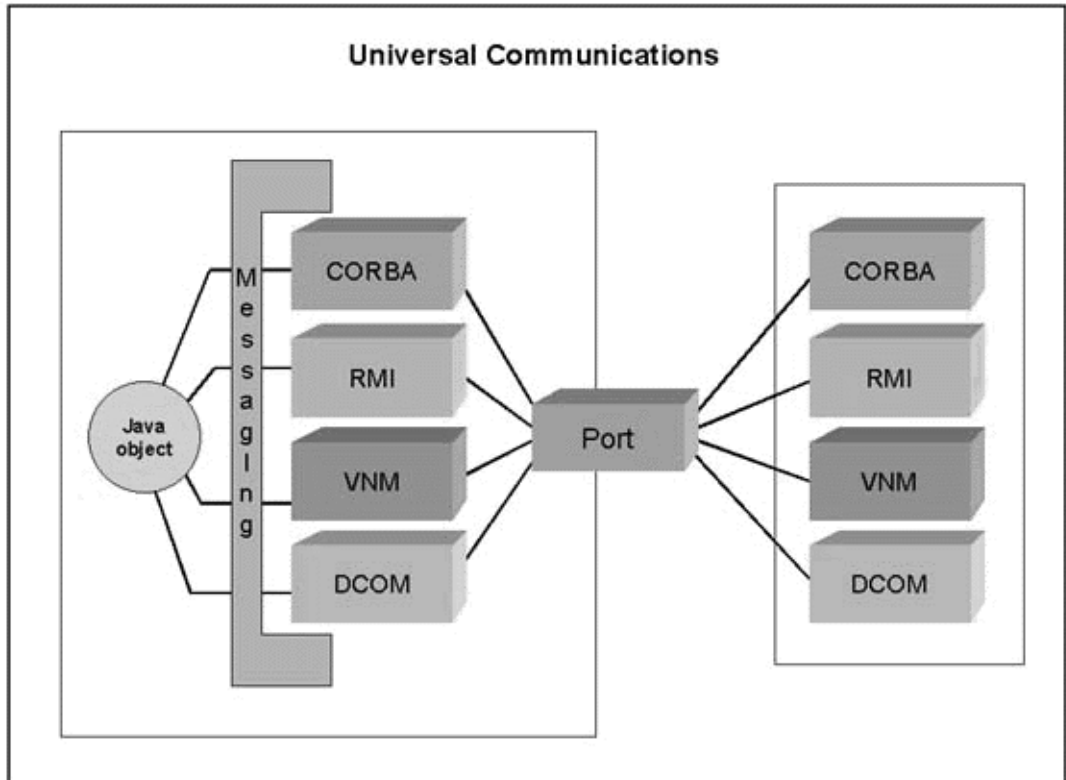
Voyager offers a universal architecture that isolates user code from the intricacies of communications and messaging protocols. The following figure depicts Voyager's universal architecture.

Universal Architecture



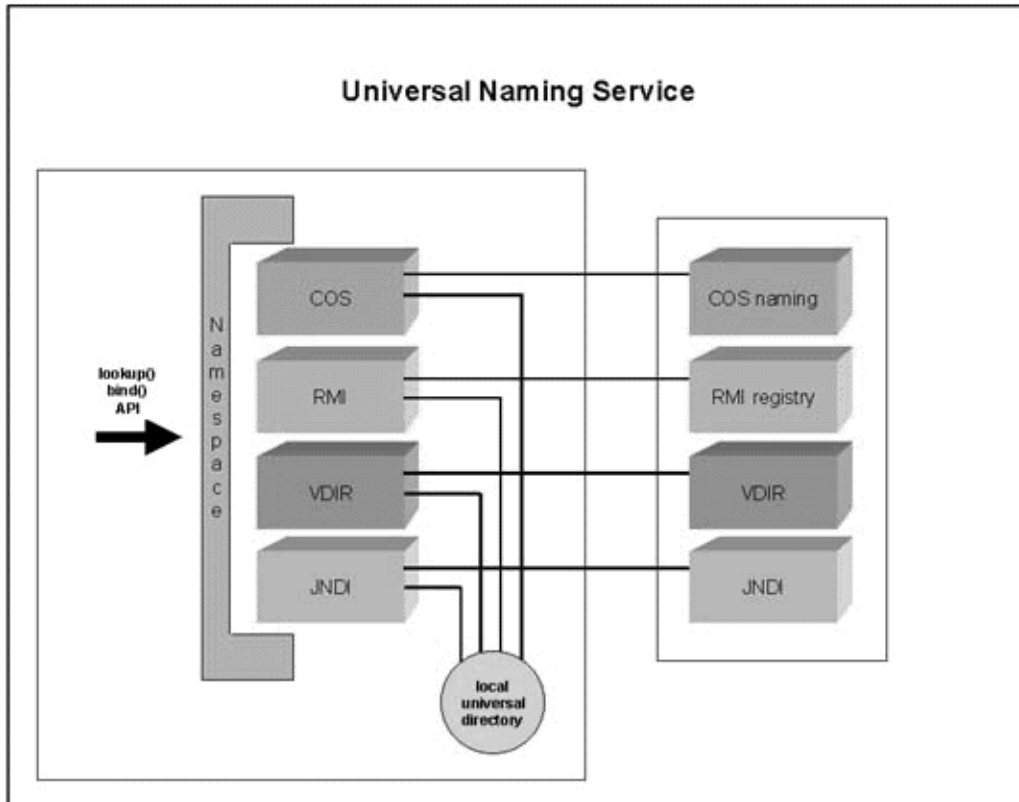
Universal Naming Service

In addition to a universal architecture, Voyager offers users a universal naming service. The following figure depicts this universal naming service.



Universal Communications Infrastructure

In addition to a universal architecture, Voyager offers users a universal communications infrastructure. The following figure depicts this universal communications infrastructure.



Voyager ORB Features

Voyager offers developers multiple, robust features, including:

Remote-Enabling a Class. Java classes are remote-enabled classes at runtime. A class does not have to be modified in any way, and no additional files are created.

Remote Construction. You can create a remote instance of any class and obtain a proxy to the newly created object. The proxy implements the same interfaces as the created object, and the proxy class is generated dynamically if it doesn't already exist.

Dynamic Class Loading. Classes can be dynamically loaded from one or more locations when necessary. This allows you to easily set up class repositories that serve your corporate Java applications.

Remote Messaging. Method calls made to a proxy are forwarded to its object. If the object is in a remote program, the arguments are serialized using the standard Java serialization mechanism and deserialized at the destination. The morphology of the arguments is maintained. By default, parameters are passed by value. However, if an object's class implements `IRemote` or `java.rmi.Remote`, the object is passed by reference instead.

Exception Handling. If a remote exception occurs, it is caught at the remote site and rethrown locally. If the appropriate logging level is selected, a complete stack trace from the remote site is displayed to the console.

Distributed Garbage Collection. The distributed garbage collector reclaims objects when there are no more local or remote references to them. It uses an efficient "delta pinging" algorithm that keeps the traffic required for garbage collection to a minimum. You can also fine-tune the behavior of the distributed garbage collection mechanism and receive notification of DGC events.

Dynamic Aggregation™. This feature allows you to add secondary objects (termed *facets*) to a primary object at runtime. For example, you can dynamically add hobbies to an employee, a repair history to a car, or a payment record to a customer. Dynamic aggregation represents a fundamental step forward for object modeling and complements the traditional mechanisms of inheritance and polymorphism.

CORBA. There is full native support for IDL, IIOP, and bidirectional IDL<->Java translation. No stub generators or helper classes are required. Voyager also provides CORBA wide-character support.

RMI. Voyager provides full RMI support. This means that you can easily use classes in Voyager that were originally designed for use with RMI.

Mobility. You can move any serializable object between programs at runtime. If a message is sent from a proxy to an object's old location, the proxy is automatically updated with the new location and the message is resent. Mobility is often useful when optimizing message traffic in a distributed system.

Autonomous Mobile Agents. You can create mobile autonomous agents that move themselves between programs and continue to execute upon arrival. It is easy to build agents that utilize movement to more efficiently satisfy their goals.

JNDI (Voyager ORB Professional only). Certain client processes may need to access named objects using the Java Naming and Directory Interface (JNDI). Voyager implements JNDI as a wrapper around its Federated Directory Service.

Persistent Naming Service (Voyager ORB Professional only). Voyager supports persistent naming service, allowing you to store the contents of the PersistentDirectory in a file in the local file system.

Activation. The activation framework allows objects to be persisted to any kind of database and automatically re-activated in the case that the program is restarted. An object does not have to be modified in any way to be activatable.

Ultra-Light Client (Voyager ORB Professional only). Voyager supports an ultra-light client with a footprint of approximately 15K, making it an ideal candidate for applet and small-footprint clients. Voyager Ultra-Light Client provides a mechanism for developing a 100% pure Java client with minimal client-code overhead by allowing applications to communicate with remote object implementations in the same fashion as if they were local.

Applets and Servlets. It is easy to create Voyager-enabled applets and servlets. Because applets cannot open network connections to any machine except their server, Voyager allows you to set up a server-side hub that can perform message routing and dynamic proxy generation on the applet's behalf.

Naming Service. The naming service provides a single, simple interface that unifies the many commercially available naming services. New naming services can be dynamically plugged into Voyager's naming service.

Multicast. You can multicast a Java message to a distributed group of objects without requiring the sender or receiver to be modified in any way.

Publish-Subscribe. You can publish a Java event on a specified topic to a distributed group of subscribers. The publish-subscribe facility supports server-side filtering and wildcard matching of topics.

Management Console (Voyager ORB Professional only). You can configure and manage multiple Voyager servers and Voyager services from a centralized, customizable graphical user interface.

Timers. A Stopwatch and Timer class facilitate common timing chores. Timer events can be distributed and multicast if necessary.

Thread Pooling. A thread pool is used when allocating and deallocating threads, resulting in higher performance.

Advanced Messaging. You can send oneway, sync, and future messages. Oneway messages return immediately and discard the return value. Future messages immediately return a placeholder to the result, which may then be polled or read in a blocking fashion.

Security. An enhanced security manager is included, as well as hooks for installing custom sockets such as SSL.

Replicated Directory Service (Voyager ORB Professional only). You can create a cluster of identical directory servers on separate systems, ensuring data availability for lookups.

Load-balanced Directory Service (Voyager ORB Professional only). You can balance application load across multiple servers, increasing performance and responsiveness.

Dynamic XML (Voyager ORB Professional only). Provides you the tools to access XML documents as regular Java classes. XML documents may also be accessed remotely.

DCOM Bridge (Voyager ORB Professional only). With Voyager's DCOM Bridge, remote DCOM objects are transparently accessible to Voyager objects, and Voyager objects are accessible as DCOM objects.

Policy Management (Voyager ORB Professional only). Voyager provides a framework for easy integration of policies and policy configuration, and a policy event notification mechanism.

TCP Connection Management (Voyager ORB Professional only). Connection management tools allow you to manage the number of live and idle connections for a Voyager server to prevent server overload.

Multi-home Support. Voyager supports multi-homed systems. A multi-homed system is one with multiple hostnames/IP addresses.

This manual focuses on detailed information on these features of the Voyager ORB and Voyager ORB Professional products.

Basics

2

This chapter covers all the features of Voyager that are required to build a simple distributed application.

In this chapter, you will learn to:

- ◆ start and stop a Voyager program
- ◆ start a Voyager server from the command line
- ◆ use interfaces for distributed computing
- ◆ create a remote object
- ◆ send messages and handle exceptions
- ◆ understand distributed garbage collection
- ◆ use the naming service
- ◆ work with proxies
- ◆ export objects
- ◆ load classes
- ◆ serve classes
- ◆ remote-enable a class that has no interface
- ◆ use threads

Starting and Stopping a Voyager Program

A program must invoke one of the following variations of `Voyager.startup()` before it can use any Voyager features:

- ◆ `startup()`

Starts Voyager as a client that initially does not accept incoming messages.

- ◆ `startup(String url)`

Starts Voyager as a server that accepts incoming messages, either on the specified URL or on a random unused port when the URL is null.

- ◆ `startup(Object object, String url)`

Required by Voyager-enabled Applets and Servlets, as described in the "Ultra-Light Client, Applets and Servlets" chapter.

The general format of a URL (uniform resource locator) follows:

`protocol://host:port/file#reference;argument`

Voyager extends the URL syntax by allowing nested protocols such as `ssl:tcp:`. Each part is optional. During startup, you only need to specify the port. Using Voyager on multi-homed systems can be done one of two ways. First you can explicitly export a proxy on a URL. Second, you can call `transport.Transport.acquireServer(XURL)` to start a server on that URL. If your machine is multi-homed with multiple host names, you can either explicitly specify the host or omit it and allow your operating system to choose the primary host. The special host "localhost" is interpreted as your local host. Examples follow:

```
Voyager.startup(); // startup as a client
Voyager.startup( null ); // startup as a server on a random unassigned port
Voyager.startup( "8000" ); // startup as a server on port 8000
Voyager.startup( "//dallas:7000" ); // startup as server on port dallas:7000
```

To shutdown a Voyager program, invoke `Voyager.shutdown()`. This method kills the Voyager internal non-daemon threads and allows the main program to terminate.

The startup and shutdown methods generate SystemEvents that you may listen to using `Voyager.addSystemListener()`.

Starting a Voyager Server from the Command Line

To start a Voyager server from the command line, use the `voyager` utility. This utility starts an empty Voyager program that accepts objects and messages from other Voyager programs until it is explicitly terminated from the command line using Control-C.

For example, to start an empty Voyager program that accepts connections on port 8000, type:

```
>voyager 8000  
voyager orb 3.1, copyright 1997-1999 objectspace
```

The `voyager` utility has several options. For more information, refer to [Appendix A, "Utilities"](#).

Using Interfaces for Distributed Computing

The Java language supports *interfaces*. An interface contains no code. It defines a set of method signatures that must be defined by the class that implements the interface. A variable whose type is an interface may refer to any object whose class implements the interface. By convention, Voyager interfaces begin with “I”, although your code is exempt from this rule. An example of an interface follows:

```
public interface IStockmarket
{
    int quote( String symbol );
    int buy( int shares, String symbol );
    int sell( int shares, String symbol );
    void news( String announcement );
}
```

If the class `Stockmarket` implements `IStockmarket`, it is legal to write:

```
IStockmarket market = new Stockmarket(); // market refers to local object
```

A remote object is represented by a special *proxy* object that implements the same interfaces as its remote counterpart. A variable whose type is an interface may refer to a remote object via a proxy, because both the remote object and its proxy implement the same interfaces.

See the section ["Remote-Enabling a Class that has No Interface"](#) for information about remote-enabling a class that does not implement an interface.

Creating a Remote Object

To create an object at a specified location, use `Factory.create()`. This method returns a proxy to the newly created object and creates the proxy class dynamically if it does not already exist.

There are several variations of `create()`, depending on whether the object is to be created locally and whether the class constructor takes arguments. You must always fully qualify the name of the class. For example, use `java.util.Vector` instead of `Vector`. To create a default instance of `Stockmarket` in the local program and another in the program running on port 8000 of the machine “dallas”, type:

```
IStockmarket market1 = (IStockmarket) Factory.create( "Stockmarket" ); // created locally
IStockmarket market2 = (IStockmarket) Factory.create( "Stockmarket", "//dallas:8000" ); // created remotely
```

To create an instance of `Stockmarket` and use the constructor that takes a `String` and an `int`, type:

```
Object[] args = new Object[] { "NASDAQ", new Integer( 42 ) };
IStockmarket market3 = (IStockmarket) Factory.create( "Stockmarket", args, "//dallas:8000" );
```

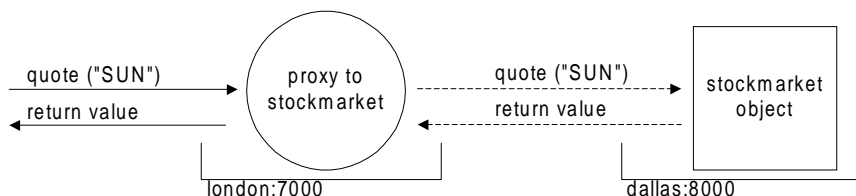
Note that primitive arguments must be wrapped in their `Object` equivalents.

Sending Messages and Handling Exceptions

A message sent via a proxy is executed according to the following rules:

- ◆ If the destination object is in the same program, the message is delivered just like a regular Java message. The arguments are not serialized or copied, resulting in very high performance.
- ◆ If the destination object is in a different program, the arguments and return value must be sent across the network. If an argument implements `com.objectspace.voyager.IRemote` or `java.rmi.Remote`, a proxy to the argument is sent (pass by reference), otherwise a copy of the argument is sent using standard Java serialization (pass by value). Morphology of the arguments is maintained—an object that is an argument or part of an argument is copied exactly once, and an argument or part of an argument that shares an object in the local program also shares a copy of the object in the remote program. Rules for an argument also apply to a return value.

The following figure shows how a remote message is processed:



If a remote method throws an exception, it is caught and re-thrown in the local program. If a Voyager-related exception, such as a network error, argument serialization error, etc., occurs and the interface method explicitly throws `java.rmi.RemoteException`[†], the exception is thrown wrapped in a `RemoteException`, otherwise it is thrown wrapped in a `com.objectspace.voyager.RuntimeRemoteException`. `RuntimeRemoteException` is also thrown if a method invoked on an object via a proxy throws an exception that is not declared in the throws clause for that method. In each case, the public detail field contains the original exception. Voyager's exception handling policy allows you to select between checked

[†]Microsoft developers can download the Java RMI API from the Microsoft website.

and unchecked exceptions. If you prefer checked exceptions, add “throws `java.rmi.RemoteException`” to every method in an interface.

Example

The [Basics1 Example](#) demonstrates basic messaging and remote construction.

Logging Information to the Console

The `Console` class allows you to log information, including stack traces of remote exceptions, to the console. Use `Console.setLogLevel()` to select a logging level:

- ◆ `Console.SILENT`

Displays no output to the console.

- ◆ `Console.EXCEPTIONS`

Displays stack traces of remote exceptions and unhandled exceptions to the console.

- ◆ `Console.VERBOSE`

Displays stack traces of remote exceptions, unhandled exceptions, and internal debug information to the console.

To set the logging level from the command line, use the `voyager -l` option with the `silent`, `exceptions`, or `verbose` argument.

Understanding Distributed Garbage Collection

Voyager's distributed garbage collector (DGC) reclaims objects when they are no longer pointed to by any local or remote references.

Voyager uses an efficient “delta ping” scheme to reduce DGC network traffic. Each program notes when references to remote objects are created and destroyed. In each DGC cycle, which is 2 minutes by default, the program sends each referenced remote program a single message containing a summary of the references to its objects that were added/removed since the last DGC cycle. By tracking this information as it changes over time, each program can tell when no remote references exist to an exported object.

DGC Notification

DGC notification support has been added to Voyager 3.1. If a class is interested in being notified when a remote reference to an object of the class is about to be discarded by DGC, it can implement the `com.objectspace.voyager.vrmp.dgc.IDGCListener` interface. The callback function `discardingReference()` is invoked when a remote reference to the object is about to be discarded. The object has the option to allow or delay discarding the reference. See the API documentation for `com.objectspace.voyager.vrmp.IDGCListener` for more details.

DGC Discard Delay Configuration

DGC reference discard delay configuration support has been added to Voyager 3.1. The `DGC.setDiscardDelay` method sets the delay between the time a remote reference is last used and the time the reference is discarded by DGC. See the API documentation for `com.objectspace.voyager.vrmp.DGC` for more details.

Using Naming Services

The Voyager integrated naming service provides unified access to a variety of commercial naming services. This section shows how to use the naming service to bind names to objects for later lookup. For more detailed information, including examples and information on how to rebind and unbind names, refer to the ["Naming Service"](#) chapter.

To bind a name to an object, invoke `Namespace.bind()` with the name expressed as an URL. The following code segment creates a `Stockmarket` on the host `//dallas:8000` and then binds it to the name `"NASDAQ"` for later lookup:

```
IStockmarket market = (IStockmarket) Factory.create( "Stockmarket", "//dallas:8000" );  
Namespace.bind( "//dallas:8000/NASDAQ", market );
```

The construction and binding step may be combined as follows:

```
IStockmarket market = (IStockmarket) Factory.create( "Stockmarket", "//dallas:8000/NASDAQ" );
```

To obtain a proxy to a named object, invoke `Namespace.lookup()`. The following example obtains a proxy to the object that was created and named by the previous code segment:

```
IStockmarket market = (IStockmarket) Namespace.lookup( "//dallas:8000/NASDAQ" );
```

Working with Proxies

All proxy classes extend `Proxy`. If a proxy class does not already exist, the Voyager class loading system generates it dynamically. Use any of the following to get a proxy to an object:

- ◆ `Factory.create(String classname, String url)`

Returns a proxy to a newly created object, where *classname* is the name of the class that you are creating an instance of, and *url* and *url* specifies where the object should be created.

- ◆ `Namespace.lookup(String name)`

Returns a proxy to the object with a particular name.

- ◆ `Proxy.of(Object object)`

If the specified object is already a proxy, returns the object; otherwise returns a proxy to the object.

A method call on a proxy is forwarded to its associated object unless it is one of the following special methods:

- ◆ `getClass(), notify(), notifyAll(), wait()`

These methods are all final methods in `Object` and are executed directly by the proxy.

- ◆ `hashCode()`

Returns the hash code of the proxy itself. Use `remoteHashCode()` to obtain the hash code of a proxy's associated object. Two proxies return the same hash code if they refer to the same object.

- ◆ `equals()`

Returns true if the argument is a proxy that refers to the same object as the receiver. Use `remoteEquals()` to compare the proxy's associated object with another object.

Additional methods in `Proxy` follow:

- ◆ `isLocal()`

Returns true if the proxy is in the same VM as its associated object.

- ◆ `getLocal()`

If the proxy is in the same VM as its associated object, returns a direct reference to the object; otherwise returns null

- ◆ `getURL()`

Returns the URL of the proxy's associated object.

- ◆ `toExternalForm()`

Returns a string that, when passed to `Namespace.lookup()`, returns a proxy to this proxy's object. This method is useful if you need to transmit a reference to an object via a medium like e-mail.

To pass an object by reference, either explicitly pass a proxy obtained using `Proxy.of()`, or implicitly pass a proxy by ensuring that the object class implements `com.objectspace.voyager.IRemote` or `java.rmi.Remote`.

Exporting Objects

To receive remote messages, an object must be *exported* to exactly one local URL. After it is exported, all remote messages to an object arrive via its export URL.

If a proxy to an unexported object is passed to a remote program, Voyager automatically exports the object to the default URL. If Voyager was started on an explicit URL, the default URL is the startup URL, otherwise the default URL is initialized to a random unassigned local URL.

The automatic export mechanism is sufficient for most applications. However, there are times where it is useful to partition objects between more than one URL. For example, security reasons might dictate to associate one group of objects with a URL that is connected to an intranet, while associating another group of objects with a URL connected to the Internet via SSL. Because programs on the Internet can only communicate via the SSL URL, they can only send messages to the group of objects that are exported on that URL.

To explicitly export an object, use one of the following static methods in Proxy:

- ◆ `export(Object object, String url)`

Exports the object to the specific URL. If connections are not already being accepted on the specified URL, automatically starts a new connection thread on the URL.

- ◆ `export(Object object)`

Exports the object to the default URL.

- ◆ `unexport(Object object)`

Unexports the object.

Note: An exported object can receive messages on exactly one transport protocol (TCP, SSL, etc.).

Example

The [Basics2A and Basics2B Examples](#) binds a name to an object exported on an explicit port.

Loading Classes

A Voyager program attempts to load new resources, usually classes, according to the following sequence:

1. Search the CLASSPATH.
2. Search the installed resource loaders from highest priority to lowest priority.

By default, a Voyager program has a single pre-installed ProxyResourceLoader at priority level 5. This loader can dynamically generate and load a proxy class from its original class.

To enable a program to load resources from remote sources, such as a web server or an SQL database, you must add more resource loaders using `VoyagerClassLoader.addResourceLoader()`.

For example, to enable class loading from a specific URL, add a URLResourceLoader constructed on a URL as follows:

- ◆ for classes in a directory not in the CLASSPATH, use `file:///full/directory/path/` (three forward slashes are intentional)
- ◆ for classes on a web server, use `http://host:port/root/`
- ◆ for classes on an HTTP-enabled Voyager program, described in "[Serving Classes](#)", use `http://host:port`

To disable Voyager's use of resource loaders (including the dynamic proxy generator), and rely solely on Java's class loader, invoke `VoyagerClassLoader.setResourceLoadingEnabled(false)`.

If Voyager is started from the command line, use `-c URL` to add an URLResourceLoader on the specified URL. The `-c` option may be specified many times on the same command line. For example, to start a Voyager server that can load classes from the directory `/bin/classes/` and from an HTTP server running on `//dallas:8000`, type:

```
voyager 8000 -c file:///bin/classes/ -c http://dallas:8000
```

To manually load a class using Voyager's class loading machinery, use `ClassManager.getClass(classname)` instead of `Class.forName(classname)`.

Serving Classes

A Voyager program can serve other programs with any resource that it can load. This is made possible by Voyager's built-in HTTP capability, which is disabled by default for security reasons. To HTTP-enable a Voyager program, invoke `ClassManager.enableResourceServer()`.

If Voyager is started from the command line, use the `-r` option to enable its HTTP server.

Example

The security example on page [Security1 Example](#) demonstrates a Voyager client capable of loading classes from a remote Voyager server that is HTTP-enabled. It also demonstrates some of the security consequences that you may want to address when using this form of classloading.

Remote-Enabling a Class that has No Interface

Voyager allows an object to be constructed remotely and sent messages even if its class does not implement an appropriate interface.

The `igen` utility generates a *default interface* from a class. A default interface has the public methods of the original class and is named using “I” followed by the name of the original class. If the original class is in the `java.*` package, the default interface is placed in the `com.objectspace.java.*` package, otherwise it is placed in the same package as the original class.

For example, to generate the default interface `com.objectspace.java.util.IVector` from a `java.util.Vector`, type:

```
igen java.util.Vector
```

When a proxy class is dynamically generated from a class that does not implement `java.rmi.Remote`, `com.objectspace.voyager.IRemote`, or its default interface, the default interface is automatically generated and added to the list of classes that the proxy class

implements. An instance of the proxy class implements the default interface even though its associated object does not.

To construct a remote `java.util.Vector` and send it messages, write:

```
import com.objectspace.java.util.IVector;

IVector vector = (IVector) Factory.create( "java.util.Vector", "//dallas:8000" );
// note that the proxy implements IVector even though Vector does not
vector.addElement( "hi" );
```

For more information about `igen`, see [Appendix A, "Utilities"](#).

Using Threads

To reduce the significant overhead of creating and destroying threads, Voyager uses a thread pool. If Voyager needs a thread, it only creates a new `Thread` object when it cannot reuse a thread from the pool. When the thread finishes, it is added to the pool unless the maximum pool size has been reached, in which case the `Thread` object is destroyed. Although there is a maximum pool size, there is no limit to the number of threads that Voyager can allocate.

The pool is initially empty and has an infinite maximum size. To change the maximum pool size, invoke `ThreadManager.setPoolSize()`, or use the `-t` option when using the `voyager` utility.

To include thread pools in your own applications, use `com.objectspace.lib.thread.ThreadPool`.

Persistence

Voyager does not include any kind of persistent storage, which is handled by database products. Instead, Voyager includes an *activation framework*, described in [Chapter 7](#), "[Activation](#)", that allows objects to be automatically loaded on demand from any kind of database. How and when objects are actually committed to the database is determined by the application developer.

Facilities like the activation framework that save and restore objects need to access the complete state of an object, including its facets described in [Chapter 3](#), "[Dynamic Aggregation™](#)" and Voyager-related properties, such as the object's export URL. The Snapshot class provides this functionality.

To persist an object and all of its state to a database, obtain a Snapshot of the object and then either store the Snapshot directly into the database or store its parts individually. To obtain a Snapshot of an object, use `Snapshot.of(object)`. The fields are set to each part of the object's state and can be accessed using the following methods:

- ◆ `getObject()`
Returns the object.
- ◆ `getProperties()`
Returns an instance of Properties containing the object's Voyager-related properties.
- ◆ `getFacets()`
Returns an array of the object's facets.

To load an object and its associated state back into memory, first recreate the original Snapshot either by loading it directly from the database or by loading the individual parts and then using `Snapshot.from(object, properties, facets)`. Then invoke `Snapshot.restore()`, which recreates the original object state from the individual fields of the Snapshot and returns a proxy to the newly created object.

Example

The [Activation1A and Activation1B Example](#) illustrates Snapshot.

Multihome Support

Multihomed computers are machines that are configured with more than one host name or IP address. Support for multihomed computers has been added to Voyager 3.1. When a Voyager server is started, it is running on a host name or IP address specified in the startup URL. To make Voyager aware of another host name or IP address, use the `addMultiHome()` method in class `com.objectspace.voyager.tcp.TcpTransport`. The method takes one parameter, which is a host name or an IP address as a string. See the API documentation for `com.objectspace.voyager.tcp.TcpTransport`.

For example, suppose we have a machine with host name `abc.mycorp.com` and another host name `xyz.mycorp.com`. Voyager starts with `abc.mycorp.com`. To make Voyager aware of the host name `xyz.mycorp.com`, use `addMultiHome()`. The method must be invoked before starting Voyager.

```
example.java
import com.objectspace.voyager.Voyager;
import com.objectspace.voyager.tcp.TcpTransport;

public class example
{
    public static void main(String[] args)
    {
        TcpTransport.addMultiHome( "xyz.mycorp.com" );
        Voyager.startup( "//abc.mycorp.com:8000" );

        // other stuff
    }

    //other methods, etc
}
```

The Voyager server is running on port 8000 of `abc.mycorp.com`. It is also known to be running on the same port of `xyz.mycorp.com`.

Dynamic Aggregation™

3

Voyager supports *dynamic aggregation*, which allows you to attach new code and data to an object at runtime.

This feature resolves the following problems that commonly occur during the construction of an object-oriented system:

- ◆ Adding behavior to a third-party component whose source is not available.
- ◆ Customizing an object in a subsystem-specific way, so it can be used by multiple subsystems.
- ◆ Extending an object's behavior at runtime, perhaps in unforeseen ways.

Dynamic aggregation represents a fundamental step forward for object modeling and complements the mechanisms of inheritance and polymorphism.

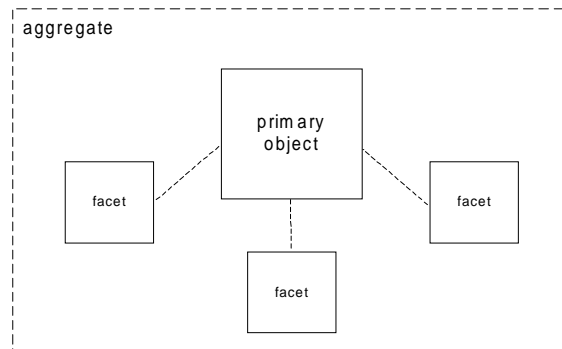
In this chapter you will learn to:

- ◆ work with dynamic aggregation
- ◆ access and add facets
- ◆ select a facet implementation
- ◆ create facet-aware classes

Working with Dynamic Aggregation

Dynamic aggregation allows you to attach secondary objects, or *facets*, to a primary object at runtime. A primary object and its facets form an *aggregate* that is typically persisted, moved, and garbage-collected as a single unit.

The following diagram illustrates a primary object and its facets.



There are several rules associated with facets:

- ◆ A class does not have to be modified in any way for its instances to play the role of a primary object and/or facet.
- ◆ The class of a facet does not have to be related in any way to the class of a primary object. An instance of a class can be added as a facet to any kind of primary object.
- ◆ Facets cannot be nested. In other words, a facet cannot have a facet.
- ◆ Facets cannot be removed. After a facet is added, it remains for the life span of the aggregate.
- ◆ A primary object and its facets have the same life span and are garbage-collected only when there are no references to either the primary object or any of its facets.

There are many uses for dynamic aggregation. For example, you can dynamically add a bonus plan facet to an employee, a repair history facet to a car, a payment record facet to a customer, or a hyperlinks facet to a generic object.

Accessing and Adding Facets

A primary object's facets are represented by an instance of Facets that is initially set to null. To access an object's Facets, use one of the following static Facets methods:

- ◆ `get(Object object)`

Returns the object's Facets, which may be null.

- ◆ `of(Object object)`

Returns the object's Facets, setting it to an initialized instance of Facets when it is currently equal to null.

Because a facet is part of an aggregation, invoking `Facets.get()` or `Facets.of()` on a facet returns the Facets instance of the facet's primary object.

To manipulate an object's facets, use the following instance methods defined in Facets:

- ◆ `get(String interfacename)`

Returns a proxy to a facet that implements the specified interface or null if no match is found. The interface name must be fully qualified.

- ◆ `of(String interfacename)`

Returns a proxy to a facet that implements the specified interface, adding one automatically if no match is found. If the interface name starts with an "I", the class without the "I" prefix is used as the default facet implementation. The interface name must be fully qualified.

- ◆ `getPrimary()`

Returns a proxy to the primary object.

- ◆ `getFacets()`

Returns an array of proxies to the primary object's facets.

Example

The [Aggregation1A and Aggregation1B Examples](#) demonstrates facets by adding an Account facet to an Employee and later accessing the facet from a remote program

Two additional static helper methods in Facets that simplify facet manipulation follow:

◆ `get(Object object, Class type)`

Returns the objects when the specified object is an instance of the specified interface. Otherwise, returns a proxy to the facet when the specified object has a facet that is an instance of the specified interface. Returns null when neither rule applies.

◆ `of(Object object, Class type)`

Returns the object when the specified object is an instance of the specified interface. Otherwise, returns a proxy to the facet when the specified object has a facet that is an instance of the specified interface. Adds and returns a proxy to a facet that implements the specified type when neither rule applies.

Provide static `get()` and `of()` methods to further simplify access to facets. For example, assuming that the name of the facet interface is `IRepairHistory`, add static `get()` and `of()` helpers methods to `RepairHistory`. For example:

```
static public IRepairHistory get( Object object )
{
    return (IRepairHistory) com.objectspace.voyager.Facets.get( object, IRepairHistory.class );
}
```

```
static public IRepairHistory of( Object object ) throws ClassCastException
{
    return (IRepairHistory) com.objectspace.voyager.Facets.of( object, IRepairHistory.class );
}
```

These methods then allow you to write code, an example of which follows:

```
// return the car's repair history facet or null if it does not have one
IRepairHistory history1 = RepairHistory.get( car1 );
```

```
// return the car's repair history facet, adding one if it does not already exist
IRepairHistory history2 = RepairHistory.of( car2 );
```

Example

The [Aggregation2A and Aggregation2B Examples](#) demonstrates use of the `of()` and `get()` methods to add and access a Security facet on an Employee.

Selecting a Facet Implementation

A facet implementation varies based on the class of primary object. For example, `BonusPlan.of()` may need to attach a different kind of bonus plan facet to a `Programmer` than to a `Manager`. Voyager uses a simple scheme for selecting the class of facet that is added during an `of(object)` operation.

Assuming that the class of object is `MyClass`, `MyFacet.of(object)` attempts to attach a facet that implements `IMyFacet` and is called `xxxMyFacet`, where the search starts with `xxx="MyClass"` and moves up `MyClass`'s superclass chain to `xxx="Object"`. Classes that match the name without implementing `IMyFacet` are ignored. If the head of the superclass chain is reached and there is no match for `ObjectMyFacet`, a last chance match is attempted using `xxx=""`.

During each search cycle, the candidate class is first looked for in the package of `MyClass` and then in the package of `MyFacet`.

For example, assume that `company.Programmer` extends `company.Employee` which in turn extends `java.lang.Object`. Assume also that the full path of `BonusPlan` is `incentive.BonusPlan`. If the statement `BonusPlan.of(object)` is executed where `object` is an instance of `Programmer`, the search process picks the first class in the following series that exists and implements `IBonusPlan`:

1. `company.ProgrammerBonusPlan`
2. `incentive.ProgrammerBonusPlan`
3. `company.EmployeeBonusPlan`
4. `incentive.EmployeeBonusPlan`
5. `company.ObjectBonusPlan`
6. `incentive.ObjectBonusPlan`
7. `company.BonusPlan`
8. `incentive.BonusPlan`

After the class is selected, the facet is instantiated using the default constructor.

Example

The [Aggregation3 Example](#) illustrates facet selection.

Packaging Facets

Use the following guidelines when packaging facet classes:

- ◆ Begin the name of your facet interface with “I”.
- ◆ Create a class whose name corresponds to the interface name and omits the “I” prefix, and populate it with the `of()` and `get()` static helper methods.
- ◆ Place the default facet implementation in the same package as the facet interface, and prefix its name with the name of the most general class to which the facet applies. For example, if the default facet applies to all objects, use the `Object` prefix.
- ◆ Place specialized facet implementations into the same package as their associated class.

Creating Facet-Aware Classes

To make a class facet-aware, implement `com.objectspace.lib.facets.IFacet`, which declares the following method:

- ◆ `isTransient()` - If this method returns true, override the regular rule for garbage collection of facets and reclaim the facet immediately when there are no more references to it. Choose this feature when a facet is stateless and does not need to be associated with the primary object after its work is complete. The `Voyager Mobility` facet is an example of a transient facet. See the ["Mobility and Agents"](#) chapter for more information.

In addition, any class that implements `IFacet` can provide a constructor that takes a single `IFacets` parameter. If provided, this constructor is invoked instead of the default constructor whenever an instance of the class is added as a facet. The `IFacets` argument is set to the `Facets` instance of the primary object.

Example

The [Aggregation4 Example](#) illustrates facet-aware classes by defining a transient `BonusPlan` facet for a `Manager`.

Advanced Messaging

4

You can send synchronous messages in Voyager using regular Java syntax. However, many applications need greater flexibility, so Voyager provides a message abstraction layer that supports more sophisticated messaging features.

In this chapter, you will learn to:

- ◆ invoke messages dynamically
- ◆ retrieve remote results by reference
- ◆ use multicast and publish/subscribe

Invoking Messages Dynamically

You can dynamically invoke messages either synchronously or asynchronously.

Synchronous Messages

By default, Voyager messages are synchronous. When a caller sends a synchronous message, the caller blocks until the message completes and the return value, if any, is received. For example, the following line of code sends a synchronous `quote()` message to an instance of `Stockmarket`:

```
int price = market.buy( 42, "SUN" );
```

You can send a synchronous message dynamically using the static `Sync.invoke()` method, which returns a `Result` object when the message has completed. You can then query the `Result` object to get the return value/exception. To send a synchronous message, call `Sync.invoke()`, passing the following parameters:

- ◆ target object
- ◆ name of the method you want to call on the target object
- ◆ parameters to the dynamically invoked method in an object array

For example, the following line of code uses `Sync` to dynamically invoke a `buy()` message on an instance of `Stockmarket`:

```
Result result = Sync.invoke( market, "buy", new Object[] { new Integer( 42 ), "SUN" } );  
int price = result.readInt();
```

Primitive arguments must be sent as their `Object` equivalents.

In most cases, the simple name of the method suffices. However, if there is more than one method with the same name in the target object, the method name must be specified with argument types using the syntax `method(type1, type2)`. Spaces in the signature are ignored, and the return type must not be specified. A version of the previous example that uses the longer version of the signature follows:

```
Result result = Sync.invoke( market, "buy(int, java.lang.String)", new Object[] { new Integer( 42 ), "SUN" } );  
int price = result.readInt();
```

You can query a Result object using the following methods. In the case of synchronous methods, the reply value is always available by the time these methods are called. Future messages allow the methods to be called before the reply value is received.

- ◆ `isAvailable()`

Returns true if the Result received its return value.

- ◆ `readXXX()`, where XXX = Boolean, Byte, Char, Short, Int, Long, Float, Double, Object

Returns the value of Result, blocking until either the value is received or the timeout period of Result elapses. If the value is not received within the timeout period, a `TimeoutException` is thrown. See the ["Future Messages"](#) section on page [Future Messages](#) for information about timeouts. The timeout countdown starts when `readXXX()` is called, not when the message is actually sent. If a remote exception occurs during a future message invocation and you attempt to call `readXXX()` on Result, the exception is automatically rethrown. See the ["Basics"](#) chapter for information about exceptions.

- ◆ `isException()`

Waits for a reply and then returns true if Result contains an exception.

- ◆ `getException()`

Waits for a reply and then returns the exception contained in Result or null when no exception occurred.

Example

The [Message1 Example](#) demonstrates invoking a synchronous instance method and static method using Voyager's dynamic invocation feature.

One-Way Messages

A one-way message does not return a result. When a caller sends a one-way message, the caller does not block while the message completes, so sending a one-way message is fast. You can send a one-way message dynamically using `OneWay`, which performs "fire-and-forget" messaging.

To send a one-way message dynamically, call the static `OneWay.invoke()` method, passing the following parameters:

- ◆ target object
- ◆ name of the method you want to call on the target object

- ◆ parameters to the dynamically invoked method in an object array

For example, the following line of code uses `OneWay` to dynamically invoke a one-way `buy()` message on an instance of `Stockmarket`:

```
Result result = OneWay.invoke( market, "buy", new Object[] { new Integer( 42 ), "SUN" } );
```

The `Result` never holds a value, and only holds an exception when an error occurs during the client-side transmission of the message. For example, if the client cannot contact the remote server during the message send, the result object holds the resultant `IOException`.

Example

The [Message2 Example](#) demonstrates sending a one-way message.

Future Messages

A future message immediately returns a `Result` object, which is a placeholder to the return value. When a caller sends a future message, the caller does not block while the message completes. You can use `Result` to retrieve the return value at any time by polling, blocking, or waiting for a callback.

To send a future message, call the static `Future.invoke()` method, passing the following parameters:

- ◆ target object
- ◆ name of the method you want to call on the target object
- ◆ parameters to the dynamically invoked method in an object array

For example, the following code uses `Future` to dynamically invoke a `quote()` message on a `Stockmarket` object and then reads the return value at a later time.

```
Result result = Future.invoke( market, "quote", new Object[] { "SUN" } );  
// perform other operations here  
int price = result.readInt(); // block for price, if necessary
```

Example

The [Message3 Example](#) demonstrates sending a future message and reading the return value with a blocking call. This example also demonstrates blocking reads when the placeholder result of the future invocation is a thrown exception.

You can be notified when a future return value arrives through the standard Java event/listener mechanism. When a return value arrives, Result sends `resultReceived()` with a `ResultEvent` object to every `ResultListener` that either was specified in the full version of `Future.invoke()` or was added to the Result object after the message was sent.

Example

The [Message4 Example](#) demonstrates receiving an event notification of the arrival of the return value to a future invocation

More than one thread can invoke `readObject()` on a Result. When Result receives the return value, all blocked threads are awakened and receive that value.

Example

The [Message5 Example](#) demonstrates Voyager's ability for multiple threads to block while waiting for the return value to a single future invocation

By default, Voyager messages are synchronous and never time out. However, you can set a timeout for a future message by using the full version of `Future.invoke()`. For example, the following line of code creates a Result with a timeout period of 10,000 milliseconds:

```
Result result = Future.invoke( market, "quote", newObject[] { "SUN" }, false, 10000, null );
```

The timeout period does not begin until Result is read.

Voyager also allows you to change the timeout value for a Result generated by a future message. Use the following Result methods to work with timeouts:

◆ `setTimeout(long timeout)`

Changes the timeout value for a Result. When Result is read, the timeout period begins. Reads that take longer to complete than the specified timeout period cause a `TimeoutException` to be thrown. See the ["Basics"](#) chapter for information about exceptions.

◆ `getTimeout()`

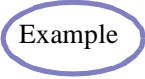
Returns the current timeout value for a Result. The default value, zero, indicates the Result never times out.

Example

The [Message6 Example](#) demonstrates Voyager's support of method invocations that time out

Retrieving Remote Results by Reference

By default, `Future.invoke()` and `Sync.invoke()` return a copy of a remote method result. If a result is large, undesirable network traffic occurs. With Voyager, you can tell `Future` or `Sync` to return a proxy to a result instead, thereby greatly reducing network traffic. If the result is not serializable, returning a proxy eliminates the need for serialization and allows the method to be invoked successfully. As expected, a proxy to a result keeps the remote result alive. To request that `Future` or `Sync` return a proxy to a result, use the full version of `invoke()` and set the `returnProxy` parameter to `true`.



Example

The [Message7 Example](#) demonstrates Voyager's support for remote method invocations that return results by reference.

Using Multicast and Publish/Subscribe

Distributed systems require features for communicating with groups of objects. For example:

- ◆ Stock quote systems use a distributed event feature to send stock price events to customers around the world.
- ◆ Voting systems use a distributed messaging feature (multicast) to poll voters around the world for their views on a particular matter.
- ◆ News services use a distributed publish/subscribe feature to send news events only to readers who are interested in the broadcast topic.

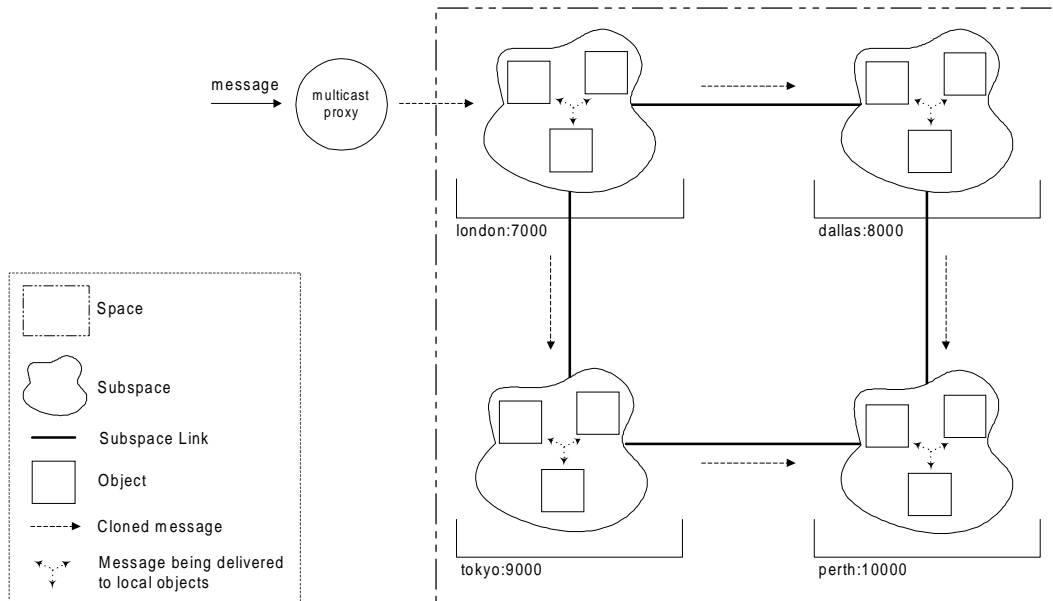
Most traditional systems use a single repeater object to replicate a message or event to each object in the target group. This approach is appropriate when the number of objects in the target group is small, but does not scale well when large numbers of objects are involved. Voyager uses scalable architecture for message/event replication called *Space*.

Understanding the Space Architecture

A Space is a distributed container that can span VMs. A Subspace is a container that cannot span VMs. A Space is created by linking together one or more Subspaces, and its contents are the union of its linked Subspaces.

A message/event sent via a multicast proxy into a Subspace is cloned to each of its neighboring Subspaces before being delivered to every object in the local Subspace, resulting in a rapid, parallel fan-out of the message to every object in the Space. As the message propagates, it leaves behind a marker unique to that message that is remembered by the Subspace for a period of five minutes. If a clone of that message re-enters the Subspace, the clone detects the marker and self-destructs. The marker allows you to connect Subspaces to form arbitrary topologies without the possibility of multiple message delivery. The more interconnected the Subspaces are, the more fault-tolerant they become in the face of individual network failures.

The following diagram illustrates sending a message to a Subspace in a Space.



Creating and Populating a Space

To create a logical Space and populate it with objects, follow these steps:

1. Construct one or more Subspace objects.

Each Subspace can reside anywhere in the network, allowing a single Space to span multiple programs.

2. Use the `subspace1.connect(subspace2)` method to connect the Subspaces in a logical Space.

Connection is symmetric; that is, if you connect `subspace1` to `subspace2`, you need not connect `subspace2` to `subspace1`. If you do, the connection will be ignored.

3. Use the `subspace1.add(object)` method to add one or more objects to each Subspace.

You can add different types of objects, including proxies, into a Subspace.

Note: Steps 2 and 3 can be done in any sequence.

You can manipulate Subspaces using additional methods defined in Subspace, including:

- ◆ `disconnect(ISubspace subspace)`
Disconnects two Subspaces. Like the `connect()` method, `disconnect()` is symmetric.
- ◆ `getNeighbors()`
Returns an array of proxies to all neighboring Subspaces.
- ◆ `isNeighbor(ISubspace subspace)`
Returns true when the specified Subspace is a neighboring Subspace.
- ◆ `remove(Object object)`
Removes the specified object from a Subspace.
- ◆ `getContents()`
Returns an array of all objects in a Subspace.
- ◆ `contains(Object object)`
Returns true when the specified object is in the Subspace.

Nested Spaces

You can nest Spaces by adding a proxy to a Subspace as an element of another Subspace. Operations on the containing Space, such as multicasting and publish/subscribe, are propagated automatically to the contained Spaces, allowing you to group smaller Spaces into a single logical Space.

Example

The [Space1 Example](#) demonstrates creating and populating a distributed Space.

Multicasting

You can multicast a Java message to a group of objects in a Space using either of two methods in Subspace:

- ◆ `multicast(String signature, Object[] args, String classname)`
Sends a one-way message to every object in the Space that is an instance of the specified class or interface.
- ◆ `getMulticastProxy(String classname)`

Returns a multicast proxy that is type-compatible with the specified class or interface. Messages sent to this proxy are multicast to every object in the Space that is an instance of the specified class or interface. Multicast messages return false, '0', 0, or null depending on the return type. You can create any number of multicast proxies with different types to the same logical Space, even to the same Subspace within a Space.

Multicast messages are always automatically propagated to nested Subspaces.

Example

The [Space2 Example](#) demonstrates typesafe multicasting of messages and JavaBeans events to objects in a Space.

Publishing and Subscribing Events

To publish an event associated with a topic to every object that implements `PublishedEventListener` in a Space, use `Subspace.publish(EventObject event, Topic topic)`.

`PublishedEventListener` defines a single method `publishedEvent(event, topic)` that receives every published event in the Space. The listener must handle the event in the appropriate manner.

A topic is specified hierarchically with fields separated by periods, like `sports.bulls` and `books.fiction.mystery`. The asterisk (*) wild card matches the next field, and the left angle bracket (<) matches all remaining fields. For example, “`games.soccer.goals`” matches “`games.soccer.*`”, “`games.*.goals`”, and “`games.<`”. Both publishers and subscribers can use wildcards to match against a range of topics.

An object can subscribe to events in three ways:

- ◆ An object can implement `PublishedEventListener` and add itself to a Space. It then receives every event that is published to the Space and must perform additional filtering and processing as necessary.
- ◆ An object can use an instance of `Subscriber` to listen to the Space on its behalf and perform event filtering/forwarding. A `Subscriber` implements `PublishedEventListener` and has methods for subscribing/unsubscribing to topics. It also contains a reference to another `PublishedEventListener`. When a `Subscriber` is added to a Space, it forwards any published event that matches a topic to its associated `PublishedEventListener`. The `PublishedEventListener` does not have to be in the same program as the `Subscriber`. for example, to perform server-side filtering, set the `Subscriber`'s `PublishedEventListener` to a

local intermediary object that performs additional processing and then forwards the event, if appropriate, to its final remote destination.

- ◆ An object can use dynamic aggregation, add a Subscriber facet, and then add the facet to the Space. The Subscriber facet forwards all selected events to the primary object, which must implement `PublishedEventListener`.

Published events are always automatically propagated to nested Subspaces.

Example

The [Space3 Example](#) demonstrates publishing events to subscribers in a Space.

Administering a Space

By default, a Subspace does nothing when its objects and neighbors are disconnected or killed. You can instruct a Subspace to purge itself of disconnected or dead objects and neighbors by using the following Subspace methods:

- ◆ `setPurgePolicy(byte policy)`

Sets a Subspace's purge policy. Four policies are available:

- `Subspace.DIED` removes proxies to objects and neighboring Subspaces that have been garbage-collected. A Subspace knows an object is dead when an `ObjectNotFoundException` is thrown as a result of sending a message to the object.
- `Subspace.DISCONNECTED` removes proxies to objects and neighboring Subspaces that are not reachable. A Subspace knows an object is disconnected when an `IOException` is thrown as a result of sending a message to the object
- `Subspace.ALL` removes proxies to dead and disconnected objects and neighbors.
- `Subspace.NONE`, the default policy, ignores dead and disconnected proxies.

- ◆ `getPurgePolicy()`

Returns the purge policy assigned to a Subspace.

- ◆ `purge(byte policy)`

Forces a Subspace to be purged immediately.

A Subspace automatically purges itself according to its purge policy every 5 minutes or 10,000 messages, whichever comes first.

Mobility and Agents

5

Mobility allows you to move objects that exchange large numbers of messages closer to each other to reduce network traffic and increase throughput. A local message is often at least 1,000 times faster than its remote equivalent. This technique is known as *locality optimization*. In addition, a program can move objects into a mobile device so that the program can remain with the device after the device has been disconnected from the network.

In addition to standard mobility support, Voyager also supports *mobile autonomous agents*, which are objects that move themselves in order to achieve their goals.

In this chapter, you will learn to:

- ◆ move an object to a new location
- ◆ obtain move notification
- ◆ understand uses for mobile agents
- ◆ create mobile agents

Moving an Object to a New Location

To move an object to new location, use `Mobility.of()` to obtain the object's mobility facet (see the "[Dynamic Aggregation™](#)" chapter) and then use the methods defined in `IMobility`:

- ◆ `moveTo(String url)`

Moves to the program with the specified URL.

- ◆ `moveTo(Object object)`

Moves to the program that contains the specified object. The object is usually specified as a proxy.

For example, the following code creates a `StockMarket` at `//dallas:8000` and then moves it to `//tokyo:9000`:

```
IStockMarket market = (IStockMarket) Factory.create( "StockMarket", "//dallas:8000" );
market.news( "at first location" ); // send message to initial location
IMobility mobility = Mobility.of( market ); // obtain mobility facet
mobility.moveTo( "//tokyo:9000" ); // move the object to a new location
// the last two lines could be written as Mobility.of( market ).moveTo( "//tokyo:9000" )
market.news( "at second location" ); // message is delivered to new location
```

The `moveTo()` method causes the following sequence of events to occur:

1. Any messages that the object is currently processing are allowed to complete and any new messages that arrive at the object are suspended. The code can only detect method calls that are synchronized, so do not attempt to move an object that might be executing non-synchronized methods.
2. The object and all of its non-transient parts are copied to the new location using Java serialization, ignoring pass-by-reference tags like `java.rmi.Remote` and `com.objectspace.voyager.IRemote`. An exception is thrown when any part of the object is not serializable or when a network error occurs. To avoid copying a particular part as an object, store a proxy to the part instead.
3. The new addresses of the object and all of its non-transient parts are cached at the old location.
4. The old object is destroyed.
5. Suspended messages sent to the old object are resumed.

6. When a message sent via a proxy arrives at the old address of a moved object, a special exception containing the object's new address is thrown back to the stale proxy. The proxy traps this exception, rebinds to the new address, and then resends the message to the updated address. If the program at the old location crashes before a stale proxy is updated, the stale proxy is unable to successfully rebind and a message sent via the proxy generates an `ObjectNotFoundException`.
7. The `moveTo()` returns after the object is successfully moved or when a mobility exception occurs. If an exception occurs, the old object is restored to its original condition, suspended messages are resumed, and the exception is rethrown wrapped in a `MobilityException`.

The rules for garbage collection are not affected by mobility. A moved object is reclaimed when there are no more local or remote references to it. The new addresses cached at the old location are not treated as references by the garbage collection system.

It is unsafe to move an object when local Java references point to it from outside the context of Voyager or when the object has one or more threads not associated with a remote message.

Example

The [Mobility1 Example](#) creates a Drone object and then moves it between programs.

Obtaining Move Notification

Sometimes an object needs to know that it is about to move or has just been moved. For example, a persistent mobile object may need to remove itself from the origin's persistent store and add itself to the destination's persistent store. Voyager provides this capability through the `IMobile` interface. If an object or any of its parts implements the `IMobile` interface, they will receive callbacks during a move in the following order:

1. `preDeparture(String source, String destination)`

This method executes on the original object at the source. If the method throws a `MobilityException`, the move aborts and no more `IMobile` callbacks occur.

2. `preArrival()`

This method is executed on the copy of the object at the destination. If the method throws a `MobilityException`, the move is aborted and no more `IMobile` callbacks occur.

3. `postArrival()`

At this point, the copy of the object becomes the real object, the object at the source becomes the stale object, and the move is deemed successful and cannot be aborted. This method executes on the copy of object at the destination immediately prior to the user-supplied callback. It is typically defined to perform activities such as adding the new object into persistent storage.

4. `postDeparture()`

This method executes on the original stale object at the source. It is typically defined to perform activities such as removing the stale object from persistence. Messages sent to the stale object via a proxy are redirected to the new object, so `postDeparture()` should not use proxies to the original object or any of its facets. Because the user-supplied callback on the new object is executed using a fresh thread, it is possible for this `postDeparture()` to be executing concurrently with the user-supplied callback.

Example

The [Mobility2 Example](#) creates a mobility-aware `Drone2` object and then moves it between programs. Also see the "[Understanding the Uses for Mobile Agents](#)" section starting on page [Understanding the Uses for Mobile Agents](#) for another example.

Understanding the Uses for Mobile Agents

A mobile autonomous agent is an object that moves itself around the network in order to achieve its goals. You can use mobile agents as follows:

- ◆ If a task must be performed independently of the computer that launches the task, a mobile agent can be created to perform this task. Once constructed, the agent can move into the network and complete the task in a remote program.
- ◆ If a program needs to send a large number of messages to objects in remote programs, an agent can be constructed to visit each program in turn and send the messages locally. Local messages are often between 1,000 and 100,000 times faster than remote messages.
- ◆ If you want to partition your programs to execute in parallel, you can distribute the processing to several agents, which migrate to remote programs and communicate with each other to achieve the final goal.
- ◆ If periodic monitoring of a remote object is required, creating an agent that moves to the remote object and monitors it locally is more efficient than monitoring the object across the network.
- ◆ If a series of operations must be performed inside a consumer device that is only occasionally connected to a network, such as a Java phone or Java pager, then an agent can move into the device, perform its task, and move back into the network only when necessary.

It is important to avoid “force-fitting” agent technology into a program. Voyager’s remote messages are adequate for many applications, and simple object mobility is often enough to close the gap between two objects communicating on a network. However, as you become familiar with the power of agents, you may find many ways to agent-enhance your current and future programs.

Creating Mobile Agents

To make an object a mobile autonomous agent, use `Agent.of()` to obtain the object's agent facet (see the "[Dynamic Aggregation™](#)" chapter) and then use the methods defined in `IAgent`:

- ◆ `moveTo(String url, String callback [, Object[] args])`

Moves to the program with the specified URL and then restarts by executing a oneway callback with optional arguments. A `MobilityException` is thrown when the callback method is not found or is not public.

- ◆ `moveTo(Object object, String callback [, Object[] args])`

Moves to the program containing the specified object and then restarts by executing a oneway callback with a proxy to the object as the first argument and the optional arguments as the remaining arguments. A `MobilityException` is thrown when the callback method is not found or is not public.

- ◆ `setAutonomous(boolean flag)`

If the flag is true, become autonomous. An autonomous agent is not reclaimed by the garbage collector even if there are no more local or remote references to it. An agent is initially autonomous by default, and typically executes `setAutonomous(false)` when it has achieved its goal and wishes to be garbage collected.

- ◆ `isAutonomous()`

Return true if this agent is autonomous.

- ◆ `getHome()`

Return the home of this agent, which is defined to be the URL of the agent when its agent facet was first accessed.

For example, an object can move itself to `//dallas:8000` and restart using `atDallas()` by executing:

```
Agent.of( this ).moveTo( "//dallas:8000", "atDallas" );
```

A successful call to `moveTo()` conceptually causes the thread of control to stop in the agent before it moves and to resume from the callback method in the agent after it moves. Therefore, only exception-handling code should follow a `moveTo()`.

Example

The [Agents1 Example](#) constructs a Trader agent that works on the stockmarket from a remote location and then moves itself to the stockmarket to work locally.

Code Mobility

There are three ways to make an agent's class files available to a host to which the agent may be traveling:

- ◆ Pre-install all the class files in the remote host's CLASSPATH. In a large system, this method requires dealing with maintenance issues.
- ◆ Keep all system classes in a single repository. This method requires that all remote hosts bootstrap the location of the resource repository at startup. See the "[Loading Classes](#)" and "[Serving Classes](#)" sections in the "[Basics](#)" chapter for additional information on resource loaders. In most cases, this option is preferred when managing a homogeneous system.
- ◆ Have an agent register a resource loader before it arrives. This method allows an agent to carry its class files and resources as it moves through the network.

The Voyager class library ships with two `IResourceLoader` implementations:

- ◆ `URLResourceLoader`

This resource loader takes a `java.net.URL` class on the constructor. The URL instance may reference the host that the agent is being launched from or a simple repository. For example, `http://classes.home.com:8000/`.

- ◆ `ArchiveResourceLoader`

This resource loader is similar to the `URLResourceLoader`, except the URL expected on the constructor must point to a `.jar` or `.zip` file. For example, `http://classes.home.com:/8000/jars/networkagent.jar`. The `ArchiveResourceLoader` does not retrieve the remote jar until the jar itself or its resources holder is requested. This reduces network traffic in case an instance of this resource loader is already installed on the remote host.

To set and retrieve an agent's resource loader, invoke the following methods on IAgent:

- ◆ `setResourceLoader(IResourceLoader resourceLoader)`

Indicates to the agent to use the given IResourceLoader instance when loading its resources.

- ◆ `getResourceLoader()`

Returns the registered IResourceLoader instance used by the agent. If additional resources are stored in a resource loader other than class files, such as certificates or sound files, the agent can access them directly using the IResourceLoader interface.

When an agent leaves a host, it always removes the resource loader it installed before it arrived. The VoyagerClassLoader maintains a reference count on each IResourceLoader instance installed at a given priority. When the count reaches zero, the given resource loader is removed from the system.

Naming Service 6

Voyager's naming service allows you to bind names to an object and lookup objects by name.

In this chapter, you will learn to:

- ◆ use a namespace
- ◆ work with federated directory services
- ◆ use the default naming service
- ◆ use JNDI
- ◆ use `PersistentDirectory`

Using a Namespace

A naming service allows names to be associated with an object for later lookup. You can use many different implementations of naming services, including:

- ◆ Voyager federated directory service
- ◆ CORBA naming service
- ◆ JNDI
- ◆ Microsoft Active Directory
- ◆ RMI registry

When a naming service is used to bind a name to an object, it adds a unique prefix so that the type of naming service can later be determined directly from the name. For example, the Voyager federated directory service uses the prefix `vdir:`, and the CORBA naming service uses the prefix `IOR:`.

The Voyager Namespace class takes advantage of these prefix codes to provide a single, simple interface that unifies access to one or more of these naming services. New naming services can be dynamically plugged into Namespace.

Each of the following static methods uses the name's prefix to determine which underlying naming service to access.

- ◆ `lookup(String name)`
Returns a proxy to the object associated with the specified name, or null when no such object is found.
- ◆ `bind(String name, Object object)`
Associates the specified name with the object, or throws an exception when the name already has an association.
- ◆ `rebind(String name, Object object)`
Associates the specified name with the object, replacing any previous association when present.
- ◆ `unbind(String name)`
Disassociates the specified name.

The default naming service is the Voyager federated directory service. If a prefix is missing from a name, it is assumed to be `vdir::`. Voyager automatically installs several naming services, as shown in the following table:

Voyager ORB	Voyager ORB Professional
Voyager federated directory service	CORBA naming service
CORA IOR resolution	JNDI
RMI registry	

For an example of accessing the CORBA IOR resolution service via the Namespace class, see the ["CORBA"](#) chapter.

Working with Federated Directory Services

The Voyager federated directory service allows you to register an object in a distributed hierarchical directory structure. You can associate objects with path names comprised of simple strings separated by slashes, such as `fruit/citrus/lemon` or `animal/mammal/cat`. The building block of the directory service is a Directory, which has the following interface:

- ◆ `put(String key, Object value)`

Associates a key with a value. If key is a simple string, associates it with the specified value in the local directory. If key is a path, looks up the Directory associated with the head of the path name and then forwards the `put()` message with the remaining tail of the path name. Returns the value previously associated with the key or null when there was none.

- ◆ `get(String key)`

Returns the value associated with a particular key. If key is a simple string, return its associated value in the local directory or null when there is none. If key is a path, looks up the Directory associated with the head of the path name and then forwards the `get()` message with the remaining tail of the path name.

- ◆ `remove(String key)`

Removes the directory entry with the specified key. If key is a simple string, removes its entry from the local directory. If key is a path, looks up the Directory associated with the head of the path name and then forwards the `remove()` message with the remaining tail of the path name. Returns the value that was associated with the key or null when there was none.

◆ `getValues()`

Returns an array of the values in the local directory.

◆ `getKeys()`

Returns an array of the keys in the local directory.

◆ `clear()`

Removes every entry from the local directory. Removing the entries has no effect on the directories that the local directory used to reference.

◆ `size()`

Returns the number of keys in the local Directory.

To create a simple directory of local objects, create a Directory object and send it the `put()` message with a string key and a local object.

```
Directory symbols = new Directory();
symbols.put( "CA", "calcium" );
symbols.put( "AU", "gold" );
// symbols.get( "CA" ) would return "calcium"
```

To create a chained directory structure, a Directory that refers to another Directory, send `put()` to a Directory object with another directory or a proxy to a remote Directory as the second parameter.

```
Directory root = new Directory();
root.put( "symbols", symbols ); // associate "symbols" with the symbols directory
// root.get( "symbols/CA" ) would return "calcium"
```

Because Directory implements `IRemote`, you can pass a local directory as a parameter to a remote directory and it is automatically sent as a proxy.

Example

The [Naming1 Example](#) sets up a simple federated directory service.

Using the Default Name Service

The Voyager federated directory system is the default naming service employed by Namespace. You can use Namespace to bind, rebind, and unbind remote objects without directly accessing a Directory object.

When a Voyager program starts up, it automatically exports a single Directory object for use by Namespace. When you execute a `lookup()` operation on Namespace and the name has no prefix, Namespace interprets the name as an URL, obtains a proxy to the Namespace Directory in the corresponding program, and then executes a remote `get()` on the Directory with the remainder of the URL as the key. For example:

Namespace Format	Directory Equivalent
<code>Namespace.lookup("Fred")</code>	<code><Directory @ local program>.get("Fred")</code>
<code>Namespace.lookup("8000/Fred")</code>	<code><Directory @ //localhost:8000>.get("Fred")</code>
<code>Namespace.lookup("//dallas:8000/Fred")</code>	<code><Directory @ //dallas:8000>.get("Fred")</code>
<code>Namespace.lookup("//dallas:8000/Fred/Bloggs")</code>	<code><Directory @ //dallas:8000>.get("Fred/Bloggs")</code>

`bind()`, `rebind()`, and `unbind()` are processed in a similar manner:

Namespace Format	Directory Equivalent
<code>Namespace.bind("Fred", object)</code>	<code><Directory @ local program>.put("Fred", object)</code>
<code>Namespace.unbind("8000/Fred/Bloggs")</code>	<code><Directory @ //localhost:8000>.remove("Fred/Bloggs")</code>

For convenience, `Factory.create()` is integrated with Namespace. If the host location is followed by a name, this name is used automatically to perform a `bind()` to the Namespace at the host location. For example, instead of typing the following:

```
IStockmarket market = (IStockmarket) Factory.create( "Stockmarket", "//dallas:8000" );
Namespace.bind( "//dallas:8000/NASDAQ", market );
```

You can type:

```
IStockmarket market = (IStockmarket) Factory.create( "Stockmarket", "//dallas:8000/NASDAQ" );
```

Example

The [Naming2 Example](#) illustrates the default naming service.

Using JNDI



Certain client processes may need to access named objects using the Java Naming and Directory Interface (JNDI). Voyager implements JNDI as a wrapper around its Federated Directory Service.

To create InitialContexts, you can use either a local Properties object or set Java system properties.

```
Properties environment = new Properties();
environment.put( "java.naming.factory.initial",
    "com.objectspace.voyager.jndi.spi.VoyagerContextFactory" );
environment.put( "java.naming.provider.url", "//dallas:8000/dir" );
Context context = new InitialContext( environment );
```

This code sets the JNDI initial context factory to be Voyager's and requests the initial context that represents the directory at //dallas:8000/dir. You *must* set the initial context factory as shown. There are other optional properties that can be set. See the JNDI documentation available from Sun for more information.

All of Voyager's Namespace functionality can be accessed through a valid JNDI context. Refer to the following examples, in which ctx is assumed to be a valid JNDI context that represents the directory at //dallas:8000/dir:

Namespace Format	JNDI Equivalent
Namespace.lookup("//dallas:8000/dir/Fred");	ctx.lookup("Fred");
Namespace.bind("//dallas:8000/dir/Bloggs", object);	ctx.bind("Bloggs", object);
Namespace.unbind("//dallas:8000/dir/Bloggs");	ctx.unbind("Bloggs");

Example

The [Naming3 Example](#) illustrates the JNDI naming service.

Using PersistentDirectory



PersistentDirectory is a specialization of Directory. A PersistentDirectory stores its contents in a file in the local file system. To create a directory of persistent objects, create a PersistentDirectory passing a filename to the constructor.

PersistentDirectory is a specialization of Directory. A PersistentDirectory stores its contents in a file in the local file system. To create a directory of persistent objects, create a PersistentDirectory passing a filename to the constructor.

```
PersistentDirectory symbols = new PersistentDirectory( "symbols.db" );  
symbols.put( "CA", "calcium" );
```

The contents of this directory can be accessed as long as the file “symbols.db” exists in the local file system, regardless of whether the program that created these entries still exists. To create a new PersistentDirectory using an existing file, use the same constructor.

```
PersistentDirectory oldSymbols = new PersistentDirectory( "symbols.db" );  
// symbols.get( "CA" ) will still return "calcium"
```


Activation 7

By default, a message sent to an object that was in a terminated and restarted program will throw an `ObjectNotFoundException`. Voyager's activation framework allows an object to survive program restarts and to receive messages as if the program had never shut down.

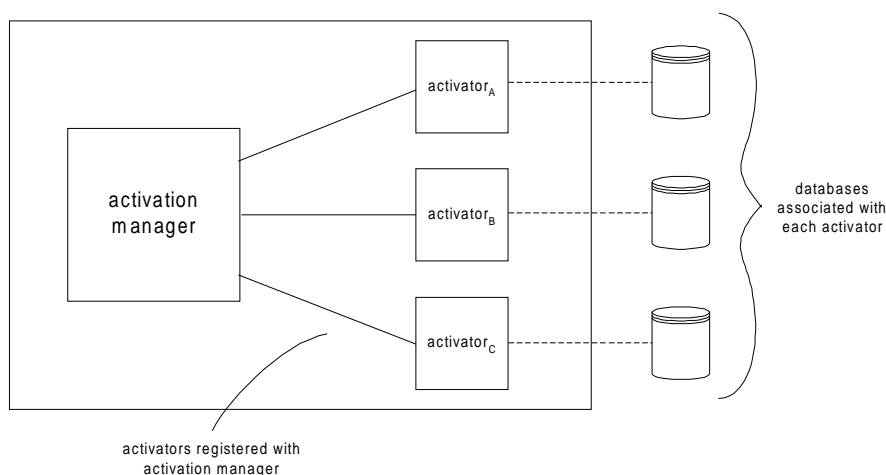
In this chapter, you will learn to:

- ◆ enable an object for activation
- ◆ activate an object
- ◆ write an activator

Enabling an Object for Activation

Every Voyager program contains a single *activation manager* that manages zero or more *activators*. Each activator is typically associated with a database and is registered using `Activation.register()` by the application program at startup.

The following diagram illustrates a typical activator setup.



An activator must implement `IActivator`, defined as follows:

- ◆ `getMemento(Proxy proxy)`

Returns a memento string, typically a database key, that can later be used to reactivate the proxy's object using `activate()`. If the activator was not designed to handle the object, returns null.

- ◆ `activate(String memento)`

Loads the object associated with the memento, and returns a proxy to the activated object. The memento was generated by a previous call to `getMemento()`.

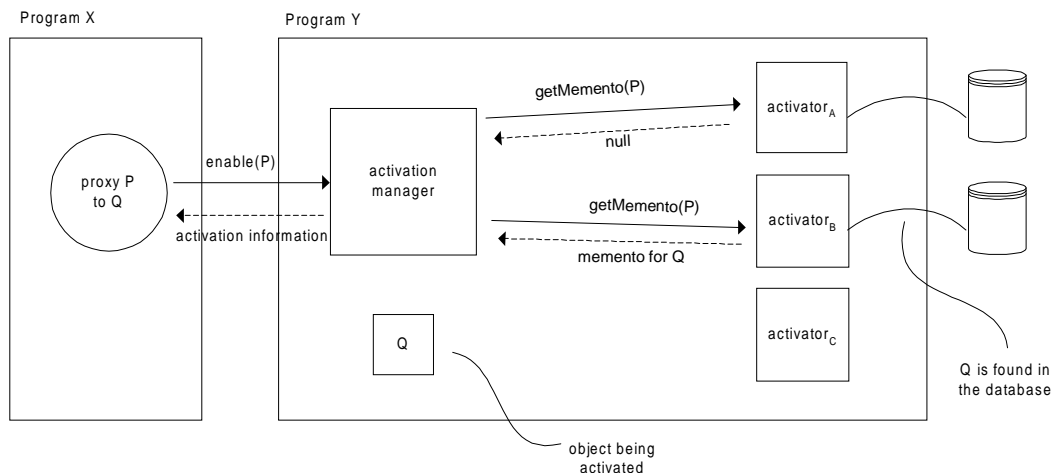
An activator is typically written by an applications programmer to activate a particular class of object, although more powerful generic activators can certainly be built. A carefully designed activator can be reused by other applications.

An object's class does not have to be modified in any way to take advantage of the activation framework. To enable an object for activation, pass the object or proxy to `Activation.enable()`, which causes the following to occur:

- ◆ The activation manager in the object's program cycles through its activators, sending each `getMemento()` with a proxy to the object until one of the activators successfully returns a memento. If every activator returns null, the object could not be enabled for activation, and an `ActivationException` is thrown. If a memento is obtained, it is cached in the program, and every new proxy to the object is tagged with *activation information* including the program's URL, the memento, and the class of activator that created the memento. Such proxies are termed *activating proxies*.
- ◆ `Activation.enable()` returns an activating proxy to the object. If a proxy was passed to `Activation.enable()`, it is automatically turned into an activating proxy.

An activation-enabled object remains enabled for its lifetime, even if its program is restarted.

The following diagram illustrates the series of events that occurs when an object is enabled for activation.



Activating an Object

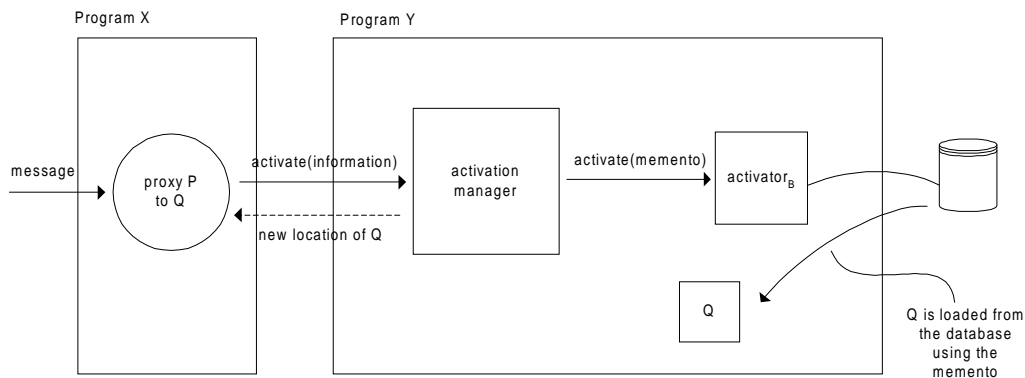
If a message is sent to a remote object whose program has been restarted, an `ObjectNotFoundException` is thrown. An activating proxy catches this exception and attempts to rebind to the object by using its activation information.

The proxy passes the activation information to the activation manager in the object's program, which in turn uses the activator class name to locate the object's activator. If an instance of the activator class is not already registered, Voyager automatically constructs one using its default constructor and registers it with the activation manager.

Once located, the activation manager sends `activate(memento)` to the object's activator. The activator loads the object into memory and returns a proxy to the newly restored object. Information in this proxy is used to complete the rebind process, and the original message is resent to the activated object.

The entire activation/rebind process is transparent to the user of an activating proxy.

The following diagram illustrates the sequence of events that occurs when an object is activated following an `ObjectNotFoundException`.



Writing an Activator

Guidelines for the implementation of each method in `IActivator` follow:

- ◆ `getMemento(Proxy proxy)`
 - Uses `Snapshot.of(proxy)` to obtain a `Snapshot` of the proxy's object.
 - Uses `Snapshot.getObject()` to obtain the object. If the activator was not designed to activate the object, returns null, which allows the activation manager to skip to the next activator
 - Locates the object in the activator's database. If the object is found, returns a string memento that will allow `activate()` to retrieve the object from the database.
 - If the object is not already in the database, an activator can choose to either automatically store the object or to throw an `ActivationException`. To store the object, either save the `Snapshot` object directly or save its parts individually.
- ◆ `activate(String memento)`
 - If a `Snapshot` of the object was stored directly into the database, loads the `Snapshot` using the memento key. If parts of the object's state were stored into the database, loads each part separately and recreates the original `Snapshot` using `Snapshot.from()`.
 - Returns the result of sending `restore()` to the `Snapshot`.

See the "[Basics](#)" chapter for more information about `Snapshots`.

Example

The [Activation1A and Activation1B Example](#) illustrates the activation framework

Security 8

Voyager includes support for the standard Java security manager system. Java applets and servlets are automatically initialized with a very restrictive security manager called `AppletSecurityManager`, so Voyager applets abide by these settings. See the "[Ultra-Light Client, Applets and Servlets](#)" chapter to learn to use applets and servlets with Voyager.

Java applications, however, have no security manager by default, so objects may perform any type of operation. Voyager includes a security manager called `VoyagerSecurityManager`, which you can install at the start of a program to restrict operations.

In this chapter, you will learn to:

- ◆ install a security manager
- ◆ identify object authority

Note: The Voyager Security product offers additional security functionality. For more information on Voyager Security, refer to the [Voyager Security Developer Guide](#).

Installing a Security Manager

You have the option of installing a security manager in a Voyager program. After it is installed, the security manager is active for the duration of the program, and it cannot be uninstalled or replaced. Each time an object attempts to execute an operation that could compromise security, the Java run-time machinery checks with the program's security manager to determine whether the operation is permitted. The following section lists legal operations by environment. If the program has no security manager, or if the security manager permits the operation, Voyager proceeds as normal. If the operation is disallowed, a run-time security exception is thrown.

The Voyager security manager distinguishes between *native* and *foreign* objects:

- ◆ Native objects are objects whose class resides in the program's CLASSPATH.
- ◆ Foreign objects are objects whose class was loaded across the network from another program.

The Voyager security manager allows native objects to perform any operation but selectively restricts foreign objects by operation.

Note: You can modify or extend the Voyager security manager behavior by extending the `VoyagerSecurityManager` class.

You can install a Voyager security manager in one of two ways:

- ◆ To start a Voyager server with a Voyager security manager, execute `voyager` with the `-s` (security) option.
- ◆ To install a Voyager security manager in a Voyager program, create a new instance of the security manager, and install it with the `System.setSecurityManager(manager)` method, where *manager* is your security manager.

Example

The [Security1 Example](#) demonstrates using Voyager's security manager to restrict operations by foreign objects.

Identifying Object Authority

The following table lists the operations allowed by the JDK SecurityManager and indicates those that VoyagerSecurityManager, which extends SecurityManager, allows an object to perform as an object in an applet, as a native object, and as a foreign object.

Operation	Object in Applet	Native Object	Foreign Object
Accept connections from any host	✓ server only	✓	✓
Connect to any host	✓ server only	✓	✓
Listen on any port		✓	✓
Perform multicast operations		✓	✓
Set factories		✓	
Manipulate threads	✓	✓	
Manipulate thread groups	✓	✓	
Execute a process		✓	
Exit the program		✓	
Access AWT event queue	✓	✓	✓
Access the system clipboard		✓	
Create windows	✓	✓	✓
Create class loader		✓	
Delete files		✓	
Read files, excluding socket file descriptors		✓	
Write files, excluding socket file descriptors		✓	
Access security APIs		✓	
Link to a dynamic library		✓	

Operation	Object in Applet	Native Object	Foreign Object
Access private/protected data and methods		✓	
Access packages		✓	✓
Define classes in packages		✓	✓
Print		✓	
Manipulate properties	✓ limited	✓	✓

The `VoyagerSecurityManager` defines a `checkMethodAccess()` method not included in its parent JDK class, `SecurityManager`. The `checkMethodAccess()` method prevents a foreign object from calling the following Voyager methods:

Class	Method
Voyager	<code>shutdown()</code> <code>addSystemListener()</code> <code>removeSystemListener()</code>
ClassManager	<code>setParentClassLoader()</code> <code>enableResourceServer()</code> <code>resetClassLoader()</code>
VoyagerClassLoader	<code>addResourceLoader()</code> <code>removeResourceLoader()</code> <code>setResourceLoadingEnabled()</code>

To change the methods disallowed by `checkMethodAccess()`, refer to the online [Voyager API Guide](#).

Ultra-Light Client, Applets and Servlets

9

Voyager supports the creation of an ultra-light client that can bind and send messages to any object in a universal namespace. This facility is useful for building fast-downloading applets or for clients that must fit into a small memory footprint. Voyager also supports regular servlets and applets.

In this chapter, you will learn to:

- ◆ implement Voyager Ultra-Light Client
- ◆ understand Voyager Ultra-Light Client limitations
- ◆ package Voyager Ultra-Light Client
- ◆ use Voyager with servlets
- ◆ use Voyager with applets

Implementing Voyager Ultra-Light Client



Voyager Ultra-Light Client is ideal for creating applets that must download fast and work with any Java-compatible browser. Voyager Ultra-Light Client uses the Namespace class to perform naming service operations. See the ["Using a Namespace"](#) section in the ["Naming Service"](#) chapter.

To import the Ultra-Light Client code into an application or applet, use the `lightclient.jar` file included in the Voyager package.

When a light client is communicating with a Voyager server, it uses the URL class loading mechanism. The Voyager server must have resource serving enabled. To enable resource serving, use the `-r` option when starting Voyager or invoke the `ClassManager.enableResourceServer()` method in the server's code.

Depending on the type of implementation, the light client may run as an application or applet.

Using Ultra-Light Client from an Application

To import the light client classes, the CLASSPATH of the importing application must contain a fully qualified path to `lightclient.jar`. There are no prerequisite method invocations required to initialize the ultra-light client code.

Using Ultra-Light Client from an Applet

To use the ultra-light client code in an applet, `lightclient.jar` must be specified in the *archive* tag of the applet. The value of the applet's *codebase* tag must contain a URL pointing to a Voyager server responsible for the class loading. An applet using the Ultra-Light Client must call `Namespace.setServerURL(Applet applet)`, with an instance of the applet as a parameter. This action allows the Ultra-Light Client code to properly initialize itself and read in the *codebase* value.

Understanding Ultra-Light Client Limitations



The only limitation of the ultra-light client is that it cannot pass objects by reference. This means that objects passed as arguments in the remote invocations must implement `java.io.Serializable` and must not implement `java.rmi.Remote` or `com.objectspace.Voyager.IRemote`; otherwise a `RuntimeException` is thrown. It also means there is no way for remote objects to message objects in an ultra-light client.

Packaging Voyager Ultra-Light Client



Depending on the implementation needs, you may consider using the following client archives for packaging:

- ◆ *lightclient.jar* contains the minimum functionality to perform simple naming operations, such as `lookup()`, `bind()`, `rebind()`, `unbind()`.
- ◆ *lightjndiclient.jar* provides the JNDI implementation along with the Namespace API.

Using Voyager with Servlets

For class loading to work correctly, Voyager must be given access to a servlet's class loader at startup. For access, start Voyager using `Voyager.startup(Object object, String address)` and pass the servlet as the first argument. The second argument can either be a port number or null when a random port is desired.

Example

The [CalcServlet Example](#) demonstrates a Voyager-enabled servlet.

Using Voyager with Applets

Like servlets, an applet must start Voyager using `Voyager.startup(Object object, String address)` and pass the applet as the first argument. The second argument can either be a port number or null when a random port is desired.

Unlike servlets, most applets are prevented from forming a network connection to any machine except their Web server, and from accepting the regular Voyager class loader that enables pluggable resource loading and dynamic proxy generation. Voyager works around these restrictions by allowing a program on the web server to act as a hub for an untrusted applet and perform messaging routing and Voyager class loading for the applet's behalf.

Use the following rules to determine how best to set up your web server and HTML file:

1. If the applet needs routing and Voyager class loading, set the applet's CODEBASE to the address of a Voyager hub. The hub can be any Voyager program whose HTTP server is enabled. To start an HTTP-enabled server from the command line, use the voyager utility with the `-r` option. To enable the internal HTTP server from within a program, invoke `ClassManager.enableResourceServer()`. Ensure that the applet's classes are accessible to the hub via its CLASSPATH and/or its optional resource loaders. See the "[Basics](#)" chapter for more information on resource loaders. For example, assuming that an HTTP-enabled hub is running on `myhost.com:8000`:

```
<APPLET CODEBASE="http://myhost.com:8000" CODE="MyApplet.class" WIDTH=300 HEIGHT=50>
</APPLET>
```

2. If the applet needs routing but not Voyager class loading, set the HTML `routerAddress` parameter to the port number of a Voyager hub on the same machine as the applet's CODEBASE. The hub can be any Voyager program, and does not have its internal HTTP server enabled. For example, assuming a hub is running on port 8000 of the applet's server:

```
<APPLET CODE="MyApplet.class" WIDTH=300 HEIGHT=50>
<PARAM name="routerAddress" value="8000">
</APPLET>
```

3. If the applet does not need routing or Voyager class loading, set the HTML `disableRouting` parameter to true. This setting is most often used for Voyager-enabled trusted applets. For example:

```
<APPLET CODE="MyApplet.class" ARCHIVE="MyApplet.jar" WIDTH=300 HEIGHT=50>
<PARAM name="disableRouting" value="true">
</APPLET>
```

If necessary, a Voyager program can manually force routing by invoking `Routing.setRouterAddress()`.

When Voyager is not used as a resource server, you must use the `pgen` utility to generate proxy classes.

Example

The [CalcApplet Example](#) demonstrates two Voyager-enabled applets.

CORBA 10

Common Object Request Broker Architecture (CORBA) is a widely supported standard that allows objects to advertise their interfaces using Interface Definition Language (IDL) and communicate across networks using a language-neutral protocol called Internet Inter-Orb Protocol (IIOP). For example, CORBA allows a C++ client running on Windows NT to communicate with a Java object located on a UNIX server.

Traditional CORBA implementations use a proxy class on the client to forward the request to the server, a skeleton class on the server to accept the request, and a helper class on each side to facilitate the IIOP encoding/decoding process. Vendors of these traditional implementations typically provide tools to automatically generate these classes from IDL.

Voyager offers the most productive implementation for building CORBA applications:

- ◆ Its universal communications architecture simultaneously supports Voyager Native Messaging (VNM), RMI, CORBA, and, shortly, DCOM, so there is no need for additional bridging products.
- ◆ The RMI and VNM modules provide distributed garbage collection and dynamic class loading for Java-centric development.
- ◆ The CORBA module supports the full IDL specification, automatically CORBA-enables objects at runtime, and generates proxies dynamically so that no stub generators, skeletons, or helper classes are required.

In this chapter, you will learn to:

- ◆ build an application
- ◆ map IDL between Java using the cgen utility
- ◆ import/export objects
- ◆ pass references between CORBA programs
- ◆ raise/catch CORBA exceptions
- ◆ use holders to support out and in out arguments
- ◆ use IDL types such as struct, union, enum, arrays, sequences, any, and typecode
- ◆ use the CORBA naming service
- ◆ understand wide-character support

Building an Application

In this example of building a simple client-server CORBA banking application, a server hosts a bank object that clients can contact in order to open an account. After an account is opened, a client can deposit and withdraw money. Any attempt to withdraw more money than is in the account causes an exception to be thrown.

The application is built using the following six steps:

1. write IDL definition files for each CORBA entity
2. generate the Java language bindings from IDL
3. write implementations of each Java interface
4. write the server program
5. write the client program
6. run the server and client programs

In this example, the IDL files, the server program, and the client program are developed in different subdirectories of `examples\corba`:

```
examples
\corba
  \common -- used to store IDL definition files and share object references
  \server -- server program
  \client -- client program
```

Step 1: Write the IDL

IDL describes the interface to an object in a language-neutral way. In addition to the familiar Java constructs like primitive data types, interfaces, and exceptions, IDL supports additional data types like structs, unions, and enums.

A list of the IDL keywords follows:

- | | | |
|------------|-------------|------------|
| ◆ any | ◆ attribute | ◆ boolean |
| ◆ case | ◆ char | ◆ const |
| ◆ context | ◆ default | ◆ double |
| ◆ enum | ◆ exception | ◆ FALSE |
| ◆ fixed | ◆ float | ◆ in |
| ◆ inout | ◆ interface | ◆ long |
| ◆ module | ◆ Object | ◆ octet |
| ◆ oneway | ◆ out | ◆ raises |
| ◆ readonly | ◆ sequence | ◆ short |
| ◆ string | ◆ struct | ◆ switch |
| ◆ TRUE | ◆ typedef | ◆ unsigned |
| ◆ union | ◆ void | ◆ wchar |
| ◆ wstring | | |

This example only uses a small subset of these keywords. The remaining keywords are covered later in this chapter.

Three entities are defined in IDL as follows:

```
// thrown when an account is overdrawn
exception OverdrawnException
{
    string message;
    long amount; // amount overdrawn
};

// defines the interface to an account
interface IAccount
{
    long deposit( in long arg1 );
    long getBalance();
    void withdraw( in long arg1 ) raises( OverdrawnException );
};

// defines the interface to a bank
interface IBank
{
    IAccount openAccount();
};
```

```
};
```

The definitions illustrate a few differences between IDL and Java:

- ◆ Each parameter *must* be qualified by in, out, or inout. In parameters are pass-by-value, and transfer a copy of the argument during a remote method call. Out and inout parameters are described in the "[Holders](#)" section.
- ◆ IDL uses the keyword raises to denote that an exception can be thrown.
- ◆ IDL items scoped within curly braces are terminated with a semicolon.
- ◆ All items in IDL are public.
- ◆ Interface names are not required to begin with "I", but it is a common convention.

You can store all of the IDL definitions in a single file; however, it is common to place each IDL definition in its own file. IDL supports the full C++ preprocessor specification, allowing IDL files to refer to each other and protect themselves from multiple inclusion. The preprocessor directives are #include, #ifdef, #ifndef, #if, #else, #elif, #endif, #define, #undef, #error, and #pragma.

Typically, OverdrawnException.idl and IAccount.idl are written as follows:

OverdrawnException.idl

```
#ifndef _OverdrawnException_idl // protect against multiple inclusion
#define _OverdrawnException_idl
```

```
exception OverdrawnException
{
    string message;
    long amount;
};
```

```
#endif // corresponds to first line #ifndef
```

IAccount.idl

```
#ifndef _IAccount_idl
#define _IAccount_idl
```

```
#include "OverdrawnException.idl" // read IDL definition
```

```
interface IAccount
{
```

```
long deposit( in long arg1 );
long getBalance();
void withdraw( in long arg1 ) raises( OverdrawnException );
};
```

```
#endif
```

As with C++, you can forward declare interfaces to avoid cases of mutual inclusion or to decrease the number of #includes. Repeated forward declaration of the same interface is legal.

IBank.idl

```
#ifndef _IBank_idl
#define _IBank_idl
```

```
interface IAccount; // forward declare IAccount
```

```
interface IBank
{
    IAccount openAccount();
};
```

```
#endif
```

In this example, all of the IDL files are stored in examples\corba\common.

Step 2: Generate Java from IDL

Before you can write the server or client programs, you must convert the IDL into its Java equivalent. The cgen utility implements the standard IDL-to-Java mapping and generates Java automatically from IDL. The mapping is defined in the OMG IDL-to-java specification and follows:

IDL	Java Equivalent	Notes
interface	interface	Like Java, IDL supports multiple interface inheritance
exception	exception	Unlike Java, IDL does not support exception inheritance
operation	method	
void	void	
char	char	
wchar	char	Voyager does not currently support wide characters
octet	byte	
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	
long double	double	
string	String	
wstring	String	Voyager does not currently support wide strings
Object	Object	

To create Java from IDL, type `cgen` followed by a list of the IDL files to process. To suppress the generation of holder classes that are only required when using `out` and `inout` parameters, use the `-h` option. The `-v` option generates a verbose description of activities.

A Java interface generated by `cgen` extends `IRemote` by default. You can use the `cgen -r` flag to make the interface extend `java.rmi.Remote` instead. Because `IRemote` and `Remote` are treated identically by Voyager, and Microsoft does not currently support RMI, we recommend that you use the default setting.

To generate Java from the banking IDL files for the server program, type:

```
% cd \voyager\examples\corba\server
% copy ..\common\*.idl .
% cgen OverdrawnException.idl IAccount.idl IBank.idl -h -v
cgen 3.0, copyright objectspace 1997-1999
```

```
reading files...
read OverdrawnException.idl
got ::OverdrawnException
read IAccount.idl
got ::IAccount
read IBank.idl
got ::IBank
```

```
writing files...
write OverdrawnException.java
write IAccount.java
write IBank.java
%
```

The Java bindings for the client program are generated in the same way.

The following Java files are generated by `cgen` from the banking application IDL files. The runtime support code that `cgen` embeds in the generated Java is omitted in the listings for clarity

OverdrawnException.java

```
public final class OverdrawnException extends Exception
{
    public java.lang.String message;
    public int amount;

    public OverdrawnException()
```



```

{
}

public OverdrawnException( java.lang.String message, int amount )
{
    super( message );
    this.message = message;
    this.amount = amount;
}

public String toString()
{
    return "OverdrawnException( " + message + ", " + amount + " )";
}
}

```

IAccount.java

```

public interface IAccount extends com.objectspace.voyager.IRemote
{
    public int deposit( int arg1 );
    public int getBalance();
    public void withdraw( int arg1 ) throws OverdrawnException;
}

```

IBank.java

```

public interface IBank extends com.objectspace.voyager.IRemote
{
    public IAccount openAccount();
}

```

Step 3: Write implementations for each Java interface

Because only the server program needs implementations of the interfaces, Account.java and Bank.java can reside in the examples\corba\server directory. None of the code contains anything special or unique to CORBA. Voyager's universal communications architecture allows instances of these classes to simultaneously process messages from any common standard.

The code for the implementation classes Account and Bank follows:

Account.java

```
public class Account implements IAccount
{
    private int balance;

    public int deposit( int amount )
    {
        balance += amount;
        return amount;
    }

    public int getBalance()
    {
        return balance;
    }

    public void withdraw( int amount ) throws OverdrawnException
    {
        if( amount > balance )
            throw new OverdrawnException( "only have $" + amount, amount - balance );

        balance -= amount;
    }
}
```

Bank.java

```
import java.util.Vector;

public class Bank implements IBank
{
    Vector accounts = new Vector();

    public IAccount openAccount()
    {
        System.out.println( "open account" );
        Account account = new Account();
        accounts.addElement( account );
        return account;
    }
}
```

Step 4: Write the Server

The server program creates a local Bank object, starts up Voyager, exports the object for use by another CORBA ORB, and then accepts incoming messages.

You can transfer an object reference between ORBS as follows:

- ◆ Pass it as a method argument or return value.

If you pass a proxy to an object as a remote method argument or return value, it is automatically transferred as a standard IIOP object reference. If you pass a regular object, Voyager automatically exports the object and then passes a proxy to the newly exported object. Because CORBA does not support distributed garbage collection, Voyager include a facility called *anchoring*. If enabled, which is the default, anchoring prevents the exported local object from being garbage collected. The anchoring feature is illustrated by the banking example when `Bank.openAccount()` is called by a remote CORBA ORB. The Account object is anchored in the local program, and then a proxy to the anchored object is returned. To toggle auto-anchoring, use `Corba.setAnchoring()`.

- ◆ Register it with a CORBA naming service.

Voyager ORB Professional includes a persistent federated CORBA naming service. Information about how to use this naming service is described in the "[Naming Service](#)" section.

- ◆ Obtain its Interoperable Object Reference (IOR).

An IOR is a string that encodes the host name, port number, type, and key of a single CORBA object. To obtain an object's IOR, pass the object or a proxy to `Corba.asIOR()`. If the argument is not already a proxy, it is converted into a proxy using `Proxy.of()` and anchored when anchoring is enabled. By default, the IOR type field is set to the first remote interface of the object. To override this setting, use the variation of `Corba.asIOR()` that allows you to specify the type field explicitly. A server can export an object to a client by writing its IOR to a file so that the client can read the IOR and bind to it using `Namespace.lookup()`. In the example, a reference to the bank object is shared with the client by storing its IOR into a shared file `..\common\Bank.IOR`.

The server program follows:

Server.java

```
import java.io.RandomAccessFile;
import com.objectspace.voyager.*;
import com.objectspace.voyager.corba.*;

public class Server
{
    static IBank bank = new Bank(); // holds onto local Bank object

    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            String ior = Corba.asIOR( bank ); // get IOR for object
            System.out.println( "bank IOR = " + ior ); // display IOR
            RandomAccessFile file = new RandomAccessFile( "..\\common\\Bank.IOR", "rw" );
            file.writeUTF( ior ); // write IOR to file
            file.close();
            System.out.println( "CORBA server is ready" );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }
    }
}
```

Step 5: Write the Client

The client program starts up Voyager, obtains a proxy to the remote bank by passing its IOR to `Namespace.lookup()`, opens an account, and then deposits or withdraws money.

When the client sends `openAccount()`, the Account is created in the remote ORB and a proxy to the Account is passed back to the client. The anchoring feature prevents the Account from being garbage collected. Voyager creates proxy classes dynamically at runtime.

The client program follows:

Client.java

```
import java.io.RandomAccessFile;
import com.objectspace.voyager.*;
import com.objectspace.voyager.corba.*;

public class Client
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            RandomAccessFile file = new RandomAccessFile( "..\\common\\Bank.IOR", "r" );
            String ior = file.readUTF(); // read IOR from file
            file.close();
            System.out.println( "Bank IOR = " + ior ); // display IOR
            IBank bank = (IBank) Namespace.lookup( ior );
            IAccount account = bank.openAccount();
            account.deposit( 1000 );
            System.out.println( "account balance = " + account.getBalance() );
            account.withdraw( 500 );
            System.out.println( "account balance = " + account.getBalance() );
            account.withdraw( 2000 );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }

        Voyager.shutdown();
    }
}
```

Step 6: Run the Application

To run the application, start the Server program first and wait for its initial screen output. Then run the Client program. Output displays as follows:

```
% java Server
bank IOR = IOR:0000000000000000e4944...0000
CORBA server is ready
open account
```

```
% java Client
bank IOR = IOR:0000000000000000e4944...0000
account balance = 1000
account balance = 500
OverdrawnException( only have $2000, 1500 )
```

Struct

An IDL struct contains one or more fields and is used to pass groups of data by value between method invocations. An example follows:

Person.idl

```
struct Person
{
    string name;
    long age;
};
```

The cgen utility generates the Java equivalent of a struct as follows:

- ◆ the class has the same name as the IDL struct and is public final.
- ◆ each member in the IDL struct is declared as a public data member.
- ◆ the class has a default constructor and a constructor that accepts a value for each member.

The code that cgen generates from Person.idl follows:

Person.java

```
public final class Person
{
    public java.lang.String name;
    public int age;

    public Person()
    {
    }

    public Person( java.lang.String name, int age )
    {
        this.name = name;
        this.age = age;
    }

    public String toString()
    {
        return "Person( " + name + ", " + age + " )";
    }

    public boolean equals( Object object )
    {
        if ( !( object instanceof Person ) )
            return false;

        Person other = (Person) object;
        return com.objectspace.voyager.corba.TypeCode.__equals( name, other.name )
            && (age == other.age); // __equals is a helper method in TypeCode
    }
}
```

Example

The [Structs](#) example illustrates how you can use an IDL struct to pass groups of data by value between CORBA ORBs.

Holders

The IDL parameter passing modes are as follows:

- ◆ in
Passes a copy of the input parameter to the remote server. Changes made to the parameter on the remote server are not propagated back to the client.
- ◆ out
No input parameter is passed. Changes made to the parameter on the remote server are propagated back to the client.
- ◆ inout
Sends a copy of the input parameter to the remote server. Changes made to the parameter on the remote server are propagated back to the client.

Out and inout parameters are often used when a method needs to return more than one value. Voyager supports out and inout modes by supporting the industry standard holder mechanism. A method can send out or inout parameters by using a corresponding holder object for each parameter to be passed. A holder object wraps the parameter value, carries it to the remote server for the method call, and carries the final value back to the client.

The `com.objectspace.lib.holder` package contains pre-built holder classes for all Java primitives, String class, and Object class. When `cgen` converts an IDL interface, struct, enum, union, typedef sequence or typedef array to Java, it automatically emits its holder class. You can suppress generation of holder classes using the `-h` option.

The `cgen` utility generates a holder for type XXX as follows:

- ◆ The class is named XXXHolder and is public final.
- ◆ There is a single public data member called “value” of type XXX.
- ◆ There is a default constructor used for sending out parameters.
- ◆ There is a constructor that takes an initial value of type XXX for sending inout parameters.
- ◆ If XXX is declared inside an interface YYY, the holder class is placed into the package YYYPackage.

- ◆ If XXX is declared outside an interface, the holder class is placed into the same package as XXX.

An example IDL class and its cgen-generated holder class follows:

Ticket.idl

```
struct Ticket
{
    long x;
    long y;
    long z;
};
```

TicketHolder.java

```
final public class TicketHolder
{
    public Ticket value;

    public TicketHolder()
    {
    }

    public TicketHolder( Ticket value )
    {
        this.value = value;
    }

    public String toString()
    {
        return String.valueOf( value );
    }
}
```

Example

The example [Holders](#) illustrates the use of out and inout parameters.

Typedef

The IDL typedef facility allows you to create convenient aliases for type definitions. In most cases, typedef is optional. However, in some cases, use of a typedef is mandatory. An example of this is shown in the ["Arrays and Sequences"](#) section. To make <id> an alias for <type>, write:

```
typedef <type> <id>;
```

Two examples of typedef from the CORBA naming service IDL follow:

CosNaming.idl

```
module CosNaming
{
    typedef string lstring; // placeholder for international string type

    struct NameComponent
    {
        lstring id;
        lstring kind;
    };
    typedef sequence< NameComponent > Name;

    enum BindingType{ nobject, ncontext };

    struct Binding
    {
        Name binding_name;
        BindingType binding_type;
    };

    // rest of CosNaming.idl
};
```

Const

An IDL const is a symbolic representation of the value of a constant expression. The constant expression must evaluate to a boolean, integer, floating point number, or enumerator. If the IDL const is declared in an IDL interface, cgen translates it into a public static final field in the generated Java interface.

IFoo.idl

```
interface Foo
{
    const float PI = 3.14;
};
```

IFoo.java

```
public interface IFoo extends com.objectspace.voyager.IRemote
{
    public static final float PI = ( float ) 3.14;
}
```

If declared outside of an interface, cgen converts the const into a Java interface with the same name that contains a single, public-static final field called value.

Math.idl

```
module Math
{
    const float PI = 3.14;
};
```

Math\PI.java

```
package Math;

public interface PI
{
    public static final float value = ( float ) 3.14;
}
```

Constant Expressions

IDL allows you to use constant expressions for const values, array dimensions, sequence dimensions, string dimensions, and case labels. A constant expression is built by combining one or more literals with one or more operators.

The valid literals follow:

- ◆ The boolean literals are TRUE and FALSE.
- ◆ An integer literal beginning with a zero and an uppercase or lowercase x (0x or 0X) is treated as a hexadecimal number. An integer literal beginning with only a zero (0) is treated as an octal number. Examples of integer literals are 42, 0x3A, 0X4C, and 0342.
- ◆ A floating point literal contains a decimal point and a fractional part. A floating point literal might have an optional signed exponent. Examples of floating point literals are 3.14, 9.8e7, and 4.3e-12.
- ◆ A character literal is placed between single quotes and can either be a letter, like 'r', or a predefined escape sequence, like '\f'. The valid escape sequences are \n (new line), \t (horizontal tab), \v (vertical tab), \b (backspace), \r (carriage return), \f (form feed), \a (alert), \\ (backslash), \? (question mark), \' (single quote), \" (double quote), \ooo (octal number), and \xhh (hex number).
- ◆ A string literal is a sequence of zero or more characters between double quotes. The characters may include any of the predefined character escape sequences. Examples of string literals are "string", "", and "who?". Voyager does not currently support concatenation of adjacent string literals.

A description of each IDL operator follows, in order of precedence:

Operator	Description	Operand Types
	Logical bitwise OR	Integer only
^	Logical bitwise XOR	Integer only
&	Logical bitwise AND	Integer only
>> and <<	Shift a bit right and left	Integer only
+ and –	Add and subtract	Integer or floating point
*, /, and %	Multiply, divide, and calculate a modulus.	Integer or floating point
+, -, and ~	Unary plus, unary minus, and unary bitwise NOT	Integer or floating point
()	Force evaluation of a sub-expression	N/A

Examples of valid IDL const expressions include:

```
const long X = 42;
const short Z = (13 * 4) << 2;
const boolean B = FALSE;
const char C = '\n';
const float PI = 3.14;
const double D = 34.22e-13;
const string S = "hi there\n";
```

Arrays and Sequences

An IDL array is an N-dimensional fixed-size container. An IDL sequence is an N-dimensional, variable-size container. A sequence may be either bounded or unbounded.

Any argument or return type that is an array or sequence must be specified using a typedef. For example:

IMath.idl

```
typedef sequence< long, 10 > LongSequence; // bounded
typedef long LongArray[ 3 ];
typedef sequence< sequence< long > > LongMatrix; // unbounded
```

```
interface IMath
{
    long addLongSequence( in LongSequence numbers );
    long addLongArray( in LongArray numbers );
    void multiMatrix( inout LongMatrix matrix, in long n );
};
```

The cgen utility and Voyager runtime processes arrays and sequences as follows:

- ◆ An IDL array is mapped to a Java array of the same dimension. An exception occurs when you try to set an array to null or to an array of the wrong size.
- ◆ An IDL sequence is mapped to a Java array of the same dimension. An exception occurs when you try to set a bounded sequence to an array that exceeds the bounds.

The code that cgen generates from IMath.idl follows:

IMath.java

```
public interface IMath extends com.objectspace.voyager.IRemote
{
    public int addLongSequence( int[] numbers );
    public int addLongArray( int[] numbers );
    public void multiMatrix( LongMatrixHolder matrix, int n );
}
```

LongMatrixHolder.java

```
final public class LongMatrixHolder
{
    public int[][] value;

    public LongMatrixHolder()
    {
    }

    public LongMatrixHolder( int[][] value )
    {
        this.value = value;
    }

    public String toString()
    {
        return String.valueOf( value );
    }
}
```

Example

The example [Arrays](#) illustrates the use of arrays and sequences.

Modules and Interfaces

An IDL module is analogous to a Java package and can define one or more of the following IDL type definitions: module, interface, exception, struct, union, enum, const, or typedef. You may add new items to a module after an existing module definition by re-opening the module and declaring the extra items. An example follows.

Containers.idl

```
module Containers
{
    interface IQueue
    {
```

```

struct Element
{
    Object object;
    long count;
};

void add( in Containers.IQueuePackage.Element object );
Containers.IQueuePackage.Element remove();
};

interface IStack
{
    void push( in Object object );
    Object pop();
};

module Containers // re-open module, add some more stuff...
{
    interface IArray
    {
        Object elementAt( in long index );
        void setElementAt( in long index, in Object object );
    };
};

```

The cgen utility packages IDL items according to the following rules:

- ◆ If the item is a struct, union, exception, or enum in the interface XXX, its Java code is placed into a subpackage XXXPackage within the interface's package.
- ◆ If the item is not declared in a module, its Java code is not in a package.
- ◆ If neither of the above scenarios is true, the item is placed into the package associated with its module.

The code that cgen generates from Containers.idl follows:

Containers\IQueue.java

```

package Containers;

public interface IQueue extends com.objectspace.voyager.IRemote
{

```



```

    public void add(Containers.IQueuePackage.Element object );
    public Containers.IQueuePackage.Element remove();
}

```

Containers\IQueuePackage\Element.java

```

package Containers.IQueuePackage;

public final class Element
{
    public java.lang.Object object;
    public int count;

    public Element()
    {
    }

    public Element( java.lang.Object object, int count )
    {
        this.object = object;
        this.count = count;
    }

    public String toString()
    {
        return "Element( " + object + ", " + count + " )";
    }
}

```

Containers\IStack.java

```

package Containers;

public interface IStack extends com.objectspace.voyager.IRemote
{
    public void push( java.lang.Object object );
    public java.lang.Object pop();
}

```

Containers\IArray.java

```

package Containers;

public interface IArray extends com.objectspace.voyager.IRemote
{

```

```
public java.lang.Object elementAt( int index );  
public void setElementAt( int index, java.lang.Object object );  
}
```

Inheritance

An IDL interface is analogous to a Java interface and can contain zero or more of the following IDL type definitions: operation, exception, struct, union, enum, const, attribute, or typedef. An IDL interface can inherit from zero or more interfaces. An example follows.

Inheritance.idl

```
interface IA  
{  
};  
  
interface IB : IA  
{  
};  
  
interface IC : IA  
{  
};  
  
interface ID : IB, IC  
{  
};
```

The cgen utility generates the Java equivalent of an interface as follows:

- ◆ An IDL interface maps to a Java interface that implements IRemote.
- ◆ IDL inheritance maps directly to Java inheritance.

To extend an interface from java.rmi.Remote instead of IRemote, use the cgen -r option.

The code that cgen generates from Inheritance.idl follows:

IA.java

```
public interface IA extends com.objectspace.voyager.IRemote  
{
```

```
}
```

IB.java

```
public interface IB extends IA, com.objectspace.voyager.IRemote
{
}
```

IC.java

```
public interface IC extends IA, com.objectspace.voyager.IRemote
{
}
```

ID.java

```
public interface ID extends IB, IC, com.objectspace.voyager.IRemote
{
}
```

Scoping

The IDL scoping rules are similar to those of Java. The IDL items that scope their elements are modules, interfaces, structs, unions, and exceptions. The IDL item Operations also scopes its argument names.

There are three ways to name an IDL item:

1. A *fully qualified* name is “::” followed by a list of one or more tokens separated by “::”. The fully qualified name “::A::B::C” is resolved as “the item C declared within the item B declared within the item A declared in global scope, that is, outside any module.”
2. An *unqualified* name is a single token. The unqualified name “C” is resolved as “the first item called C that is found by starting in the current scope and searching upwards towards the global scope.” In the case of multiple interface inheritance, it is an error when more than one item satisfies this criterion.
3. A *partially qualified* name is a list of one or more tokens separated by “::”. The partially qualified name “A::B::C” is resolved as “the item C declared within the item B declared within the item whose unqualified name is A”.

An example IDL file that contains numerous examples of each type of name and the value to which each resolves follows:

Scoping.idl

```
module Scoping
{
    const long X = 10;

    interface IA
    {
        const long X = 20;
        const long A = X; // 20
        const long B = Scoping::X; // 10
        const long C = ::Scoping::X; // 10
    };

    interface IB : IA
    {
        const long X = 30;
        const long A = X; // 30
        const long B = Scoping::X; // 10
        const long C = ::Scoping::IA::X; // 20
        const long D = IA::X; // 20
        const long E = ::Scoping::IB::X; // 30
    };
};
```

The code that cgen generates from Scoping.idl follows:

X.java

```
package Scoping;

public interface X
{
    public static final int value = ( int ) 10;
}
```

IA.java

```
package Scoping;

public interface IA extends com.objectspace.voyager.IRemote
```

```

{
public static final int X = ( int ) 20;
public static final int A = ( int ) 20;
public static final int B = ( int ) 10;
public static final int C = ( int ) 10;
}

```

IB.java

```
package Scoping;
```

```

public interface IB extends Scoping.IA, com.objectspace.voyager.IRemote
{
public static final int X = ( int ) 30;
public static final int A = ( int ) 30;
public static final int B = ( int ) 10;
public static final int C = ( int ) 20;
public static final int D = ( int ) 20;
public static final int E = ( int ) 30;
}

```

Enum

An IDL enum is a typesafe way to represent one or more enumerations. An example follows:

Color.idl

```

enum Color
{
red, green, blue
};

```

The cgen utility generates the Java equivalent of an enum in the following way:

- ◆ The class has the same name as the IDL enum and is public final.
- ◆ Each label is allocated an int value, starting at zero and incremented by one.
- ◆ Each label has a public static final field for its int value.
- ◆ Each label has a public static final field for its symbolic value.

- ◆ The class has a static method `from_int()` that maps an `int` to an instance of the enum.
- ◆ The class has a method `value()` that returns the `int` value of an enum.

The code that cgen generates from `Color.idl` follows:

Color.java

```
public final class Color
{
    public static final int _red = 0;
    public static final Color red = new Color( _red );
    public static final int _green = 1;
    public static final Color green = new Color( _green );
    public static final int _blue = 2;
    public static final Color blue = new Color( _blue );
    private static final Color[] values = new Color[]{ red, green, blue };
    private static final String[] names = new String[]{ "red", "green", "blue" };

    public static Color from_int( int value )
    {
        return values[ value ];
    }

    private int value;

    public String toString()
    {
        return "Color( " + names[ value ] + " )";
    }

    private Color( int value )
    {
        this.value = value;
    }

    public int value()
    {
        return value;
    }
}
```

Example

The example [Enums](#) illustrates the use of enums.

Union

An IDL union is an entity that can represent one of several kinds of element, depending on the value of a discriminator field. The discriminator must be a boolean, short, long, enum, or char. An example follows:

Winnings.idl

```
union Winnings switch( short )
{
  case 1:
  case 2:
    short regular;

  case 3:
    long jackpot;

  default:
    octet booby;
};
```

The cgen utility generates the Java equivalent of a union as follows:

- ◆ The class has the same name as the IDL union, is public final, and has a default constructor.
- ◆ The discriminator is stored as a public field of the appropriate type.
- ◆ The discriminator has an accessor called `discriminator()`.
- ◆ The actual value is always stored as an Object in a public field called `value`.
- ◆ each possible element has an accessor and mutator.

- ◆ If an element has more than one case, the default mutator sets the discriminator to the lowest value.
- ◆ If an element has more than one case, a mutator that takes an additional explicit value is added.
- ◆ If an explicit default is not provided and one is possible, a method called `_default()` is added.

The code that cgen generates from `Winnings.idl` follows:

Winnings.java

```
public final class Winnings
{
    public short discriminator;
    public java.lang.Object value;

    public Winnings()
    {
    }

    public String toString()
    {
        return "Winnings( " + value + " )";
    }

    public boolean equals( Object object )
    {
        return object instanceof Winnings && ((Winnings) object).value.equals( value );
    }

    public short discriminator()
    {
        return discriminator;
    }

    public short regular()
    {
        return ((Short) value).shortValue();
    }

    public void regular( short value )
```



```

{
    this.value = new Short( value );
    this.discriminator = 1;
}

public void regular( short discriminator, short value )
{
    this.value = new Short( value );
    this.discriminator = discriminator;
}

public int jackpot()
{
    return ((Integer) value).intValue();
}

public void jackpot( int value )
{
    this.value = new Integer( value );
    this.discriminator = 3;
}

public byte booby()
{
    return ((Byte) value).byteValue();
}

public void booby( byte value )
{
    this.value = new Byte( value );
    this.discriminator = 0;
}
}

```

Example

The example [Unions](#) illustrates the use of unions.

TypeCode and Any

Every IDL type has an associated typecode that describes the type name, repository id, fields, and other information. Typecodes are a form of a language-neutral reflection mechanism.

To obtain the `TypeCode` of a specific object or class, use `TypeCode.getTypeCode()`. `TypeCode` contains accessors like `kind()`, `id()`, `name()`, and `memberCount()` which allow you to access information about a type. The `kind()` method returns an `int` that indicates the type category. For example, `tk_long` (3) represents the long type, `tk_string` (18) represents a string type, and `tk_struct` (15) represents a struct type. The full range of `tk_xxx` constants is defined in `Constants`.

Not all of the `TypeCode` accessors are legal for every type. For example, `memberCount()` is an illegal operation when the typecode represents a long. In these cases, a `TypeCodeException` is thrown.

Typecodes are represented in IDL by `CORBA::TypeCode` and can be used as arguments and return types.

An IDL `any` holds an encoded object and its associated typecode. The `Any.setXXX()` family of methods encoded their argument and store it in the `any`. The `Any.getXXX()` family of methods return the object in a decoded state, and `Any.type()` returns its encoded object's typecode. Because an `any` holds an object in an encoded state, it may be passed into an ORB that knows nothing of the object's type as long as the ORB does not attempt to invoke `Any.getXXX()`.

An example of some operations that accept `any` and `TypeCode` arguments follows:

IPrinter.idl

```
interface IPrinter
{
    void printAny( in any x );
    void printTypeCode( in CORBA::TypeCode x );
};
```

The Java code generated by `cgen` from `IPrinter.idl` follows:

IPrinter.java

```
public interface IPrinter extends com.objectspace.voyager.IRemote
```

```

{
public void printAny( com.objectspace.voyager.corba.Any x );
public void printTypeCode( com.objectspace.voyager.corba.TypeCode x );
}

```

Example

The example [TypeCodes](#) illustrates the use of anys and typecodes.

Attributes

An IDL attribute allows you to specify that an interface provides accessors to a typed value. An example follows:

ICar.idl

```

interface ICar
{
    readonly attribute string make;
    attribute long speed;
};

```

The cgen utility generates the Java equivalent of an attribute as follows:

- ◆ A readonly attribute is mapped to an accessor with the same name.
- ◆ A writable attribute is mapped to an accessor and a mutator with the same name.

The code that cgen generates from ICar.idl follows:

ICar.java

```

public interface ICar extends com.objectspace.voyager.IRemote
{
    public java.lang.String make();
    public int speed();
    public void speed( int value );
}

```

Note: The standard OMG naming does not correspond to the industry-standard JavaBeans convention of `setXXX()` and `getXXX()`.

Example

The example [Attributes](#) illustrates the use of attributes.

User Exceptions

An IDL exception is raised when an error occurs within an operation. An example declaration of an IDL exception follows:

OverdrawnException.idl

```
exception OverdrawnException
{
    string message;
    long amount;
};
```

The `cgen` utility generates the Java equivalent of an exception as follows:

- ◆ The class has the same name as the IDL exception and is public final.
- ◆ The class extends `java.lang.Exception`.
- ◆ Every field in the IDL exception is declared as a public data member.
- ◆ The class has a default constructor and a constructor that takes a value for each field.
- ◆ If the first field is a string, it is passed to the `java.lang.Exception` constructor.

The code that `cgen` generates from `OverdrawnException.idl` follows:

OverdrawnException.java

```
public final class OverdrawnException extends Exception
```

```

{
    public java.lang.String message;
    public int amount;

    public OverdrawnException()
    {
    }

    public OverdrawnException( java.lang.String message, int amount )
    {
        super( message );
        this.message = message;
        this.amount = amount;
    }

    public String toString()
    {
        return "OverdrawnException( " + message + ", " + amount + " )";
    }
}

```

System Exceptions

If a CORBA ORB generates an exception, it is thrown as a `SystemException` that extends `RuntimeException` and contains a reason message, minor code, and status code.

The valid values of the status code are `COMPLETED_YES`, `COMPLETED_NO`, and `COMPLETED_MAYBE`, defined in `IConstants`.

Voyager converts its own internal exceptions into `SystemExceptions` as follows:

Voyager converts its own internal exceptions into `SystemExceptions` as follows:

Voyager Exception	CORBA SystemException reason
<code>IOException</code>	<code>IDL:omg.org/CORBA/MARSHAL:1.0</code>
<code>ClassNotFoundException</code>	<code>IDL:omg.org/CORBA/MARSHAL:1.0</code>
<code>MethodNotFoundException</code>	<code>IDL:omg.org/CORBA/NO_IMPLEMENT:1.0</code>

NoSuchObjectException	IDL:omg.org/CORBA/OBJECT_NOT_EXIST:1.0
ObjectNotFoundException	IDL:omg.org/CORBA/OBJECT_NOT_EXIST:1.0
RemoteException	IDL:omg.org/CORBA/UNKNOWN:1.0
RuntimeRemoteException	IDL:omg.org/CORBA/UNKNOWN:1.0

Voyager always sets the minor code to 0 and the status to COMPLETED_NO.

Standard Object Methods

Standard methods defined in the CORBA version of Object follow:

- ◆ boolean _is_nil()
- ◆ boolean _is_a(String identifier)
- ◆ boolean _is_equivalent(Object that)
- ◆ boolean _non_existent()
- ◆ int _hash(int maximum)
- ◆ Object _duplicate()
- ◆ void _release()

These methods act locally on the object or proxy and never generate any network traffic.

To execute one of these methods on an object or proxy, use `CorbaObject.of()` to obtain a `CorbaObject` wrapper and then execute the required method. `CorbaObject` makes the object that it wraps behave as if it inherited from `CORBA::Object`.

For example:

```
IX x = (IX) Namespace.lookup( ior );
System.out.println( CorbaObject.of( x )._is_nil() );
System.out.println( CorbaObject.of( x )._is_a( "IDL:IX:1.0" ) );
System.out.println( CorbaObject.of( x )._hash( 100 ) );
```

Narrowing

To navigate between unrelated interfaces on a remote object, use `Corba.narrow()` instead of a regular Java cast. For example, if `x` is a remote reference of type `IX` to an object that implements the unrelated interfaces `IX` and `IY`, the following code returns a remote reference of type `IY` to the same object.

```
IX x = (IX) Namespace.lookup( ior );  
IY y = (IY) Corba.narrow( x, IY.class );
```

Example

The example [Narrowing](#) illustrates the use of narrowing.

Java to IDL

The Java to IDL facility allows a regular Java application to export a subset of its objects to CORBA without requiring modification of any of the original source code. Use `cgen` to generate IDL for the objects you want to export, and share references to the objects using a naming service, IORs, or by passing them as arguments to remote CORBA method invocations.

To generate IDL from Java, invoke `cgen` with a list of the `.java/.class` files whose IDL you require. `cgen` translates Java interfaces, methods, and exceptions using the following rules:

Interfaces

- ◆ A Java interface can only be translated when it directly or indirectly extends `IRemote` or `Remote`.
- ◆ A Java interface is placed into the IDL module that corresponds to its Java package.
- ◆ Java inheritance maps to IDL inheritance, and non-remote interfaces are ignored.

Methods

- ◆ All Java parameters are mapped to an IDL in argument.
- ◆ Parameter types that are primitives, Strings, Objects, or remote interfaces can be mapped to IDL.
- ◆ Arrays of mappable types are mapped to IDL unbound sequences.
- ◆ Any method that contains a parameter type that cannot be mapped to IDL is ignored.
- ◆ If two or more methods have the same name, their names are mangled based on their parameter types.
- ◆ Incoming mangled IDL invocations are automatically unmangled by the Voyager CORBA runtime.

Exceptions

- ◆ A Java exception is mapped to an IDL exception containing all the public non-static fields.
- ◆ If the exception does not extend Exception, the IDL contains a union of all the inherited fields.

The following example illustrates the Java to IDL mapping:

IAccount.java

```
public interface IAccount extends com.objectspace.voyager.IRemote
{
    public int deposit( int amount );
    public int getBalance();
    public void withdraw( int amount ) throws OverdrawnException;
    public void withdraw( int[] amounts ) throws BatchException;
}
```

OverdrawnException.java

```
public class OverdrawnException extends Exception
{
    public int amount;

    public OverdrawnException()
    {
    }
}
```



```

public OverdrawnException( String message, int amount )
{
    super( message );
    this.amount = amount;
}

public String toString()
{
    return "OverdrawnException( " + getMessage() + ", " + amount + " )";
}
}

```

BatchException.java

```

public class BatchException extends OverdrawnException
{
    public int count;

    public BatchException()
    {
    }

    public BatchException( String message, int amount, int count )
    {
        super( message, amount );
        this.count = count;
    }

    public String toString()
    {
        return "BatchException( " + getMessage() + ", " + amount + ", " + count + " )";
    }
}

```

The IDL that cgen generates from these files follows:

IAccount.idl

```

#ifndef _IAccount_idl
#define _IAccount_idl

#include "OverdrawnException.idl"
#include "BatchException.idl"

```

```

interface IAccount
{
    typedef sequence< long > tmp1;
    long deposit( in long arg1 );
    long getBalance();
    void withdraw__long( in long arg1 ) raises( ::OverdrawnException );
    void withdraw__sequence_long( in tmp1 arg1 ) raises( ::BatchException );
};

```

```

#endif

```

OverdrawnException.idl

```

#ifndef _OverdrawnException_idl
#define _OverdrawnException_idl

```

```

exception OverdrawnException
{
    string _message;
    long amount;
};

```

```

#endif

```

BatchException.idl

```

#ifndef _BatchException_idl
#define _BatchException_idl

```

```

exception BatchException
{
    string _message;
    long amount;
    long count;
};

```

```

#endif

```

Example

The example [Java to IDL](#) illustrates the use of the cgen to generate Java from IDL.

Prefixes, Versions, and Repository IDs

By default, the repository ID for an IDL type is equal to “IDL:”, followed by the full IDL type converted from “::” format to “/” format, followed by version number “:1.0”. There are a few preprocessor directives that modify an IDL type’s repository ID:

- ◆ `pragma prefix "<prefix>"` appends the specified prefix to the repository id of every IDL type from the occurrence of the `#pragma` until the `#pragma` goes out of scope.
- ◆ `pragma version <identifier> major.minor` sets the repository id version number of the specified IDL type to be `major.minor`.
- ◆ `pragma ID <identifier> "<repositoryID>"` sets the repository id of the specified IDL type to be the specified string. You must use `-m` for resolution.

These pragmas only modify on-the-wire encoding of object references and do not affect the cgen Java mapping. An example of these `#pragmas` in action follows:

X.idl

```
#pragma prefix "p.q"
```

```
module X
```

```
{  
  interface IA  
  {  
  };  
}
```

```
#pragma prefix "s.t"
```

```
interface IB  
{  
  #pragma version IB 2.0  
};
```

```
interface IC  
{  
  #pragma ID IC "IDL:x.y/IZ:3.0"  
};  
};
```

The effect of these `#pragmas` follows:

IDL type	Repository ID without #pragmas	Repository ID with #pragmas
::X::IA	IDL:X/IA:1.0	IDL:p.q/X/IA:1.0
::X::IB	IDL:X/IB:1.0	IDL:s.t/X/IB:2.0
::X::IC	IDL:X/IC:1.0	IDL:x.y/IZ:3.0

Example

The example [Pragma](#) illustrates the use of #pragmas.

Repackaging

By default, cgen places IDL entities into Java packages based on their IDL module. For example, if you run cgen on CosNaming.idl, the Java classes are placed into the package CosNaming, which is the IDL module that defines NameComponent, Binding, and the other CORBA name service items.

To override the default mapping rule at compile, use cgen -m <repository-id> <java-name> as follows:

- ◆ If the <repository-id> has no prefix, use the format A/B/C.
- ◆ If the <repository-id> has the prefix X.Y, use the format [X.Y]A/B/C.
- ◆ The <java-name> should be in the format A.B.C.
- ◆ If the <repository-id> resolves to an individual item, only that item is mapped.
- ◆ If the <repository-id> resolves to a module, all items within that module are mapped recursively.

For example, map the CosNaming module to com.objectspace.voyager.corba.naming by typing:

```
cgen CosNaming.idl -d \voyager -m [omg.org]CosNaming com.objectspace.voyager.corba.naming
```

If you have a set of several mapping options, it is recommended that you store them into a text file and use the cgen -a option to read the options. For example:

Mapping.txt

```
-m [omg.org]CosNaming com.objectspace.voyager.corba.naming
-m foo.bar Foo.MyBar
-m gaia.rocks voyager.version5
```

```
cgen -a Mapping.txt MyFile.idl
```

The table following the module definition illustrates how the module definition is mapped by different -m commands.

```
module A
{
  module B
  {
    interface C
    {
    };

    interface D
    {
    };
  };
};
```

Cgen argument	maps A/B/C to...	and maps A/B/D to...
<none>	A.B.C	A.B.D
-m A/B/C X.Y.Z	X.Y.Z	A.B.D
-m A/B X.Y	X.Y.C	X.Y.D
-m A X	X.B.C	X.B.D

The CORBA runtime system must be informed of the mappings used by cgen in order to dynamically map repository ids to their corresponding Java classes. You can supply this information in one of two ways.

1. The voyager -m and -a options work exactly like their cgen counterparts; therefore, you can start a Voyager server with the appropriate mappings.
2. The Corba.bindIdToJava(String id, String java) method allows you to add mappings from a Voyager application. The id and java strings should be formatted exactly like their cgen counterparts. For example, the voyager CORBA naming service adds its mapping rule by executing Corba.bindIdToJava("[omg.org]CosNaming", "com.objectspace.voyager.corba.naming") at startup.

The following table shows the run-time bindIdToJava() mappings that correspond to the compile-time cgen mappings previously described.

bindIdToJava() arguments	maps A/B/C to...	and maps A/B/D to...
<none>	A.B.C	A.B.D
("A/B/C","X.Y.Z")	X.Y.Z	A.B.D
("A/B","X.Y")	X.Y.C	X.Y.D
("A","X")	X.B.C	X.B.D

Dynamic Method Invocation

The Voyager OneWay, Sync, and Future classes work with the CORBA subsystem. To dynamically invoke a remote CORBA operation, pass the reference, method name, and argument array to the appropriate static invoke() method. Primitive values must be wrapped in their Object equivalents.

For example, ITimer.idl is defined as follows:

```
interface ITimer
{
    long wait( in long delay );
};
```

Code to execute a oneway wait(6000) operation when timerIOR is the IOR of a Timer implementation follows:

```
ITimer timer = (ITimer) Namespace.lookup( timerIOR );
Object[] args = new Object[]{ new Integer( 6000 ) };
OneWay.invoke( timer, "wait", args );
```

The cgen utility ignores the IDL oneway keyword, so this is the only way to execute CORBA operations asynchronously.

Example

The example [Dynamic](#) illustrates the use of dynamic method invocation.

Naming Service



The CORBA naming service allows objects to be bound to names in a hierarchical and federated fashion.

Voyager supports the CORBA naming service as follows:

- ◆ It allows access to any CORBA naming service via the universal Namespace API.
- ◆ It includes all of the standard classes for low-level access to a CORBA naming service. These classes were automatically generated by `cgen` from `CosNaming.idl` and placed into the `com.objectspace.voyager.corba.naming` package.
- ◆ It wraps the Voyager naming service with an adapter that makes it interface-compatible with a traditional implementation of a CORBA naming service. This approach is also adopted by Voyager's RMI subsystem and allows a Voyager naming service to simultaneously appear like an RMI registry to an RMI client and a CORBA naming service to a CORBA client.

Accessing a CORBA Naming Service via the Namespace API

Namespace interprets a URL that begins with "cos:" as a reference to a CORBA naming service. Operations like `bind()`, `unbind()`, and `lookup()` are automatically converted into their equivalent `CosNaming` operations and sent to the target naming service. This method is the simplest way to access a CORBA naming service because it does not require any knowledge of the `CosNaming` classes.

If the URL refers to a Voyager program, it will access the program's default naming service via the special CORBA adaptation layer. If the URL refers to the location of another vendor's CORBA ORB, it will access the CORBA naming service inside that ORB. Because there is no standardized way to obtain the `NamingContext` for an ORB's naming service, you must have previously associated that URL with a `NamingContext` using `CorbaNamingService.bindURLToNamingContext()`.

Accessing a CORBA Naming Service via the IDL-Generated Classes

You can access a CORBA naming service via the Namespace API or the IDL-generated class.

All of the CosNaming operations require you to first obtain a NamingContext. To obtain the NamingContext of your local Voyager naming service, use `CorbaNamingService.getDefaultNamingContext()`. To obtain the NamingContext of a remote CORBA naming service, use `CorbaNamingService.getNamingContext(String url)`. The URL is resolved to a CORBA naming service using the same rules as Namespace.

After a NamingContext is obtained, you can apply all of the CosNaming operations such as `bind()`, `resolve()`, and `list()`.

Example

The example [Naming Service](#) illustrates the use of the CORBA naming service.

Transactions

The Voyager CORBA system supports transaction propagation for integration with OTS-compliant systems. See the [Voyager Transactions Developer Guide](#) for more information. Also see the `cgen -t` option in the "[Utilities](#)" appendix.

CORBA Wide Character and Wide String Support

Voyager now supports wide characters and wide strings. When using the `cgen` utility to generate IDL files from Java, the Java primitive `char` is mapped to IDL `wchar`, and `java.lang.String` is mapped to IDL `wstring`.

RMI 11

The universal communications architecture allows Voyager programs to be both a universal client and a universal server by supporting simultaneous bi-directional communication with other CORBA, RMI, and DCOM[†] programs.

In this chapter, you will learn to:

- ◆ use Voyager as an RMI client
- ◆ use Voyager as an RMI server
- ◆ use the universal directory as an RMI Registry

†

Voyager 4, due later in 1999

Using Voyager as an RMI Client

Namespace interprets a URL that begins with “rmi:” as a reference to an RMI registry. For example, the following line returns a proxy to the object called “MyFoo” located in the RMI registry running on port 8000 of the host “dallas”. If the RMI registry is running on the default port (1099), the port number can be omitted.

```
IFoo foo = (IFoo) Namespace.lookup( "rmi://dallas:8000/MyFoo" );
```

Example

The [RMI Example 1](#) shows a Voyager RMI client looking up an object placed into an RMI registry by a SunSoft RMI server. The SunSoft RMI server must set its codebase property to a path containing the `examples.rmi` package.

Using Voyager as an RMI Server

To bind an object into an RMI registry, use the following format:

```
Namespace.bind( "rmi://dallas:8000/MyFoo", foo );
```

If the port number is omitted, it defaults to 1099. Voyager does not currently allow the class of `foo` to extend `UnicastRemoteObject`.

During the binding process, the RMI registry needs to resolve the class of the object that is being bound. Because this class is not available to the RMI registry through its local `CLASSPATH`, Voyager annotates it with the URL of an HTTP server that can serve up the class. This URL is set using `RmiRegistry.setServerCodeBase(String url)` and usually refers to an HTTP-enabled Voyager server. Start the server using the `-r` option or use `ClassManager.enableResourceServer()`.

Example

The [RMI Example 2](#) shows a SunSoft RMI client looking up an object placed into an RMI registry by a Voyager RMI server.

Using Universal Directory Integration

The universal directory can act as an RMI registry. Any Namespace or Naming binding/lookup operation that starts with “rmi:” and refers to a Voyager program is routed to that program’s universal directory.

Example

The [RMI Example 3](#) shows a SunSoft RMI client looking up an object placed into a Voyager universal directory.

DCOM 12

Component Object Model (COM) is the Microsoft standard that allows objects written in any language to locate and communicate with one another. Distributed Component Object Model (DCOM) is Microsoft's extension to COM. It allows objects distributed across a network to communicate using a language-neutral protocol called Object Remote Procedure Call (ORPC). For example, DCOM allows a C++ client running on a Windows 98 machine to communicate with a Java object running on a Windows NT server.

The Voyager-DCOM Bridge integrates Voyager and COM. It allows Voyager clients to access COM components. It also allows COM clients to access and send messages to Voyager objects.

The Voyager-DCOM Bridge uses Microsoft's Java/COM technology. Microsoft's Java/COM technology allows COM clients to send messages to Java objects by wrapping them in COM callable proxies. It also allows Java clients to send messages to COM objects by wrapping them in Java callable proxies. The Microsoft terminology for a COM callable proxy is a COM Callable Wrapper (CCW). The terminology for a Java callable proxy is a Java Callable Wrapper (JCW).

This chapter references several Microsoft utility programs (e.g. OLEVIEW, REGSVR32, DCOMCNFG, JACTIVEX). These utility programs are part of the Microsoft Platform Software Development Kit (SDK).

In this chapter, you learn to use the Voyager DCOM Bridge to:

- ◆ communicate from java clients to COM components

- ◆ communicate from COM clients to java objects
- ◆ troubleshoot DCOM

Communicating from Voyager Client to COM Component

To use the Voyager DCOM Bridge to access a COM object in a Voyager server from a Voyager client, use the following procedure.

Example

The [DCOM Example 2](#) illustrates the use of this procedure.

Step 1. Register the COM component with the Microsoft Windows Registry. Reference the Microsoft Windows Platform SDK for documentation on how to register COM components.

Step 2. Use Microsoft's JACTIVEX to create @COM annotated Java interfaces and @COM annotated JCWs from the component's type library. When compiled with Microsoft's VJ++ compiler, the @COM annotations are converted to class attributes. Microsoft's JVM uses these attributes at runtime to convert Java messages to COM messages. Note that the @COM annotations are ignored by non-Microsoft compilers. Also note the corresponding class attributes are ignored by non-Microsoft JVMs.

Step 3. Implement a Voyager COM server. For example, the following code shows how to bind the name MyGorilla to an instance of a Gorilla JCW to a Voyager Namespace on a Voyager server running at port 7000.

```
import com.objectspace.voyager.Voyager;  
import com.objectspace.voyager.Namespace;  
import com.objectspace.voyager.com.COM;  
  
public class YourCOMServer  
{  
    public static void main( String[] args )  
    {  

```



```

try
{
    COM.initialize();
    Voyager.startup( "7000" );
    Namespace.bind( "MyGorilla", new GorillaJCW() );
}
catch( Exception exception )
{
    System.out.println( exception.getMessage() );
}
}
}

```

In this example, a single instance of the Gorilla COM component is shared by all Voyager clients. The instance is created at the time the Gorilla's JCW is created. While this is the behavior Voyager programmers expect, it deviates from the COM programming paradigm. In COM, component activation is deferred until the time a client chooses to create an instance of (bind to) the object. Also activation policy is delegated to the object's class factory. For example, the class factory may choose to implement a singleton creation policy, or it may choose to share objects from a dynamically managed pool. To accommodate COM programmers who prefer this behavior, use the following code. Note the additional import statement and the modification to the `Namespace.bind` statement:

```

import com.objectspace.voyager.Voyager;
import com.objectspace.voyager.Namespace;
import com.objectspace.voyager.com.COM;
import com.objectspace.voyager.directory.ClassFactory;

public class YourCOMServer
{
    public static void main( String[] args )
    {
        try
        {
            COM.initialize();
            Voyager.startup( "7000" );
            Namespace.bind( "MyGorilla", new ClassFactory( "GorillaJCW" ) );
        }
        catch( Exception exception )
        {
            System.out.println( exception.getMessage() );
        }
    }
}

```

```
}  
}  
}
```

Step 4. Implement the Voyager client. For example, the following code shows how to implement a Voyager client to access an instance of a Gorilla COM component bound to the name MyGorilla on a Voyager server running at port 7000 on the machine MyServer.

```
import com.objectspace.voyager.Voyager;  
import com.objectspace.voyager.Namespace;  
import com.objectspace.voyager.com.COM;  
  
public class YourCOMServer  
{  
    public static void main( String[] args )  
    {  
        try  
        {  
            COM.initialize();  
            Voyager.startup( "7000" );  
            IApe ape = (IApe)Namespace.lookup( "//MyServer:7000/MyGorilla" );  
            ape.Swing();  
        }  
        catch( Exception exception )  
        {  
            System.out.println( exception.getMessage() );  
        }  
    }  
}
```

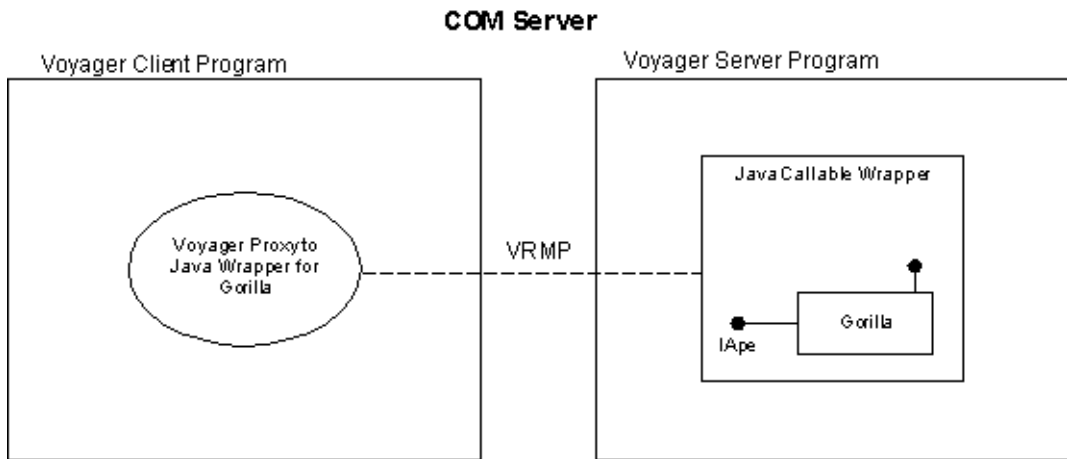


Figure 5

"Figure 5" shows the runtime state of a Voyager client program accessing a COM implemented Gorilla object via the Voyager Namespace. The COM server hosts an instance of the JCW for the Gorilla object. The JCW wraps the Gorilla COM component. The COM object may be instantiated in the COM server process or a different process depending upon the COM registry settings. The Voyager client program hosts a Voyager proxy to the remote JCW for the Gorilla object. The client sends messages to the Voyager proxy via the Gorilla object's IApe interface. Messages sent to the Gorilla object use the VRMP protocol instead of ORPC because Gorilla is hosted in a Voyager server.

Constraints

The current implementation of the Voyager-DCOM Bridge imposes the following constraints.

- ◆ Parameters and return values must be automation types. Automation types include short, long, double, float, string, bool, date, currency, Variant, SafeArray, IUnknown, and IDispatch. Variants and SafeArrays cannot contain object references (e.g. IUnknown or IDispatch).
- ◆ Values for parameters defined as [out] or [in, out] in Microsoft's IDL are not marshaled back to the client.

Consider the following scenario. A COM component implements an interface that defines a method ReverseString which has a single IDL defined [in, out] parameter

called `MyString`. A Voyager client initializes a string to `ABC` and calls `ReverseString`. The COM component reverses the marshaled value of the string to `CBA`. When control returns to the client, the value of the string is still `ABC`.

Voyager uses the VRMP protocol for remoting COM messages. The VRMP protocol does not marshal return values of `[out]` or `[in, out]` parameters as specified by DCOM.

- ◆ COM identity is not preserved.

Consider the following scenario. A COM component supports event listeners. A Voyager client successfully adds itself as an event listener. Subsequently, the client attempts to remove itself as an event listener. The COM component is unable to locate the client in its collection of event listeners and the call fails.

Voyager creates a unique proxy each time object references are passed as parameters to remote objects. In this scenario, the client is passed as a parameter to the COM component two times. Voyager creates one proxy for the call to `AddEventListener`, and a second proxy for the call to `RemoveEventListener`. The Microsoft JVM wraps each proxy in a CCW. Each CCW has its own identity.

A work-around for this constraint is to add a method to the event listener interface that tests for equality. In the implementation of `RemoveEventListener`, use the equality test rather than COM identity to test for membership in the event listener collection. This results in an extra network roundtrip each time the equality test is made.

- ◆ Client identity is lost.

Consider the following scenario. A COM component is configured to grant anyone in the Windows Group named Project Managers access permission. User Bill is in the Project Managers group and is logged on to the client machine. Bill binds to a COM component via the Voyager-DCOM Bridge and sends a message to it. He receives an access denied error message.

By the time COM gets an opportunity to check who is trying to access the component, the client's identity is lost. The message is sent from the Voyager proxy to the JCW using the VRMP protocol. The VRMP protocol does not propagate client identity as specified by DCOM. Loss of client identity breaks the programming model recommended by Microsoft, in which security is enforced by the business logic (a.k.a. the component) tier. Microsoft Transaction Manager (MTS) managed components use this model.

- ◆ COM causality is not preserved.

Consider the following scenario. A typical use of causality in COM is for single threaded apartment (STA) components to install a message filter which rejects top-level callpending (non-reentrant) calls, and accepts only nested (reentrant) calls. By default COM accepts both types of calls. DCOM generates and propagates a causality ID that uniquely identifies a call chain. This ID can be used to distinguish nested from top-level callpending calls.

Voyager uses the VRMP protocol for remoting COM messages. The VRMP protocol does not propagate causality as specified by DCOM. Therefore, a COM implemented STA component that installs a message filter to accept only nested calls will reject all calls. It will see all calls as top-level callpending calls. Reference *Essential COM* by Don Box for additional information on STA components and message filters.

- ◆ Proxies need to be statically created and installed in the client's classpath for clients that access COM components that use either Variants or SafeArrays.
- ◆ COM implemented STA components cause deadlock for reentrant calls.

Consider the following scenario. A COM component supports event listeners. A Voyager client adds itself as an event listener. The client sends a message to the component. The message results in the creation of an event object. The event object's threading model is also STA, therefore the event object is created in the same apartment as the COM component. Upon being notified of the event, the client asks the event object for its description. This results in a reentrant call to the apartment causing the server to deadlock.

Voyager uses the VRMP protocol for remoting the event notification message. Notification occurs on the thread associated with the STA that houses both the component and the event. Normally, cross-apartment COM calls trigger the STA to start a process that results in the apartment's thread becoming available to handle reentrant calls. The VRMP protocol doesn't trigger this process. Therefore, the thread associated with the apartment blocks and the reentrant call causes deadlock. Reference *Essential COM* by Don Box for additional information on COM's threading model.

- ◆ Unique implementations for the same method defined by multiple COM interfaces implemented by the same COM component are not preserved.

Consider the following scenario. A COM component Foo implements interfaces IThis and IThat. Both interfaces declare the method GetNumberOfTimesCalled. Foo provides a unique implementation for each interface that appropriately returns the number of times the IThis and IThat methods have been called. A Voyager client binds to an

instance of a Foo object, and subsequently calls `GetNumberOfTimesCalled` through `IThis` and `IThat` interfaces. Although one of Foo's implementations will be called in each case, it is undefined as to which implementation will be called.

- ◆ Clients running in non-Microsoft Java VMs need access to Microsoft defined classes in the `com.ms.com` package.

The JCW classes generated using `jactivex` refer to Microsoft classes such as `com.ms.com.IUnknown` and `com.ms.com._Guid`. These classes are installed as part of the MSVM. They are contained in one of the zip files located in the `<WINDOWS_DIR>\Java\Packages` directory. Unfortunately, the name of the zip file varies from machine to machine. They are typically contained within the largest zip file in the specified folder. During installation of Voyager, all classes in the `com.ms.com` package should be manually extracted from the zip file and inserted into a new zip file. This new zip file should be added to the CLASSPATH of Voyager clients not using a Microsoft VM.

Communicating from COM Client to Voyager Object

The Voyager-DCOM Bridge allows a COM client to access a Voyager object and is implemented in the COM co-class `VoyagerLib.Bridge`. The Voyager DCOM Bridge implements the COM dual interface `INamespace`, and it is packaged in the in-proc server `VoyagerBridge.DLL`.

The following procedure explains how to implement and deploy a simple application that allows a Visual Basic client to access an instance of a Gorilla object bound to the name `MyGorilla` on a Voyager server running at port 7000 on a machine named `MyServer`.

Step 1. Implement the Voyager server. The following code shows how to bind an instance of a Java implemented Gorilla object with the Voyager Namespace on a Voyager server running at port 7000. For additional information on how to use the Voyager Namespace, reference the *Voyager ORB Developer Guide*. Once the server has been implemented and started, the Voyager object is available for use by COM clients.

```
import com.objectspace.voyager.Voyager;  
import com.objectspace.voyager.Namespace;
```

```

public class YourServer
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup( "7000" );
            Namespace.bind( "MyGorilla", new Gorilla() );
        }
        catch( Exception exception )
        {
            System.out.println( exception.getMessage() );
        }
    }
}

```

Step 2. Implement the client. Prior to implementing the client, you must choose whether you will use vtable or late binding to access the Voyager-DCOM Bridge. It is not difficult to rebuild the client if you subsequently change your mind.

Vtable binding maximizes performance. However, it requires the runtime availability of type library information on the client machine. Type library information is embedded in the in-proc server VoyagerBridge.DLL. It is also packaged separately in VoyagerBridge.TLB. To declare and create an instance of the Bridge from Visual Basic using vtable binding, select Voyager Bridge from the Project References menu and include the following code:

```

Dim Namespace as VoyagerLib.INamespace
Set Namespace = New VoyagerLib.Bridge

```

Late binding minimizes client configuration. It does not require the runtime availability of type library information on the client machine. In Visual Basic, late binding is implied whenever the Object data type is used. To declare and create an instance of the Bridge from Visual Basic using late binding, include the following code:

```

Dim Namespace as Object
Set Namespace = CreateObject( "VoyagerLib.Bridge" )

```

For additional information on vtable and late binding, reference *Programming Distributed Applications with COM and Microsoft Visual Basic 6.0* by Ted Pattison.

The following code shows how to implement a COM client in Visual Basic using vtable binding to access an instance of a Gorilla object. This instance is assumed to be bound to the name MyGorilla on a Voyager server running at port 7000 on the machine MyServer.

```
Sub AccessJavaObjectViaNamespace ()
    Dim Namespace As VoyagerLib.Inamespace
    Set Namespace = New VoyagerLib.Bridge

    Dim Gorilla As Object
    Set Gorilla = Namespace.Lookup( "//MyServer:7000/MyGorilla" )

    Gorilla.Swing

End Sub
```

Step 3. Deploy the client application. Voyager supports two Voyager-DCOM Bridge configurations: local and remote.

The local configuration maximizes performance. However, it requires the installation of the Bridge on each client machine. In this configuration, an instance of the Bridge is loaded in the client's process and accessed directly.

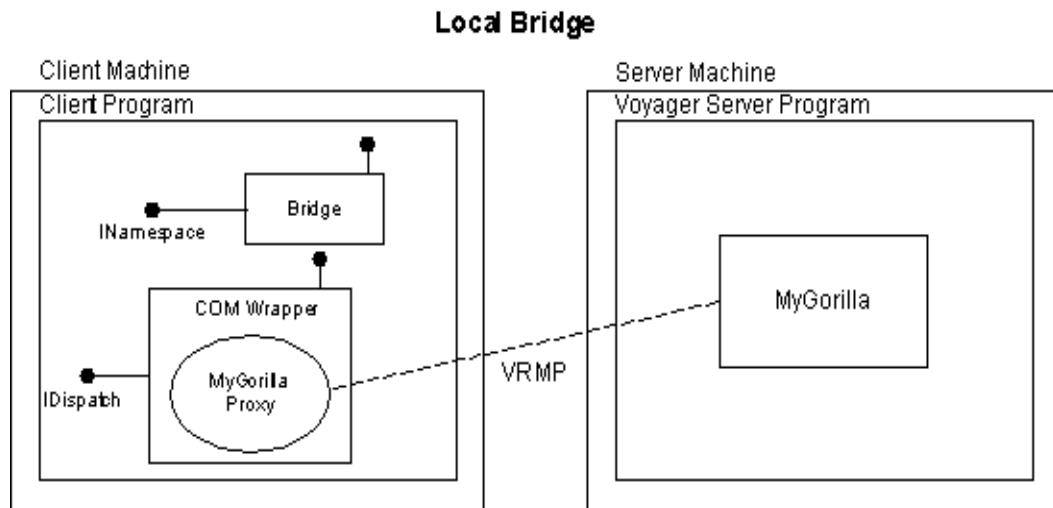


Figure 6

"Figure 6" shows the runtime state of a COM client program using the local bridge configuration to access an instance of a Gorilla object. The Bridge is loaded in the client program's memory space. The client program uses the INamespace interface of the Bridge to bind to the remote Gorilla object. The Bridge creates a Voyager proxy to the Gorilla object. The Microsoft JVM wraps the Voyager proxy with a COM Callable Wrapper (CCW). The CCW exposes the IApe methods implemented by Gorilla through an IDispatch interface. Messages sent to the Gorilla object use the VRMP protocol. Note the absence of ORPC messages. Also note that the Bridge, Voyager, and the Microsoft JVM need to be installed on the client machine.

The remote configuration minimizes client configuration. It allows the Bridge to be installed on a single remote machine. The bridge machine may be shared by multiple clients. In this configuration, the Bridge is hosted by a surrogate process, and accessed via DCOM.

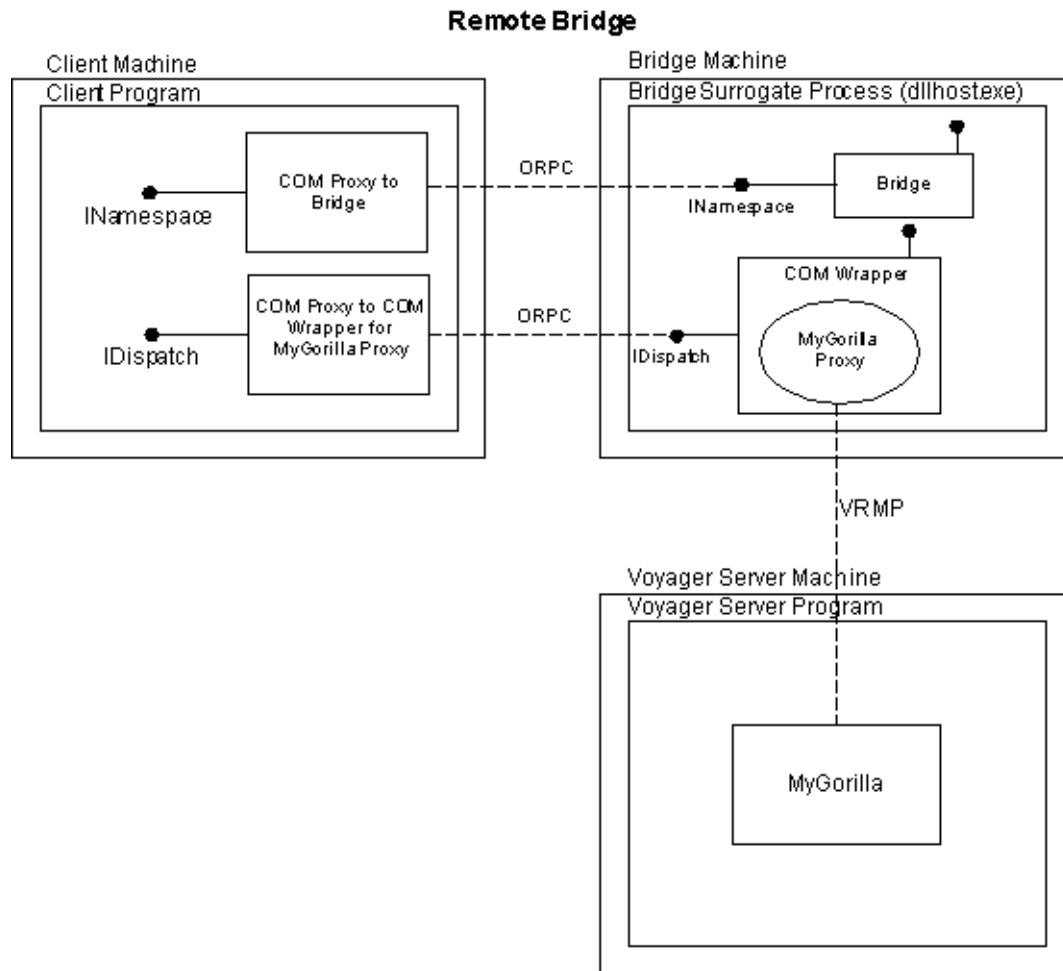


Figure 7

"Figure 7" shows the runtime state of a COM client program using the remote bridge configuration to access an instance of a Gorilla object. The Bridge is loaded in the memory space of the surrogate process `dllhost.exe` running on the bridge machine. The client program uses the `INamespace` interface of the Bridge to bind to the remote Gorilla object. The Bridge creates a Voyager proxy to the Gorilla object. The Microsoft JVM wraps the Voyager proxy with a CCW. The CCW exposes the `IApe` methods implemented by Gorilla through an `IDispatch` interface. A COM proxy to the CCW is

unmarshaled in the client program's memory space. Messages sent to Gorilla object use the VRMP protocol. Note ORPC messages between the client and objects accessed through the Bridge. Also note that the Bridge, Voyager, and the Microsoft JVM need not be installed on the client machine.

Configuring the Local Bridge

Step 1. Install the Voyager-DCOM Bridge on the client machine by entering the following command at the DOS prompt.

```
> REGSVR32 VoyagerBridge.dll
```

Step 2. Use OLEVIEW to test whether registration is successful. Start OLEVIEW and select the CoCreateInstance Flags item from the Object menu.

Step 3. Verify CLSCTX_INPROC_SERVER is selected.

Step 4. Use the explorer interface to locate the Voyager Bridge component.

Step 5. Click the left mouse button on the plus sign to the left of the component name.

If registration is successful, OLEVIEW will expand a list of interfaces supported by the Bridge. OLEVIEW generates this list by creating an instance of the Bridge, and querying for each interface specified in the registry under the HKEY_CLASSES_ROOT\Interfaces key.

Configuring the Remote Bridge

Start by installing the Voyager-DCOM Bridge on the bridge machine as described in the Local Bridge Configuration section. Once the Bridge has been installed on the remote bridge machine, three additional configuration steps are required. The first two steps are performed on the bridge machine. The final step is performed on the client machine.

1. Configure surrogate process
2. Configure bridge security
3. Configure client for remote bridge

Configure Surrogate Process

The first step is to configure `dllhost.exe` to be launched as the surrogate process for the Bridge. This is necessary because the Bridge is packaged as an in-proc server.

Step 1. Start OLEVIEW and use its explorer interface to select the Voyager Bridge component.

Step 2. Select the Implementation tab.

Step 3. Select the Inproc Server tab.

Step 4. Check the Use Surrogate Process box.

Use OLEVIEW to test whether this step was successful.

Step 5. Start OLEVIEW and select the CoCreateInstance Flags item from the Object menu.

Step 6. Verify `CLSCTX_LOCAL_SERVER` is the first item selected.

Step 7. Repeat the test described in the Local Bridge Configuration section. Note that this time success results in the launching of `dllhost.exe` as evidenced by running Task Manager on an NT machine.

Configure Bridge Security

The next step is to configure security for the Bridge.

Step 1. Start DCOMCNFG and select the Voyager Bridge component.

Step 2. Select the Security tab.

Step 3. Specify the desired custom access and launch permissions.

Step 4. Select the Identity tab.

Step 5. Specify the desired user account used to run the Bridge. It is typically recommended to select “This user”, and enter an appropriate user account. This allows the Bridge to execute with known security privileges, regardless of the client. It also allows the Bridge to execute whether or not a user is logged in on the bridge machine.

Use OLEVIEW to test whether this step was successful.

Step 6. Repeat the steps used to verify whether the surrogate process was successfully specified.

Step 7. Log in as a trusted user and verify that you are able to use the Bridge.

Step 8. Log in as an untrusted user and verify that you are NOT able to use the bridge.

Configure Client for Remote Bridge

The final step is to configure the client machine to point to the remote bridge machine. A registry script file `VoyagerBridge.REG` is bundled with Voyager. It can be used to configure the client's registry to point to the remote bridge machine.

Step 1. For vtable binding, edit the script to point to the actual runtime location of the type library `VoyagerBridge.TLB`.

Step 2. For late binding, delete the line in the script file that specified the location of the type library.

Step 3. In either case, edit the script to point to the actual bridge machine

Use OLEVIEW to test whether this step was successful.

Step 4. Start OLEVIEW and select the `CoCreateInstanceFlags` item from the Object menu.

Step 5. Verify `CLSCTX_REMOTE_SERVER` is the first item selected.

Step 6. Repeat the steps used to verify whether the surrogate process was successfully specified. If not already launched, the surrogate process `dllhost.exe` is launched on the bridge machine.

Step 7. Remember to log in on the client machine as a trusted user.

Run the client.

Example

The [DCOM Example 1](#) demonstrates the use of the Voyager-DCOM Bridge to access Voyager objects from COM clients.

Constraints

The current implementation of the Voyager-DCOM Bridge imposes the following constraints.

- ◆ Parameters and return values must be automation compatible types. Automation types include short, long, double, float, string, bool, date, currency, Variant, SafeArray, IUnknown, and IDispatch. Java objects are also supported for the local bridge configuration.
- ◆ Interface classes for the objects accessed via the bridge need to be in the CLASSPATH on the bridge machine.

Troubleshooting

The following table describes some common error messages you may encounter.

Error	Message	Problem	Solution
429	ActiveX component can't create object	Bridge not installed on bridge machine	For in-proc configuration, use regsvr32 to register VoyagerBridge.dll. For the remote bridge, use VoyagerBridge.reg file to configure registry.
463	Class not registered on local machine	Bridge not registered on client machine to point to remote bridge.	Use VoyagerBridge.reg to configure client machine for remote bridge configuration.
7	Out of memory	Microsoft Java not properly installed on bridge machine	
800706BE	Automation error. The remote procedure call failed (Sometimes Out of memory - Error 7)	Surrogate process (dllhost.exe) not activated on remote bridge	Use OLEVIEW or application-specific means to activate remote bridge.
70	Permission denied	Client not configured for launch\access permission on remote bridge	Use DCOMCNFG Security tab to configure launch\access permissions.

Error	Message	Problem	Solution
462	The remote server machine does not exist or is not unavailable	Incorrect host name specified for remote bridge (via OLEVIEW, DCOMCNFG, or CreateObject)	For declarative configuration, use OLEVIEW or DCOMCNFG to specify a valid host name. For programmatic, specify a valid host name as an argument to CreateObject.
429	ActiveX component can't create object	Wrong ProgID passed to CreateObject	Use VoyagerLib.Bridge for ProgID.
7	Out of memory	Client configured to run remotely. Attempted to create in-proc.	
Error Calling Lookup on Bridge			
513	Error starting Voyager: java.lang.ClassNotFoundException	Voyager not in class path on bridge machine	Verify Voyager is in class path on bridge machine. Note: if you have specified the class path via the user (as opposed to system) CLASSPATH environment variable, the MS JVM only picks up the settings if you are running the Bridge as the launching user.
514	RuntimeException(</host:port>;java.lang.ClassNotFoundException: Your.Object__Proxy)	Byte code for Directory proxy not available on bridge machine	Use PGEN to statically generate a Directory Proxy class and make sure it is available in the class path.

Error	Message	Problem	Solution
514	RuntimeRemoteException(</host:port>;java.lang.ClassNotFoundException:Your.Object__Proxy)	Bridge machine configured to access Internet via proxy server (MS JVM and our proxy server have problem - SUN JVM worked fine)	Launch Internet Explorer and select Internet Options from the View menu to access the dialog box. On the Connection tab, make sure the 'Access the Internet using a proxy server' check box is not selected.
514	RuntimeRemoteException(</host:port>;java.lang.ClassNotFoundException:Your.Object__Proxy)	Server didn't call ClassManager.enableResourceServer, or federated server didn't call VoyagerClassLoader.addURLResource	For security reasons, Voyager servers need to be explicitly enabled as resource loaders and Voyager clients need to grant permission to URL resource loaders. A federated server acts as both a server and a client.
515	RuntimeRemoteException(java.net.ConnectException:Connection refused)	Voyager not running at valid computer\port specified in Bridge.Lookup	Make sure the computer and port have been correctly specified. If they have been, make sure a Voyager server is running at that location.
516	RuntimeRemoteException(java.net.UnknownHostException:Server name:BadHostName)	Invalid machine name specified in Bridge.Lookup	Specify a valid machine name.
517	com.objectspace.voyager.NameSpaceException:no object bound to the name vdir://host:port/YourFederatedServerName/YourObjectName;	Unknown object name specified in Bridge.Lookup for target Voyager server	Make sure the object name is spelled correctly.

Error	Message	Problem	Solution
Error Messaging Voyager Object			
80004005	java.lang.IncompatibleClassChangeError	Unable to pass java objects (created with java moniker) via remote bridge	Use in-proc bridge if you need to pass java objects.

Dynamic XML

13

XML (extensible markup language) is the W3C standard for information exchange on the Internet. XML simplifies information exchange between applications and provides great flexibility in how information can be presented and used by end users.

Dynamic XML (DXML) radically simplifies XML development by allowing developers to create, read and write XML documents as if they were regular Java objects. Without DXML, developers face a much larger learning curve and must master the intricacies of low-level XML technology like parsers, Document Type Definitions (DTD) and the Document Object Model (DOM).

In this chapter, you learn to:

- ◆ understand XML
- ◆ manipulate XML (the old way) using DOM
- ◆ manipulate XML (the new way) using DXML
- ◆ understand how DXML maps DTDs to Java

Note: DXML currently requires IBM's XML parser. You can download a copy of the parser at <http://www.alphaworks.ibm.com/tech/xml4j>. DXML has been tested with versions 1.1.16 and 2.09 of the XML parser; however, any version from 1.1 and higher is compatible with DXML.

Understanding XML

XML is a markup language that allows you to store data in a structured way that is independent of any particular presentation of the data. XML information is typically stored in three different kinds of documents:

- ◆ *Document Type Definitions (DTD)*, which describe the grammar of one or more XML elements.
- ◆ *XML Data Documents (XML)*, which contain the data associated with one or more instances of XML elements.
- ◆ *Extensible Style Documents (XSL)*, which describe how the contents of an XML data document should be rendered to a particular display device such as a browser.

For example, to store phonebook information using XML, first create a DTD that describes the XML elements related to phonebook entries. In the following file, * means zero or more entries, + means one or more entries, and #PCDATA means “parsed character data”.

phonebook.dtd

```
<?xml encoding="US-ASCII"?>
<!ELEMENT phonebook (name)*>
<!ELEMENT name (firstname,lastname,phonenumber+) >
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT phonenumber (#PCDATA)>
```

Next, create an XML data document that starts with a DOCTYPE reference to the corresponding DTD and is followed by instances of the declared XML elements. Note that a DOCTYPE reference must specify the root element type of the document. For example:

phonebook.xml

```
<?xml version="1.0"?>
<!DOCTYPE phonebook SYSTEM "phonebook.dtd">
<phonebook>
  <name>
    <firstname>James</firstname>
    <lastname>Johnson</lastname>
    <phonenumber>8675309</phonenumber>
```

```
</name>  
</phonebook>
```

Finally, create zero or more XSL documents that specify how the XML data should be displayed to a particular target device. An example of an XSL document is not shown here because the focus of this chapter is on reading and writing XML data, not on its presentation.

Manipulating XML Using DOM

The contents of an XML document can be read using an XML parser and manipulated using the standard low-level Document Object Model (DOM) API. DOM requires you to navigate XML documents in terms of nodes and returns XML data as strings.

The following example uses an XML parser and DOM to read the `phonebook.xml` file from the previous section and display the node contents in a hierarchical fashion.

PhonebookDOM.java

```
import org.w3c.dom.*; // standard DOM API
import com.ibm.xml.parsers.*; // IBM XML parser

public class PhonebookDOM
{
    public static void main( String[] args )
    {
        try
        {
            DOMParser parser = new DOMParser();
            parser.parse( "phonebook.xml" );
            Document document = parser.getDocument(); // root of document
            display( 0, 0, document ); // display nodes recursively
        }
        catch( Exception exception )
        {
            exception.printStackTrace();
        }
    }

    public static void display( int level, int child, Node node )
    {
        if( node == null )
            return;

        for( int i = 0; i < level; i++ ) // indent
            System.out.print( " " );

        // display node
        System.out.print( level + "." + child + ": " );
```

```

System.out.println( node.getNodeName() + " = " + asString( node.getNodeValue() ));
NodeList list = node.getChildNodes();

// display children
for( int i = 0; i < list.getLength(); i++ )
    display( level + 1, i, list.item( i ) );
}

public static String asString( String string )
{
    return string == null ? "<null>" : string.replace( '\n', ' ');
}
}

```

The output of this program looks like this:

```

>java PhonebookDOM
0.0: #document = <null>
1.0: phonebook = <null>
1.1: phonebook = <null>
2.0: #text =
2.1: name = <null>
3.0: #text =
3.1: firstname = <null>
4.0: #text = James
3.2: #text =
3.3: lastname = <null>
4.0: #text = Johnson
3.4: #text =
3.5: phonenumber = <null>
4.0: #text = 8675309
3.6: #text =
2.2: #text =
2.3: name = <null>
3.0: #text =
3.1: firstname = <null>
4.0: #text = Bert
3.2: #text =
3.3: lastname = <null>
4.0: #text = Jenner
3.4: #text =
3.5: phonenumber = <null>

```

```
4.0: #text = 1348924
3.6: #text =
3.7: phonenumber = <null>
4.0: #text = 1462345
3.8: #text =
2.4: #text =
```

As you can see, DOM is a low-level API and only allows values to be manipulated by traversing and reading/writing nodes. Because of this, the following problems occur when attempting to manipulate XML using DOM:

- ◆ the code does not clearly show which fields are being accessed
- ◆ the node structure requires a program to carefully track nesting levels
- ◆ fields are always accessed as strings, so the code is not typesafe
- ◆ programs tend to break easily when a DTD is modified
- ◆ operations that modify the document are complex to write and debug

In addition, DOM cannot be used for reading remote XML documents. All of these problems are solved using DXML.

Manipulating XML using DXML

Dynamic XML allows you to read and write XML documents as if they were regular Java objects. Programs written using DXML instead of DOM are smaller and much easier to develop. In addition, you do not have to remember the level of depth in the document tree and the context of the current location.

To manipulate XML using DXML, first run the `xgen` utility on the DTDs (wildcards are supported). `xgen` generates Java interfaces for every element defined in the DTD. The name of an interface's package derives from the name of the DTD file. `xgen` also creates the following special files in the package directory:

- ◆ `lEntities.java` - contains a list of constants that represent the entities declared in the DTD. The constants are defined as public static final `String` fields and have the same names as the entities in the original DTD document. This file is not generally used by developers, but it may become useful if you want to take advantage of the XML entities declared in the source DTD document.
- ◆ `DocumentTypeInfo.java` - used internally by the DXML runtime engine and not used by developers.

For example, to run `xgen` on the `phonebook.dtd` file and place the resultant Java interfaces into the "phonebook" package, type:

xgen phonebook.dtd

Each Java interface includes set and get methods for the subelements, allowing you to easily navigate in a typesafe way from the document's root interface down into any nested element. Here are the interfaces created from `phonebook.dtd`.

lname.java

```
package phonebook;
```

```
public interface lname
{
    /* Method declarations for lastname element */
    public void setlastname( java.lang.String arg1 );
    public java.lang.String getlastname();
    /* Method declarations for firstname element */
    public void setfirstname( java.lang.String arg1 );
    public java.lang.String getfirstname();
}
```

```

/* Method declarations for phonenumber element */
public void setphonenumbers( java.lang.String[] arg1 );
public java.lang.String[] getphonenumbers();
public java.util.Enumeration getphonenumberElements();
public java.lang.String getphonenumberAt( int arg1 ) throws java.lang.ArrayIndexOutOfBoundsException;
public void insertphonenumberAt( java.lang.String arg1, int arg2 ) throws
    java.lang.ArrayIndexOutOfBoundsException;
public boolean removephonenumber( java.lang.String arg1 );
public boolean removephonenumberAt( int arg1 ) throws java.lang.ArrayIndexOutOfBoundsException;
}

```

lphonebook.java

package phonebook;

```

public interface lphonebook
{
    /* method declarations for phonebook element */
    public phonebook.lname[] getnames();
    public void setnames( phonebook.lname[] arg1 );
    public java.util.Enumeration getnameElements();
    public phonebook.lname getnameAt( int arg1 ) throws java.lang.ArrayIndexOutOfBoundsException;
    public void insertnameAt( phonebook.lname arg1, int arg2 ) throws
        java.lang.ArrayIndexOutOfBoundsException;
    public boolean removenname( phonebook.lname arg1 );
    public boolean removennameAt( int arg1 ) throws java.lang.ArrayIndexOutOfBoundsException;
    public boolean removeAllnames();
    public phonebook.lname newname();
}

```

A complete description of how DXML maps DTDs to Java is included later in this chapter.

Once the interfaces corresponding to the elements in the DTD have been created, the following static methods in `com.objectspace.xml.Xml` may be used to open an existing XML document or create a new XML document that uses the DTD. Both methods return an instance of `com.objectspace.xml.IXml` that represents the XML document.

- ◆ `openDocument(input)` - return an `IXml` corresponding to the existing XML document, where `input` can be a `java.io.InputStream`, a `java.io.File`, a `java.net.URL` or a `String` that is the XML document itself. Note that this XML document must have a document type declaration with an external reference to the DTD from which the document derives.

- ◆ `newDocument(File file, String root)` - return an `IXml` corresponding to a new XML document that uses the DTD specified by the file parameter and whose root element is specified by the root parameter.

The following methods declared in `IXml` allow you to navigate the XML document and save its contents:

- ◆ `getRoot()` - return an Object reference to the root of the XML document. To navigate through the document, cast this reference to the appropriate Java interface that was generated by `xgen`.
- ◆ `saveDocument(output)` - save the XML document, where output can be a `java.io.OutputStream` or a `java.io.File`.

The following example shows how the `phonebook.xml` file from the first section can be read, modified and then written as if it were a regular Java object.

PhonebookDXML.java

```
import java.io.*;
import com.objectspace.xml.*; // DXML library
import phonebook.*; // generated by xgen

public class PhonebookDXML
{
    public static void main( String[] args )
    {
        try
        {
            IXml xml = Xml.openDocument( new File( "phonebook.xml" ) ); // read file
            Iphonebook phonebook = (Iphonebook) xml.getRoot(); // root

            // display original contents
            display( "original", phonebook );

            // insert a new name and display new contents
            Iname name = phonebook.newname();
            name.setlastname( "Glass" );
            name.setfirstname( "Graham" );
            name.setphonenumbers( new String[] { "1342567" } );
            phonebook.insertnameAt( name, 0 );
            display( "after insert", phonebook );

            // remove a previous name and display new contents
```

```

        phonebook.removeNameAt( 1 );
        display( "after remove", phonebook );

        // save new XML data to new file
        xml.saveDocument( new File( "phonebook2.xml" ) );
    }
    catch( Exception exception )
    {
        exception.printStackTrace();
    }
}

public static void display( String title, IPhonebook phonebook )
{
    System.out.println( title + ":" );
    IName[] names = phonebook.getNames();

    for( int i = 0; i < names.length; i++ )
    {
        IName name = names[ i ];
        System.out.print( name.getFirstName() + " " + name.getLastName() + ": " );
        String[] numbers = name.getPhoneNumbers();

        // display all phone numbers
        for( int j = 0; j < numbers.length; j++ )
            System.out.print( numbers[ j ] + " " );

        System.out.println();
    }

    System.out.println();
}
}

```

The output of this program would look like this:

```

>java PhonebookDXML
original:
James Johnson: 8675309
Bert Jenner: 1348924 1462345

after insert:

```

Graham Glass: 1342567
James Johnson: 8675309
Bert Jenner: 1348924 1462345

after remove:

Graham Glass: 1342567
Bert Jenner: 1348924 1462345

Here are the contents of the newly created phonebook2.xml:

phonebook2.xml

```
<?xml version="1.0"?>
<!DOCTYPE phonebook SYSTEM "phonebook.dtd" >
<phonebook>
  <name>
    <lastname>Glass</lastname>
    <firstname>Graham</firstname>
    <phonenumber>1342567</phonenumber>
  </name>
  <name>
    <firstname>Bert</firstname>
    <lastname>Jenner</lastname>
    <phonenumber>1348924</phonenumber>
    <phonenumber>1462345</phonenumber>
  </name>
</phonebook>
```

For more complete and elaborate examples see the end of the chapter.

How DXML Maps DTDs to Java

This section provides a list of the rules xgen uses when generating Java interfaces from DTDs, divided into the following topics:

- ◆ basic rules
- ◆ containment constraint rules
- ◆ compound element rules
- ◆ attribute rules

Basic rules

1. *Any element whose content model differs from ANY, EMPTY, and #PCDATA will become a Java interface.*

Since the elements with content model as ANY, EMPTY or #PCDATA are the containers of the actual data, there is no need to create a Java interface for these kind of elements. A simple rule is that any element containing other elements will become an interface.

<!ELEMENT B (A,C)> - Element B will become an interface, it contains A and C
<!ELEMENT A (#PCDATA)> - Element A will not be an interface, it carries text data
<!ELEMENT C (D*)> - Element C will become an interface, it contains 0 or more D
<!ELEMENT D ANY> - Element D will not be an interface, it carries any kind of data

2. *The name of a generated Java interface is comprised of the letter 'I' plus the element name.*

To comply with ObjectSpace coding standards, the names of the interfaces are prefixed with “I”. For example:

<!ELEMENT Foo (A,C)> generates public interface IFoo { }
<!ELEMENT Bar (D*)> generates public interface IBar { }

3. *All interfaces generated from a DTD belong to a Java package. The package name is the name of the source DTD file.*

This is done to allow the DXML engine to locate proper interfaces at runtime. The name of the DTD file may uniquely identify the interfaces generated from it. As a result, you may have multiple DTDs describing the same data structures from different perspectives. The interface names may be the same, but the package names will be different.

address.dtd

```
<!ELEMENT Address (Name,Street,City,State,Zipcode)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Zipcode (#PCDATA)>
```

Given this DTD, the following interface is created (notice the package declaration):

```
package address;
```

```
public interface IAddress
{
    .....
    .....
}
```

4. *Any element whose content model is declared as #PCDATA is accessed as an instance of java.lang.String. There are at least 2 methods created, one method to set and the other to get the value. Additional methods may be created depending on the containment constraints applied. Any element whose content model is declared as Any is accessed as an instance of java.lang.Object.*

The #PCDATA content model usually defines character data, which maps to java.lang.String; therefore, in the implementation, the data is considered to be of type String. If there are containment constraints applied to an element, there may be methods added to access the element as an array, or as an optional element.

file.dtd

```
<!ELEMENT File (Name,Size)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Size (#PCDATA)>
```

xgen Output

```
package file;
```

```
public interface IFile
```

```
    public String getName ();
    public void setName ( String arg );
    public String getSize();
    public void setSize( String arg );
}
```

file.xml

```

<!DOCTYPE File SYSTEM "file.dtd">
<File>
  <Name>readme.txt</Name>
  <Size>1204</Size>
</File>

```

To access the data in file.xml

```

import file.IFile;
public void readFileData( IFile myFile )
{
  String fileName = myFile.getName();
  String fileSize = myFile.getSize();
  System.out.println( "File name - " + fileName );
  System.out.println( "File size - " + fileSize );
}

```

5. *If element A is declared as a child of element B, and element A will become a Java interface, there are at least 2 methods created to access A from B. Additional methods may be created depending on the containment constraints applied to the element.*
- From rule #1, you know that any element containing other elements will become a Java interface. Semantically, if element B contains element A, there should be a way to navigate to A through B. In DXML, this is accomplished with interfaces.

DTD

```

<!ELEMENT B (A)>
<!ELEMENT A (C,D)>
<!ELEMENT C (#PCDATA)>
<!ELEMENT D (#PCDATA)>

```

xgen Output

```

public interface IB
{
  public IA getA (); // need to have interface IB to get to IA
  public void setA ( IA arg );
}

public interface IA
{
  public String getC (); // need to have interface IA to get to C
  public void setC ( String arg ); // no need for interface IC, since C is declared as #PCDATA
}

```



```

public String getD ();
public void setD ( String arg ); // no need for interface ID, since D is declared as #PCDATA
}

```

6. *If there are two or more elements A in the declaration of element B, the IA access methods in IB interface will be suffixed with a number to distinguish specific element A. For any two elements, xgen generates the same methods with identical signatures. Java does not allow methods with the same signatures in the same class or interface. Therefore a number is added to methods generated for each element in the content model if it contains multiple instances of that element.*

DTD

```

<!ELEMENT B (A,A)>
<!ELEMENT A (#PCDATA)>

```

xgen Output

```

public interface IB
{
    public String getA1();
    public void setA1( String arg );
    public String getA2();
    public void setA2( String arg );
}

```

7. *The element content may consist of a series of content particles arranged either as a **choice** or as a **sequence**. A choice requires content to appear in your document that satisfies exactly one of the content particles. A sequence requires content to appear in your document that satisfies all of the content particles, in the order specified. If the content particles are arranged as a choice, a remove method is added for each content particle.*

DTD

```

<!ELEMENT A (B|C)>
<!ELEMENT B (#PCDATA)>
<!ELEMENT C (#PCDATA)>

```

xgen Output

```

public interface IA
{
    public String getB ();
    public void setB( String arg );
    public void removeB();
}

```

```

public String getC ();
public void setC( String arg );
public void removeC();
}

```

Containment constraint rules

Containment constraint rules are applicable to the elements declared with multiple or conditional containment. There are three occurrence indicators - *,+,?.

1. *For any content particle (either a single element type name or a group), if there is a * or + occurrence indicator attached, the content particle is treated as an array.*

DTD

```

<!ELEMENT Y(X)*>
<!ELEMENT X(#PCDATA)>

```

xgen Output

```

public interface IY
{
    public String[] getXs();
    public void setXs( String[] arg );
    public Enumeration getXElements();
    public String getXAt( int arg );
    public void insertXAt( String arg, int i );
    public boolean removeX(String arg );
    public void removeXAt( int arg1 );
    .....
}

```

2. *If the occurrence indicator is * or ?, a remove method is added to the generated methods. For the example above, the method would be:*

```

public void removeXs( String[] arg );

```

3. *If the occurrence indicator is ?, null will be returned from the getMethod if there are zero occurrences.*

Compound element rules

A compound element is a group of elements that may be logically treated as a single content particle. Usually a compound element consists of several content particles arranged either as a *choice* or as a *sequence* (see #7 from Basic rules).

These are examples of compound elements:

<!ELEMENT A (B,(C|D|E),F)>

<!ELEMENT message (header,(words,links)*,footer)>

1. *The name of a Java interface created to represent a compound element is comprised of the content particle names of which the compound element consists. The particle names are concatenated with each other with “Or” if the elements are arranged as a choice, or with “And” if the elements are arranged as a sequence.*

DTD

<!ELEMENT Name (Firstname,Lastname)*>

XGEN Output

```
public interface IName
{
    public IFirstnameAndLastname[] getFirstnameAndLastnames();
    public void setFirstnameAndLastnames( IFirstnameAndLastname[] arg );
    public Enumeration getFirstnameAndLastnameElements();
    public String getFirstnameAndLastnameAt( int arg );
    public void insertFirstnameAndLastnameAt( String arg, int i );
    public boolean removeFirstnameAndLastname( IFirstnameAndLastname arg );
    public void removeFirstnameAndLastnameAt( int arg );
}
```

Attribute Rules

XML attributes are name-value pairs attached to elements. For instance, a date attribute can be attached to a file element:

<file date="4/1/99" >...</file>

In a DTD, attributes are defined using the following syntax:

<!ELEMENT element-name ... >

<!ATTLIST element-name attribute-name attribute-type attribute-mode>

where attribute-type is one of:

- ◆ CDATA - string
- ◆ ID - must be unique within the document
- ◆ IDREF - must match an ID

- ◆ ENTITY - must match the name of a declared entity
- ◆ ENTITIES - collection of ENTITY
- ◆ NMTOKEN - restricted string
- ◆ NMTOKENS - collection of NMTOKEN
- ◆ NOTATION - must match the name of a declared notation
- ◆ <enumeration> - list of tokens

and attribute-mode is one of:

- ◆ #REQUIRED - the attribute must be present in the XML file
- ◆ #IMPLIED - the attribute may not be present in the XML file
- ◆ #FIXED - the definition of the attribute must include a default value; the attribute cannot be assigned a value other than the default in the XML file
- ◆ <omitted> - a default value must be provided; if the attribute is omitted, the XML document will be processed as if this attribute was set with this default value

Name-value pairs are represented in DXML using hash tables. If an element defines at least one attribute, xgen adds the following methods to the element's Java interface:

- ◆ *public Hashtable getAttributes()* - return a hash table whose keys are the attribute names and whose values are, for each attribute, the value in the XML file if set, otherwise the default value.
- ◆ *public void setAttribute(String name, String value)* - set the attribute with the specified name to the specified value. If no such attribute exists, throw a `NoSuchElementException`. If the attribute mode is #FIXED and the value is not the default value, throw a `FixedAttributeException`.
- ◆ *public boolean removeAttribute(String name)* - remove the attribute with the specified name. If no such attribute exists, throw a `NoSuchElementException`. If the attribute mode is #REQUIRED, throw a `RequiredAttributeException`. Return true if the attribute was successfully remove and false otherwise.

Note: Validity constraints related to entities, IDs and enumerations are not currently enforced. For instance, DXML does not disallow setting an enumerated attribute with a value that is not part of the declaration of the enumeration.

Timers 14

Voyager's timer Services include the Stopwatch and Timer classes. You can use a Stopwatch object to clock time intervals and print time measurement statistics. You can use a Timer object to generate timer events and add listeners to timers.

In this chapter, you will learn to:

- ◆ clock time intervals
- ◆ use timers and timer events

Clocking Time Intervals

Use Voyager's Stopwatch class to clock time intervals. You can start and stop a Stopwatch object an unlimited number of times before resetting it; every start/stop cycle is called a *lap*. You can access the cumulative lap time, average lap time, and last lap time, and you can record individual lap times.

Use the following methods defined in Stopwatch to clock time intervals:

- ◆ `getDate()`
Returns the current date.
- ◆ `getMilliseconds()`
Returns the current time in milliseconds since January 1, 1970, 00:00:00 GMT.
- ◆ `reset()`
Resets the stopwatch, clears lap times, and sets the lap count to zero.
- ◆ `start()`
Starts a stopwatch.
- ◆ `stop()`
Stops a stopwatch, increments the lap count, and, when enabled, records the lap time.
- ◆ `lap()`
Stops the stopwatch temporarily to record the lap time and immediately restart it.
- ◆ `setRecordLapTimes(boolean flag)`
Enables or disables the recording of lap times.
- ◆ `isRecordLapTimes()`
Returns a boolean indicating whether lap-time recording is enabled.
- ◆ `getLapCount()`
Returns the current completed lap count.
- ◆ `getLapTime()`
Returns the last completed lap time.

- ◆ `getLapTimes()`

Returns a long array of recorded lap times. If lap-time recording is disabled, an empty array is returned.

- ◆ `getTotalTime()`

Returns the sum of all completed lap times.

- ◆ `getAverageLapTime()`

Returns the average lap time.

Example

The [Stopwatch1 Example](#) starts and stops a Stopwatch object and prints various time measurement statistics.

Using Timers and TimerEvents

Voyager's Timer class acts like an alarm clock. You can set a Timer object to send a TimerEvent to one or more listeners. Upon receiving an event, a listener performs an action. When the action is complete, the timer can continue by sending a TimerEvent to its next listener. To set up a timer and listeners, follow these steps:

1. Construct a timer and one or more listeners.
2. Set the timer to generate one-shot or periodic events.
3. Add the listeners to the timer.

Constructing a Timer

When you construct a timer, it is placed in a TimerGroup. Each TimerGroup has its own thread, and all timers in a TimerGroup share its thread to generate events. Unless specified otherwise, a timer is placed in the default TimerGroup and its thread priority is set to normal (Thread.NORM_PRIORITY).

You can make a group of timers use a separate thread by assigning the timers to a discrete TimerGroup at construction. First, construct a new TimerGroup, optionally supplying a thread priority as a parameter, and then construct timers with the new TimerGroup as a parameter:

```
TimerGroup newgroup = new TimerGroup( Thread.MIN_PRIORITY );
Timer timer1 = new Timer( newgroup );
Timer timer2 = new Timer( newgroup );
```

Setting a Timer

You can set a timer to generate an event at a particular point in time, after a specified period of time, or periodically with the following methods defined in `Timer`:

- ◆ `alarmAt(Date date)`
Sets the timer to generate an event at the specified time.
- ◆ `alarmAfter(long milliseconds)`
Sets the timer to generate an event after the specified number of milliseconds.
- ◆ `alarmEvery(long period)`
Sets the timer to generate an event every time the specified period of time (in milliseconds) elapses.

Other `Timer` methods used to work with timer events include:

- ◆ `clearAlarm()`
Cancels the generation of the timer's event.
- ◆ `getAlarm()`
Returns the time that the timer is scheduled to generate its next event.
- ◆ `getPeriodicity()`
Returns the number of milliseconds between the timer's events.

Adding a Listener to a Timer

A timer generates an event only if it has a listener. Add an object to a timer as a listener using these steps:

1. Ensure that the object's class implements the `TimerListener` interface.
2. Send `addTimerListener()` to the timer with an instance of the object as a parameter.

To remove a listener from a timer, call `removeTimerListener(TimerListener listener)` on the timer.

Multiple listeners to a timer use a single thread, the timer's `TimerGroup` thread, to perform actions upon receiving events. You can override this default behavior by wrapping a listener with a `TimerListenerThread`; that is, you can construct a `TimerListenerThread` object with an instance of the listener as a parameter. `TimerListenerThread` implements `TimerListener`.

For example, suppose a `listener1` object listens to a `timer1` timer. The following code wraps `listener1` with a `TimerListenerThread` and then adds the wrapped listener to `timer1`.

```
TimerListener timerListener1 = new TimerListenerThread( listener1 );
timer1.addTimerListener( timerListener1 );
```

A listener wrapped with a `TimerListenerThread` is dynamically allocated a new thread from a thread pool when it receives an event. In this way, the timer can use its `TimerGroup` thread to continue delivering events to other listeners without waiting for the wrapped listener to perform its action.

By default, the priority of a new thread allocated by `TimerListenerThread` is equal to the priority of the current thread. To override the default, specify the desired priority when you construct the `TimerListenerThread` object:

```
new TimerListenerThread( listener1, Thread.MAX_PRIORITY )
```

Example

The [Timer1 Example](#) demonstrates a ramification of Voyager's default thread behavior, that is, sharing a `TimerGroup` thread. Two listeners receive `TimerEvent` events via the same thread, so the second listener does not receive a `TimerEvent` until the first listener completes its `timerExpired()` method.

The [Timer2 Example](#) demonstrates creating a new `TimerGroup`. A `timer1` listener receives an event from the default `TimerGroup`'s thread, and a `timer2` listener receives an event from the new `TimerGroup`'s thread.

The [Timer3 Example](#) demonstrates allocating listeners separate threads to perform actions upon receiving `TimerEvent` events. The second listener receives a `TimerEvent` before the first listener's `timerExpired()` method completes.

Replication 15

Data stored in directory servers is usually considered to be critical. Therefore, it is necessary to ensure that the data stored in such directories is readily accessible, and that it is not lost due to faults in either the software or hardware maintaining that data. Persistent data stores, such as Voyager's Persistent Directory, provide a solution that ensures that the data placed in the directory server is preserved even in cases where the directory server or the machine it runs on has a fault. Voyager's Persistent Directory reconstructs the data that was placed into the Directory Server upon restarting the server.

Although persistent data stores solve the problem of lost data, they do not solve the problem of availability. Should the directory server become unavailable, clients and servers relying on it may be unable to function correctly, or at all.

Replication solves the problem of availability by copying data stored in a directory to directories on other systems. These directories, or peers, combine to form a cluster of replicated directory servers. As long as at least one peer in the cluster remains available, clients and servers can continue to function normally.

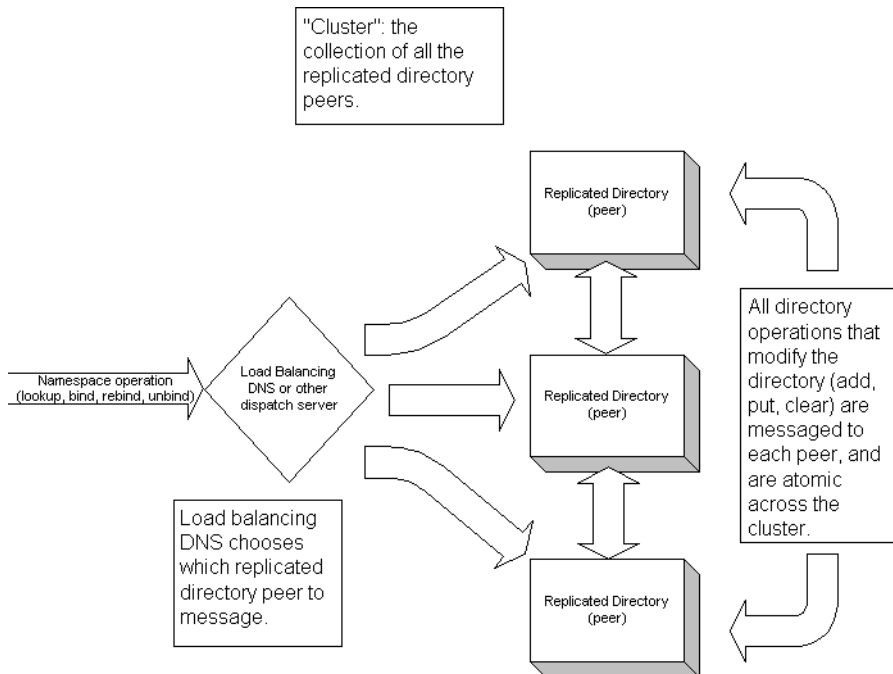
In this chapter, you learn to:

- ◆ understand replication
- ◆ use replication
- ◆ add directories

- ◆ use a load balancing DNS server

Understanding Replication

Voyager's Replicated Directory Service solves the availability problem described above. A Replicated Directory Service consists of a cluster of replicated directories. Each directory server (peer) in the Replicated Directory Service cluster is identical to the other peers in the cluster. Messages that modify the data in one peer are automatically forwarded to the other peers in the cluster, ensuring consistency.



For directory replication to be useful in any system, the following requirements must be met by the replicated directories:

- ◆ All operations that modify a directory must be atomic on the cluster.
- ◆ All cluster peers must be identical.
- ◆ While using a load balanced DNS server, performing directory actions on the cluster must appear identical to performing the same actions on a single directory.
- ◆ Administrators must be able to add and remove cluster peers when necessary.
- ◆ New cluster peers must be synchronized with the cluster irrespective to the amount of data already inserted into the cluster.
- ◆ The cluster must be able to re-synchronize peers that become temporarily unavailable due to system faults.

Voyager's Replicated Directory Services satisfy these requirements. Furthermore, no additional APIs are required. Whether using a stand-alone Voyager Directory Server or a replicated Directory Server cluster, all access is through the same Namespace and IDirectory interfaces.

Using Replication

Voyager maintains a list of the location of the peers in a directory server. The exact location of this list does not matter; all that is needed is a URL to a directory where the list is to be stored (`///dallas:8000/clusterlist`, for example).

Note: You *may* specify the location of that list in one of the replicated directory peers of the cluster, but that information *will not* be replicated to the rest of the cluster.

Note: No maintenance on your part is required of this list; all you need to know is the URL to its location. Since this list maintains an important set of data, it is a good idea to ensure that the directory that maintains that data is persistent.

Voyager's Replicated Directory Services can be started in two ways: from the command line or programmatically in application code. For the purposes of the following example, the cluster list will be placed at `dallas:8000/clusterlist`. Also assume that the cluster will be started with two nodes, one on `austin:8000` and one on `dallas:7000`.

To start a replicated directory server from the command line, issue the following command:

```
> voyager <port> -replication <URL>
```

The <URL> is an optional argument to specify the location of a directory that contains the list of replicated directory peers. If no URL is specified, the local directory server will contain the list. The following list shows valid formats for the URL:

- ◆ //host:port/ - store the list in the root of the directory server on the specified host and port
- ◆ //host:port/path - store the list in the directory server on host at port in the specified path
- ◆ //host:port- same as //host:port/
- ◆ host:port/path - same as //host:port/path
- ◆ host:port - same as //host:port/
- ◆ port - same as //localhost:port/
- ◆ port/path - same as //localhost:port/path
- ◆ [none] - use the local directory server, store in root path

Note that if any URL argument is specified, at least a port must be specified.

In our example, we would issue:

```
austin> voyager 8000 -replication dallas:8000/clusterlist  
dallas> voyager 7000 -replication 8000/clusterlist
```

To initiate directory replication from within your code, use the following code (assuming Voyager has been started with `Voyager.startup()`):

```
import com.objectspace.voyager.replication.Replication;  
Replication.install( "dallas.objectspace.com:8000/clusterlist" );
```

Again, any of the valid URL formats are permissible. Pass null to store the peer list in the local directory server.

You can turn off replication programmatically as well:

```
import com.objectspace.voyager.replication.Replication;  
Replication.uninstall();
```

Note: If you are using both load balancing and replication, you must uninstall them in the reverse order they were installed.

Note: If you uninstall replication, changes made to the directory server are no longer replicated. If replication is later installed again, any changes will be propagated to the other peers to synchronize the data.

Once you have started your Voyager Replicated Directory Service cluster in either of these ways, you can begin to store and retrieve objects in the cluster using either Namespace or the IDirectory interface, as with standard Voyager Directory Services. Note that all operations that modify a directory in the cluster will be more expensive (take longer), but lookup operations will be unaffected. The extra overhead incurred for modifying operations is due to locking the cluster, replicating the message, and unlocking the cluster, which is the cost of ensuring the operations are atomic across the cluster.

Example

The and [Replication Example 2](#) demonstrate how to create and use a replicated cluster.

Adding Directories

If replication is enabled for a Voyager Directory Server, the entire directory structure will be replicated. However, if you create your own directory (an object that implements `com.objectspace.voyager.directory.IDirectory`), and bind that directory into a peer, as in the following example:

```
Directory dir = new Directory();
Namespace.bind( "/mydir", dir );
dir.put( "foo", new Object() );
```

a proxy to that directory is bound in the peer. The directory itself is not replicated, only the proxy to that directory. The replicated directory server will continue to federate into that directory, but if that directory becomes unavailable, the branch of the directory tree under that directory also becomes unavailable.

To explicitly add a subdirectory to a cluster, use the `IDirectory.createSubdirectory()` method.

Using a Load Balancing DNS Server

A load balancing DNS server associates the same hostname to multiple machines with different IP addresses. If you are using a load balancing DNS server, the behavior of proxy addressing should be changed to use IP addresses rather than hostnames by setting the system property `USE_IP_ADDRESSING=true` for each peer. This can be done in one of three ways:

1. From a Properties file, using the `-p` option on the command line
2. From the management console
3. From the command line

```
Voyager <port> <args> -x -DUSE_IP_ADDRESSING=true
```

This example assumes you are using Sun's VM.

Note: If your server's IP address is set through DHCP, and your application uses Activation or Entity Beans, do not enable IP addressing.

Load Balancing 16

Load, loosely defined, is utilization level of a particular physical machine or application. Machine load refers to the load that the running application software places on the machine itself. Application load generally refers to the number of concurrent threads, client connections, or executions in general that an application as a whole is processing. Higher machine load produces longer response times for operations, thus making applications run slower. The application load of the applications executing on that machine, the speed of the processor, the amount of memory available, and other factors all contribute to the load of a machine.

While developing distributed software systems, it is difficult to determine the amount of application load the system may create. Even after system creation, additional factors may require that the application be able to handle a higher load, such as supporting more client connections than originally expected. But regardless of the load the application must support, sacrificing machine performance is undesirable. Therefore, it is important to have a means to reduce individual machine load without decreasing the supported load of the application as a whole.

One solution to reduce individual machine load is to use load sharing, or load balancing. In such an architecture, identical servers (or server objects) are placed on distinct physical machines. These servers register themselves with a dispatcher, such as a directory server. When a client needs to interact with a server, it requests the service that it needs from the dispatcher. The dispatcher contains a load balancing policy that determines which of the registered servers to use to execute the service. A load balancing policy could be as

simple as a round-robin policy or as sophisticated as a policy that can poll the registered servers to find the server with the lowest actual machine load.

Voyager ORB Professional includes a load balancing mechanism. This chapter describes how Voyager allows load balancing to be easily incorporated into your distributed software systems.

This chapter assumes a basic understanding of Voyager, including Namespace, Voyager Directory Services, and Proxy objects.

In this chapter, you learn to:

- ◆ balance application load with Voyager
- ◆ create a load balanced directory server
- ◆ work with load balanced directories

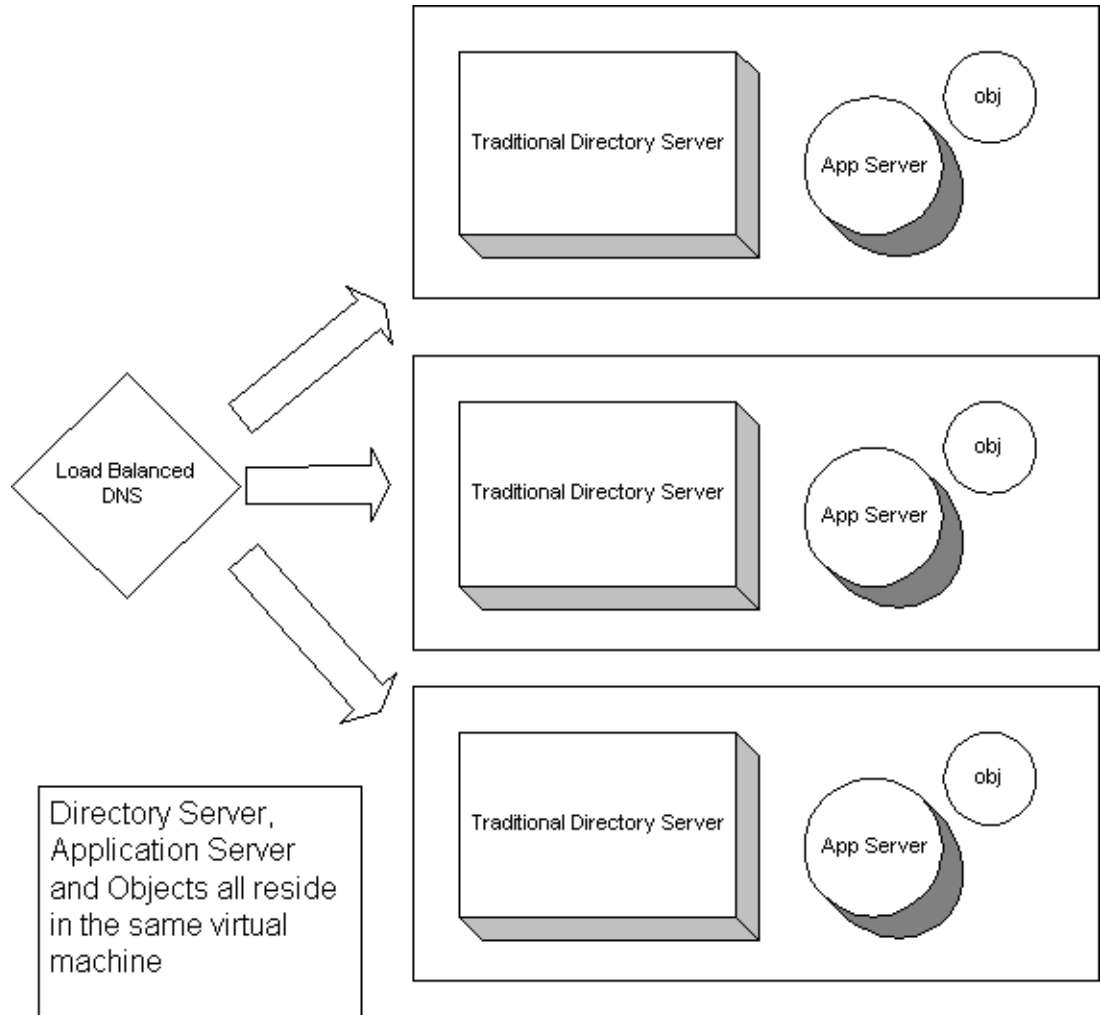
Balancing Application Load with Voyager

Voyager supports two different methods of load balancing services provided by application servers:

- ◆ co-located directory server/application server
- ◆ separate directory server/application servers.

Co-located directory server/application server

In this setup, both the application server and the directory server reside in the same virtual machine. A load balancing Domain Name Service (DNS) handles all load balancing and fault tolerance. The application server uses its internal directory server for storage. Each application server is configured using identical configuration information, so all the data stored in each internal directory server will also be identical.



Advantages

This setup offers the following advantages:

- ◆ It requires fewer virtual machines than the second load balancing method, so less memory is required.

- ◆ Since directory fault tolerance is handled by the DNS (if the directory server goes down, so would the application server), no directory replication is required, so network bandwidth is conserved.
- ◆ Another advantage, though minimal, is easier administration, since only one application (application server/directory server combination) is started on each machine.

Disadvantages

There are also a few disadvantages to this setup.

- ◆ Since the directory server and application server reside in the same virtual machine (and therefore, memory space), the directory server is penalized for the application server's client connection traffic and usage.
- ◆ This architecture is not as flexible as one with a separation of application servers and directory servers.
- ◆ This setup requires proxies to server objects to use IP addresses rather than hostnames. This causes problems if your application uses the Activation framework or Entity Beans and the server's IP address is set through DHCP.

Installation

First, ensure that the system property `USE_IP_ADDRESSING` is set to true for the server. This can be done one of three ways:

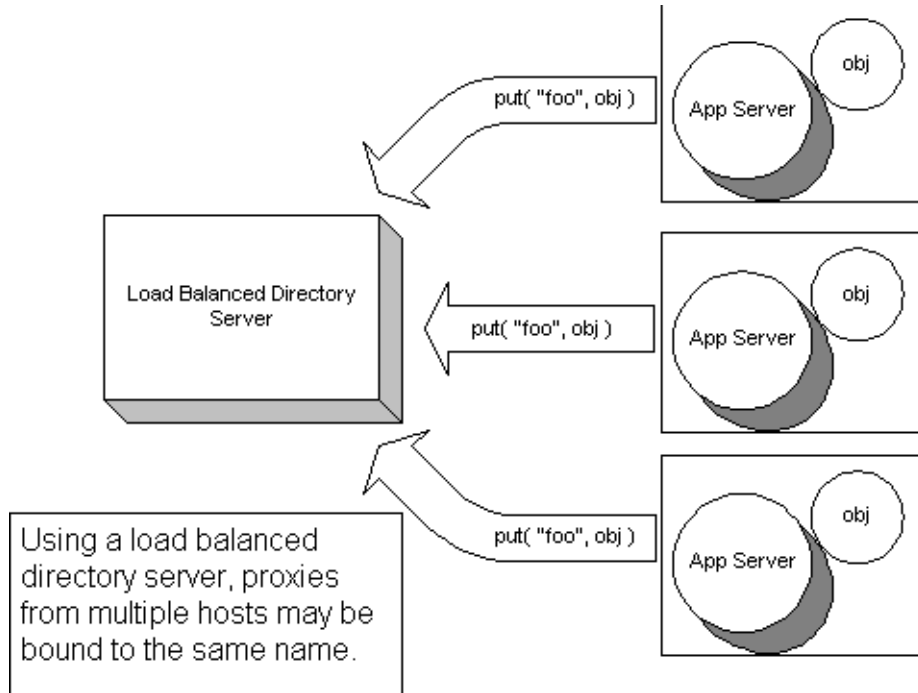
1. From a Properties file, using the `-p` option on the command line.
2. From the management console.
3. From the command line

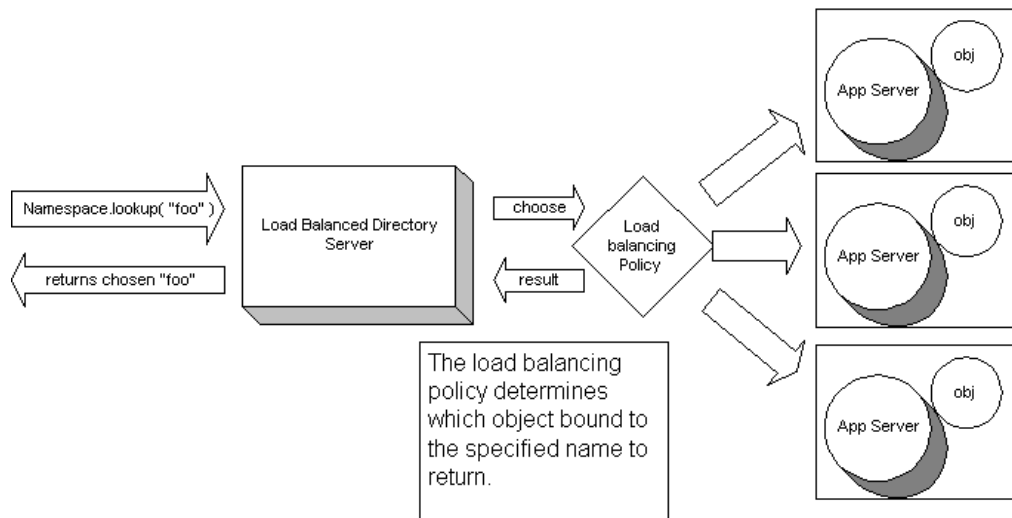
Voyager <port> <args> -x -DUSE_IP_ADDRESSING=true
(this example assumes you are using Sun's VM).

Next, when administering the application server through the console, set the bean home URL to be the same as the URL of the application server. Select the EJB node of the server's tree in the management console, and enter the local URL into the Bean Home field. For example, if your application server is running on `dallas.objectspace.com`, port 8000, set the bean home URL to `dallas.objectspace.com:8000`.

Separate directory server/application servers

In this system architecture, Voyager uses the Voyager Directory Server as its dispatch mechanism for load balancing. All server objects (such as bean home objects in EJB) are bound by name to a load-balanced directory server. This directory server could be a single directory server, or it could be a cluster of replicated load-balanced directory servers (see the "[Replication](#)" chapter for more information). The load-balanced directory server allows multiple objects to be bound to the same name (with a few exclusions, which are explained below). Client applications then request server objects by name from the directory server as usual. When the load-balanced directory receives the request, it uses its load balancing policy to determine which of the bound objects to return.





Advantages

There are a number of advantages to using this load balancing architecture.

- ◆ Rather than relying on a load balancing DNS, you can create your own custom load balancing policy to determine how load is distributed over your server objects.
- ◆ With this approach you can also easily tailor your system setup according to where load is occurring, adding more directory servers or application servers when necessary.
- ◆ Your directory servers are not penalized for application server traffic or processing and vice versa.

Disadvantages

There are also drawbacks to using this architecture:

- ◆ Increased complexity, as you now have more servers running.
- ◆ Minimally increased internal network traffic, since the directory servers reside outside of the application server VMs.

Creating a Load Balanced Directory Server

Voyager's load balancing mechanism can be enabled either from the command line or within your application code. In either case, you will need to determine beforehand what portion(s) of your directory tree will be load balanced. For the purposes of the examples, a Voyager server will be started on port 8000, and will use the path /loadbalance as the load balanced root.

To start a load balanced directory server from the command line, use the following command:

```
> voyager <URL> -lb [<path>]
```

<URL> is the startup URL of the voyager server, and <path> is the base path that will be load balanced. If the path is not specified, the entire directory server will be load balanced. In the example, we would execute the following:

```
> voyager 8000 -lb /loadbalance
```

To create a load balanced directory programmatically, write the following in the application code:

```
import com.objectspace.voyager.*;
import com.objectspace.voyager.loadbalance.LoadBalancedDirectory;
Voyager.startup( "8000" );
LoadBalancedDirectory dir = new LoadBalancedDirectory();
Namespace.bind( "/loadbalance", dir );
```

Working with Load-Balanced Directories

As stated previously, Voyager allows you to determine which portions of your directory tree are load balanced and which are not, since it is not always desirable to have your entire directory tree load balanced.

Load balanced directories behave differently than traditional directory services. In traditional directory services, only one object may be associated with a particular name. For example, if we execute:

```
Namespace.bind( "foo", new Object() );  
Namespace.bind( "foo", new String() );
```

A `NamespaceException` is thrown upon the second bind, since the first object is already bound to that name.

Using load balanced directories, however, this restriction is lifted with a few exclusions, depending on the type of object being bound to the directory:

- ◆ Only one non-proxy object may be bound to a particular name.
- ◆ Only one directory object (one that implements `com.objectspace.Voyager.directory.IDirectory`) may be bound to a particular name, even if it is a Proxy to that directory.
- ◆ Two Proxy objects with the same ORB address, coming from the same ORB server, may not be bound to the same name. For example, the following code in the same application is legal:

```
Namespace.bind( "/loadbalance/foo", Proxy.export( "one", "8000" ) );  
Namespace.bind( "/loadbalance/foo", Proxy.export( "two", "9000" ) );
```

This works because the proxies are exported on different ports. The following example code is illegal because the two Proxy objects will have the same address.

```
Namespace.bind( "/loadbalance/foo", Proxy.export( "one", "8000" ) );  
Namespace.bind( "/loadbalance/foo", Proxy.export( "two", 8000 ) );
```

Example

and [Load Balancing Example 2](#) demonstrate putting values into a load balanced directory.

Removing items

Just as the rules are a little different for adding values to a load-balanced directory, so are the rules for removing. When a remove message is received by a load-balanced directory for a particular key, the directory checks the type of value associated with that key. If it is not a list of proxies, the key/value pair is removed from the directory. If that key corresponds to a list of proxies, the directory searches for a proxy coming from the

host that is sending the remove message. If a proxy from that host exists, that proxy is the only proxy removed from the directory. If no proxy from that host exists, however, all the proxies associated with that key are removed from the directory.

Adding Directories

Once a load-balanced directory root is set, all subdirectories under that root will be load balanced as well. If you execute the following code:

```
Directory dir = new Directory();  
Namespace.bind( "/loadbalance/newdir", dir );
```

all subsequent binds and lookups through Namespace to /loadbalance/newdir are load balanced.

However, in the above example, if you continue to operate on dir, the put and get operations are **not** load balanced. To have those operations load balanced, execute the following:

```
IDirectory dir = new Directory();  
Namespace.bind( "/loadbalance/newdir", dir );  
dir = (IDirectory) Namespace.lookup( "/loadbalance/newdir" );
```

All subsequent messages to dir are load balanced.

Example

[Load Balancing Example 2](#) demonstrates how to correctly put directories into load-balanced directories.

Load Balancing Policies

As stated earlier, the Voyager load balanced directories rely on a load balancing policy to determine which Proxy object to return. By default, this is a round-robin policy. You may, however, create your own load balancing policy.

In order to create your own load balancing policy, your implementation class must implement `com.objectspace.voyager.loadbalance.ILoadBalancePolicy`. This interface has three methods:

```
public interface  
    ILoadBalancePolicy  
{  
    Object chooseObject( HostList list );
```

```
HostList createList();
IManegeable getManager();
}
```

A `HostList` is created for each load-balanced entry in the directory. It contains a mapping of the `URLProxy` objects contained in the list. The `createList()` method offers the ability to create custom `HostList` classes if special state or functionality is required. For example, the round-robin policy needs to keep track of which object it returned last, so a custom `HostList` class was created for that purpose (`HostQueue`). But if no special state or functionality is required, using the standard `HostList` class will suffice.

`chooseObject()` is the method called to choose the appropriate proxy object based upon the load balancing policy. It will be called if more than one proxy exists for the specified lookup.

`ILoadBalancePolicy` extends the `IPolicy` interface of the management framework. Therefore, all load balancing policies must implement `getManager()`. `IManegeable` is also a part of the management framework. For more information on either interface or the management framework, refer to "[Load Balancing](#)". It is valid for `getManager()` to return null (no manager).

Two methods exist to incorporate your custom policy into the load-balanced directory:

1. Through the command line, via:

```
voyager 8000 -lb <path> -lb_policy <classname>
```

2. Or programmatically through the following method:

```
com.objectspace.voyager.loadbalance.LoadBalancePolicyManager.setPolicy( ILoadBalancePolicy );
```

Example

[Load Balancing Example 4](#) demonstrates an example load balancing policy.

Affinity

By default, Voyager load-balanced directories possess a quality called “affinity”. That is, if a server binds an object to a particular name in a load-balanced directory, and then later requests an object with that name, the load-balanced directory will short-circuit the load-balancing policy to return that server’s local object. For example, in EJB, if multiple beans are being served from one particular host, these beans may all register

themselves in a remote load-balanced directory server. If these beans need to interact with each other, they would usually want to interact with local beans if possible. Affinity ensures this happens.

Affinity may be enabled and disabled at any time programmatically by invoking `LoadBalancePolicyManager.setAffinityEnabled()`. You may want to disable affinity if the service being looked up is computationally intensive and you want to ensure that the load is distributed via the load balancing policy. To disable affinity, set the system property `objectspace.loadbalance.affinity_disabled`. With affinity disabled, no checking is done to see if a local object exists for the requestor of the object.

Example

[Load Balancing Example 5](#) and [Load Balancing Example 6](#) demonstrate how affinity affects which objects are returned.

Configuration and Management

17

Several of Voyager's internal settings can be modified at runtime using static methods. For example, you can change the maximum thread pool size at runtime by using `ThreadManager.setPoolSize()`.

The same settings can also be set using standard Java property files, which are read at startup. This approach allows values to be set and changed without modification of the application source code.

You can extend Voyager's configuration and management capabilities to fit the specific needs of your system. With the management framework, objects, services, and entire subsystems can be outfitted for remote configuration and management. Then with the workshop framework, objects can be viewed and manipulated using ObjectSpace's Workshop framework.

In this chapter, you will learn to:

- ◆ understand Voyager properties
- ◆ specify a properties file
- ◆ specify multiple values
- ◆ use the Voyager management framework
- ◆ work with the ObjectSpace workshop framework

- ◆ use the Voyager Management Console

Note: The configuration and configuration frameworks features are only available with Voyager ORB Professional.

Understanding Voyager Properties

The following table summarizes Voyager's user-customizable properties. Each property is case sensitive.

Property	Value
voyager.ClassManager.enableResourceServer	true false
voyager.ThreadManager.setMaxIdleThreads	<int>
voyager.activation.Activation.register	#<classname>
voyager.loader.VoyagerClassLoader.addResourceLoader	#<classname>
voyager.loader.VoyagerClassLoader.addURLResource	<URL>
voyager.loader.VoyagerClassLoader.setResourceLoadingEnabled	true false
voyager.router.Routing.setRouterAddress	<XURL>
voyager.tcp.TcpTransport.setServerListenBacklog	<int>
voyager.transport.Transport.register	#<classname>
voyager.transport.Transport.setDefaultTransport	<transport id>
lib.util.Console.setLogLevel	exceptions silent verbose

voyager.ClassManager.enableResourceServer

This property allows a Voyager server to be able to serve resources, typically classes, via HTTP. It is equivalent to the `ClassManager.enableResourceServer()` method. The default value is false. For example:

```
voyager.ClassManager.enableResourceServer=true
```

voyager.ThreadManager.setMaxIdleThreads

This property allows users to specify the maximum number of threads Voyager will cache. Higher numbers are useful for servers with more memory, while lower numbers are often useful when memory resources are limited or must be shared with other applications. This property is equivalent to the `ThreadManager.setMaxIdleThreads()` method. The default value is `Integer.MAX_VALUE`. For example:

```
voyager.ThreadManager.setMaxIdleThreads=50
```

voyager.activation.Activation.register

This property allows activators to be plugged into the activation framework. It is equivalent to the `Activation.register()` method. The classname provided must resolve to a public class that implements `IAActivator` and has a public no-argument constructor. For example:

```
voyager.activation.Activation.register=#com.acme.AcmeActivator
```

voyager.loader.VoyagerClassLoader.addResourceLoader

This property allows resource loaders to be added to Voyager's classloading mechanism. A custom resource loader allows Voyager to retrieve resources, including Java classes, from custom sources. The property is equivalent to the `VoyagerClassLoader.addResourceLoader()` method. The classname provided must resolve to a public class that implements `IResourceLoader` and has a public no-argument constructor. For example:

```
voyager.loader.VoyagerClassLoader.addResourceLoader=#com.acme.AcmeLoader
```

voyager.loader.VoyagerClassLoader.addURLResource

This property allows the Voyager server to load classes from the specified URL. If the URL is an HTTP URL, it must map to an HTTP server, such as a remote Voyager that has resource serving enabled. This property is equivalent to the `VoyagerClassLoader.addURLResource()` method. For example:

```
voyager.loader.VoyagerClassLoader.addURLResource=http://acme.com:8000
```

voyager.loader.VoyagerClassLoader.setResourceLoadingEnabled

This property enables or disables resource loading. It is primarily used to disable resource loading, which includes dynamic proxy generation, loading from a remote URL, etc. It is equivalent to the `VoyagerClassLoader.setResourceLoadingEnabled()` method

and has a default value of true. All proxies, including those for select Voyager classes, must be generated manually with the pgen utility. For example:

```
voyager.loader.VoyagerClassLoader.setResourceLoadingEnabled=false
```

voyager.router.Routing.setRouterAddress

This property allows a routing address to be specified. If an application specifies a router, all remote messages are automatically sent to the router for redirection to their final destination. It is equivalent to the `Routing.setRouterAddress()` method. For example:

```
voyager.router.Routing.setRouterAddress=//10.2.15.194:8000
```

voyager.tcp.TcpTransport.setServerListenBacklog

This property can be used to set the listen backlog for all TCP server sockets used by Voyager. It is equivalent to the `TcpTransport.setServerListenBacklog()` method. The default value is 50. For example:

```
voyager.tcp.TcpTransport.setServerListenBacklog=20
```

voyager.transport.Transport.register

This property allows a custom transport mechanism to be registered with Voyager's pluggable transport system. It is equivalent to the `Transport.register()` method. By default, a TCP transport mechanism is registered with Voyager. The classname provided must resolve to a public class that implements `ITransport` and has a public no-argument constructor. For example:

```
voyager.transport.Transport.register=#com.acme.SSLTransport
```

voyager.transport.Transport.setDefaultTransport

This property allows users to set the default transport used by Voyager. It is equivalent to the `Transport.setDefaultTransport()` mechanism. By default, all remote messaging is handled by the TCP transport. If a custom transport is set to the default, an instance of the transport must be registered with Transport. For example:

```
voyager.transport.Transport.setDefaultTransport=ssl
```

lib.util.Console.setLogLevel

This property allows the Console log level to be set. It is equivalent to the `Console.setLogLevel()` method. Available options are silent, exceptions, and verbose. For example:

```
lib.util.Console.setLogLevel=exceptions
```

Specifying a Properties File

Voyager servers are started with the voyager utility. The voyager utility included with Voyager ORB Professional has an extra flag, the -p flag, for specifying a properties file. For example, the following code will start a Voyager server on port 8000 with the properties file acme.properties. Once all flags have been processed, Voyager will read in the properties file and configure itself as specified in the properties file. Voyager will then startup.

```
voyager 8000 -p acme.properties
```

Example

The [Configuration1 Example](#) demonstrates custom configuration of a Voyager server.

For custom applications in which the voyager utility is not used to start up a server, a Voyager properties file can be loaded using a PropertyLoader.

Example

The [Configuration2 Example](#) demonstrates custom configuration of a user application

Specifying Multiple Values

Some Voyager properties can be specified more than once. For instance, an application may need to be able to load classes from several URLs. Therefore, Voyager allows the addURLResource property to be specified more than once using an indexing syntax. For example, the following code will install URL resource loaders for two different remote HTTP servers.

```
voyager.loader.VoyagerClassLoader.addURLResource[1]=http://acme.com:8000  
voyager.loader.VoyagerClassLoader.addURLResource[2]=http://acme2.com:8000
```

Example

The [Configuration3 Example](#) demonstrates specifying multiple Voyager properties.

Using the Voyager Management Framework



Voyager's Management Framework provides interfaces that allow services to be configured and administered remotely. Any service, subsystem, or object can potentially be fitted with the necessary interfaces to allow it to be managed from the Voyager Management Console or another management tool.

IConfiguration

Implement the IConfiguration interface to allow an object to configure some aspect of a Voyager server or service when that server starts up. For instance, if your application has an e-mail service, you may want to configure that service with the address of an SMTP server elsewhere on the network. The install() method will be called by Voyager when that server starts up. install() should set any necessary startup information on the service. This method is also responsible for creating a Management Agent for its service or object, if necessary. The URL passed to the install() method indicates the location in the Voyager Directory Server where this IConfiguration object was found. It can, therefore, be used to bind a Management Agent back to the same place using an Installer. See the "Installer" section on page [Installer](#).

Example

The [Configuration and Management](#) example demonstrates the use of Configuration objects.

IManagementAgent

Implement the IManagementAgent interface to allow objects within your server to be manipulated at runtime. A Management Agent is an object residing on your server that an outside entity can communicate with to request information about or act upon your system's objects.

The getManagedObjects() method returns the objects that the Management Agent is responsible for or authorized to act upon. getActions() returns a list of possible actions that

can be taken by this agent. `performAction()` executes one of these actions. `getEventTypes()` returns an array of Strings that describe arbitrary events. `addEventListener()` adds a listener. See the "[Listeners and Events](#)" section for more information.

Example

The [Configuration and Management](#) example demonstrates use of management agents.

Listeners and Events

Management events are, for the most part, left for you to define. Management Agents can generate events, and instances of `EventListener` can be added as listeners to those events. Nothing distinguishes an event except the String that identifies it. The `CompositeListener` class helps organize listeners.

Example

The [Configuration and Management](#) example demonstrates use of events and listeners.

Installer

The `Installer` is a utility class that allows an instance of `IConfiguration` or `IManagementAgent` to be bound into the Voyager Directory Server at a later time than when it was created. For instance, in the case of a service, the configuration object might handle the instantiation of a Management Agent. Then it creates an `Installer` to pass to the service, so that when the service finally comes online, it runs the installer that binds its Management Agent back into the Voyager Directory Server.

Example

The [Configuration and Management](#) example demonstrates use of the `Installer` class.

Working with the ObjectSpace

Workshop Framework

The Workshop framework offers a way of representing hierarchical data to the user. Hierarchical data structures, such as directories and file systems, can be represented in any number of ways. The Workshop framework provides abstractions for data and views that are independent of representation.

Nodes, Tools, and the ToolBox

Each data element is considered to be a node. Nodes may have child nodes that represent sub-data structures. Interface `INode` should be implemented by any data element, or wrapper, that will be displayed by your system. Tools are classes that implement the `ITool` interface and know how to represent particular nodes. The `ToolBox` implements `IToolBox` and is responsible for associating tools and other resources with given nodes.

Example

The example [Configuration and Management](#) provides a sample implementation of the `ITool` interface.

Workshop, WorkSession, and WorkContext

The Workshop is an abstraction of your representation to the user. `WorkSession` allows the Workshop to question its user, such as whether the session is dead or whether the application can be closed. `WorkContext` allows the user or the tools to make requests of the Workshop, such as refreshing the screen or showing a status message.

Using the Voyager Management

Console



The Voyager Management Console (VMC) is an implementation of the Workshop framework. It is capable of traversing a JNDI directory, discovering its contents, and displaying and manipulating data elements. Any object in the directory that has a Tool class defined for it may be viewed using the VMC.

Directory Structure

A specific directory structure is used with the VMC. Certain conventions are followed so that, as much as possible, even the VMC can be configured from the directory server. These conventions follow:

- ◆ Objects bound with names beginning in “__” (double underscore) are not visible.
- ◆ The name __AGENT is reserved for instances or proxies of type IManagementAgent.
- ◆ In the root context, the directory called __Resources contains all of the configuration information for the VMC and the server profile template for configuring Voyager servers.
- ◆ Mappings between object class names and class names of the tools that are capable of representing them are in the directory __Resources/Tools in the following format: objectClassName/1.0 is bound to a String, toolClassName. If there is more than one tool class for a given object, use 2.0, 3.0, etc. For instance, two tools for the class MyObject would be bound to __Resources/Tools/MyObject/1.0 ◇ MyObjectToolA and __Resources/Tools/MyObject/2.0 ◇ MyObjectToolB.
- ◆ Icons are stored by arbitrary name in the __Resources/Icons directory as byte arrays.
- ◆ The server profile template is at __Resources/ConfigTemplate. It contains a binding for the server configuration, in this case __Resources/ConfigTemplate/1.0 ◇ com.objectspace.voyager.system.VoyagerServerConfiguration. ConfigTemplate also has a subdirectory for each installed service, which in turn has its configuration class names bound the same way.

Server profiles may be grouped arbitrarily within the directory.

Service Configuration

Services are added to the server profile template in the Voyager Directory Server using serialized instances of `IConfiguration`. These serialized objects are found either in the `.jar` file that the service was distributed in or in some other file in the distribution. When a user installs a service via the VMC, the configuration object is retrieved from the `.jar` file or other file, and its `install()` method is invoked. This, in turn, populates the server profile template with configuration class names and also enters information specific to that particular service, such as Tool class mappings and icons, in the “__Resources” directory.

Console Behavior

VMC’s ToolBox implementation refers to the directory’s “__Resources” directory for tool mappings and icons. The console first checks whether the “__Resources” directory exists, prompting the user to create it if it does not exist. Then, the console locates the visible entries in the directory. For every object that is bound into the directory, the console attempts to create a Tool for it, referring to the Tool mappings. If no mapping exists, a message will be displayed. “__AGENT” is a special case. This binding is reserved for a proxy to a management agent. If this proxy points to an object on a server that no longer exists, the binding is removed. Otherwise, the management agent is queried for its managed objects, and Tools are created for them.

Policy Management 18

Voyager ORB Professional provides a framework for easy integration of policies and policy configuration. A policy is characterized by certain qualities. A policy is:

- ◆ unique to a particular Voyager ORB
- ◆ global in scope within a particular Voyager ORB
- ◆ configurable via code and/or a GUI
- ◆ persistent, allowing the ORB to be stopped and restarted with the same policies in force.

Voyager provides support for these qualities as well as a PolicyEvent mechanism based on the Java 1.1 event-listener mechanism.

Unless otherwise noted, all mentioned Voyager classes and interfaces are located in the `com.objectspace.voyager.management` package.

In this chapter, you learn to:

- ◆ understand IPolicy
- ◆ understand policy events

Understanding IPolicy

Voyager policies implement the IPolicy interface. IPolicy extends the java.io.Serializable interface; therefore, any Voyager policy must properly handle serialization semantics.

Note: A class implementing IPolicy is free to implement the java.io.Externalizable interface since the Externalizable interface extends the Serializable interface.

It is recommended that the serialVersionUID be explicitly set for any class implementing IPolicy to aid in version management. For more information concerning Java serialization semantics, consult the Object Serialization Specification.

The IPolicy only requires the getConfigurator() method to be implemented. This method returns an object implementing the IPolicyConfigurator interface; however, it is expected that this object will also extend the java.awt.Component class, providing a convenient GUI for configuring an instance of the policy class. A null return value means that no configurator has been defined for this policy type.

There are two reasons that getConfigurator() does not simply return a java.awt.Component. First, the IPolicyConfigurator interface requires methods for managing manageable systems generically. Second, it cannot be guaranteed that the java.awt.Component class will be present. For example, Voyager may be running in a scaled-back Java VM where GUIs are not supported.

This latter reason suggests certain semantics for the implementation of the getConfigurator() method. The returned object should be created using reflection without any explicit dependencies on the existence of Java GUI classes. Any exception can then be caught and handled by returning a null value to indicate that a GUI is not present.

For example, we may have a SamplePolicyConfigurator class that extends javax.swing.JPanel for configuring SamplePolicy objects. SamplePolicy implements IPolicy; therefore, it must implement the getConfigurator() method. That method would look like the following code:

```
public IPolicyConfigurator getConfigurator()
{
    IPolicyConfigurator configurator = null;

    try
    {
```



```

Class configuratorClass = Class.forName("SamplePolicyConfigurator");
configurator = (IPolicyConfigurator)configuratorClass.newInstance();
}
catch (Exception e)
{
// Log exception to the Voyager Console
com.objectspace.lib.util.Console.logStackTrace(e);
}

return configurator;
}

```

This allows the SamplePolicy class to provide a configurator while remaining ignorant of any GUI classes.

For more information on reflection, consult the Java Core Reflection Specification.

Understanding Policy Events

The Voyager managed policy framework provides two types of policy events.

- ◆ *Policy Changed Events* indicate that the policy associated with a policy manager has been changed.
- ◆ *Policy Applied Events* indicate that the policy associated with a policy manager has been applied to determine a course of action.

Objects desiring knowledge of either type of event will implement the PolicyListener interface and register with one or more policy managers, most likely classes implementing the IPolicyManager interface. Policy listeners will then receive PolicyEvent objects as policy events are generated.

Manageable Systems

A system that uses Voyager's managed policy framework is usually designed such that its behavior is governed by the currently established policy. Such systems are *manageable*. Manageable systems have a class implementing the IManageable interface. This interface defines an accessor and a mutator for the manageable system's

established policy. Manageable systems also register their `IManageable` object under a unique name with the Voyager ORB.

The `Manageables` class provides the methods necessary for `IManageable` objects to be added and removed from the Voyager ORB. In addition, the `Manageables` class provides access to the set of registered systems and their current policies. The `getManageables()` method returns an `IManageables` object through which registered manageable systems can be accessed via their registered names. Furthermore, the `getManageables(String url)` method can be used to access the registered manageable systems in a remote Voyager ORB.

For example, a program could set the `SamplePolicy` for the Voyager ORB at `//machine1:9000` with a small set of code. This example presumes that the desired manageable system is registered under the name `SamplePolicyManager`.

```
public void setRemoteSamplePolicy( SamplePolicy policy )
{
    try
    {
        IManageables remoteManageables = Manageables.getManageables( "//machine1:9000" );
        remoteManageables.setPolicy( "SamplePolicyManager", polic y);
    }
    catch ( Exception e )
    {
        // Log exception to the Voyager Console
        com.objectspace.lib.util.Console.logStackTrace( e );
    }
}
```

The Voyager Management Console includes a tab for Policy Management. For a given Voyager ORB, any registered manageable systems are queried for their current policies and their configurators. Any discovered configurators will be available here. Therefore, integrating a custom configuration into the Voyager Management Console is as simple as registering the manageable system with its own Voyager ORB.

If a Voyager ORB is not started from a directory, then the Voyager Management Console may not be available. Registered manageable systems can still be configured from their GUIs using the configurator application provided by the policy management framework:

```
> java com.objectspace.voyager.management.policy.admin.ConfiguratorApp <url>
```

where `<url>` indicates the Voyager ORB whose registered manageable systems are to be managed.

Policy Persistors

The Manageables class also provides the mechanism for persisting managed policies. `getPolicyPersistor()` returns an object responsible for handling the persistence of managed policies. A system that desires to persist its policy would use the policy persistor to save the policy as well as retrieve the policy the next time the system is restarted.

Programmatically, a manageable system would save its policy under the name `SamplePolicy` with the following statement:

```
Manageables.getPolicyPersistor().setPolicy( "SamplePolicy", policy );
```

The same manageable system would then retrieve its saved policy when it is next restarted with the following statement:

```
IPolicy policy = Manageables.getPolicyPersistor().getPolicy( "SamplePolicy" );
```

The actual storage mechanism used by the ORB depends on two factors. First of all, policies may be persisted to a file on disk if the ORB is started with the `-policy_file` parameter. For example, if the Voyager ORB is started with the following statement:

```
> voyager 8000 -policy_file managed_policy_storage
```

then the Voyager ORB started on port 8000 will be using the file named `managed_policy_storage` to store its managed policies. Two ORBs should not share the same policy storage file; however, one ORB's policy storage file could be copied and used by another Voyager ORB.

If the `-policy_file` parameter is not present, then the Manageables framework will attempt to use the Voyager ORB's directory service to store its persisted policies. Therefore, if the Voyager ORB is started via the following statement:

```
> voyager -d 9000/dir
```

then the Voyager directory located at `9000/dir` will be used to store and retrieve persisted policies.

If the Voyager ORB has no directory service available, persisted policies are stored in memory. When the Voyager ORB is taken down, all persisted policies will be forgotten. During system development, this may be a useful arrangement, but upon deployment, either file-based or directory-based policy persistence should be used.

TCP Connection Management

19

Voyager ORB Professional provides the ability to manage the TCP socket connections underlying Voyager-to-Voyager communications. Voyager's TCP Connection Management system registers with "Policy Management" framework under the name defined by the value of `IManagedTcpPolicyManager.MANAGEABLE_NAME`, currently defined as "Connection Management".

To enable TCP connection management for a stand-alone Voyager ORB, use the `-managed_tcp` flag:

```
> voyager 8000 -managed_tcp
```

To enable TCP connection management in Java source code:

```
public static void main ( String args [] )  
  
{  
    com.objectspace.voyager.tcp.managed.ManagedTcpTransport.enable( true );  
    Voyager.startup();  
    // remainder of Voyager-related code  
}
```

There is a cost associated with managing connections. Every time a new connection is desired, Voyager must examine the current policy and the current set of connections to determine if the new connection is allowed. As the number of connections increases, this cost will grow. Furthermore, the more complicated the policy restrictions are, the longer they will take to analyze. Although in most cases this will not be noticeable, high-volume

Voyager networks may wish to tune connection management parameters or even revert to non-managed behavior.

Connection management supersedes any custom transport implementations. If a custom transport implementation is desired, then connection management cannot be used. At this time, this includes HTTP and Socks tunneling support from the Voyager Security module.

A client connection in Voyager is used when one ORB is communicating a function invocation to a remote object. A server connection is involved whenever an exported Proxy receives a remote invocation request for a local object.

In this chapter, you learn to:

- ◆ understand connection management policies
- ◆ understand case policies
- ◆ establish case policies
- ◆ define policy listeners
- ◆ use the TCP connection management GUI

Understanding Connection Management Policies

Voyager connections are segregated into client connections and server connections. A client connection, associated with a Proxy to a remote object, sends function invocation requests. A server connection, associated with a local object exported to remote Voyager ORBs, receives invocation requests.

TCP connections are managed by an established policy, an instance of the `ManagedTcpPolicy` class. A `ManagedTcpPolicy` contains a collection of `CasePolicy` objects describing the restrictions on connections between different Voyager ORBs according to their IP addresses and ports. If a new connection would violate the set limits, then the requesting thread will block until the new connection is allowed. Note that this could cause deadlock problems if distributed objects recursively call methods upon one another such that they use up all allowed connections.

ManagedTcpPolicy also maintains a retry count. This count is used when a client socket is being created. If the creation fails, the system will retry up to the number of retries allowed. This applies to all client sockets regardless of the location of the server to which they are connecting. There is no delay between retries.

Understanding Case Policies

A CasePolicy consists of six characteristics describing how connections should be limited or disconnected.

Maximum Number of Server Connections

Server connections may be capped at a particular number. Server connections include currently active connections accepting invocation requests as well as pending connections awaiting a client to connect.

Maximum Number of Client Connections

Client connections may also be limited. Client connections deliver invocation requests to remote objects.

Maximum Number of Idle Client Connections

A client connection is idle if it is not sending an invocation request or awaiting a response from an invocation request. Idle client connections are pooled, allowing a small number of connections to handle many proxies, as long as invocations are relatively infrequent.

A server connection is idle if it is awaiting an invocation request.

Maximum Number of Live Connections

Live connections includes all connections, server and client, as well as idle and active.

Client Connection Idle Time

A client connection can be given an idle time limit. If a client connection idles longer than this limit, it will be removed from use and closed.

Server Connection Idle Time

A server connection can also be given an idle time limit. This is the amount of time that a server connection will wait during a TCP socket read before aborting by closing the connection. The attached client connection will likely suffer runtime exceptions due to communications failure.

Establishing Case Policies

Case policies must be added to a `ManagedTcpPolicy` object which is then set as the policy for a given ORB. The easiest way is often to obtain the current policy, modify it, then establish the modified policy as the new ruling policy.

`ManagedTcpTransport` implements `IManageable` from Voyager's management package. It will return a copy of the current policy from the `getPolicy()` method. Policy is established using the `setPolicy()` method.

`CasePolicy` objects are managed by three methods on the `ManagedTcpPolicy` class:

- ◆ `public void setCasePolicy(HostAddressRange range, CasePolicy casePolicy);`
Sets the ruling `CasePolicy` for the given range of addresses. All connections that fall within the given range will be subject to the restrictions of the new `CasePolicy`.
- ◆ `public CasePolicy getCasePolicy(HostAddressRange range);`
Retrieves the `CasePolicy` established for the given range of addresses. If no `CasePolicy` is explicitly established, then the least-restrictive `CasePolicy` is returned, no connection or idle time limits.
- ◆ `public void removeCasePolicy(HostAddressRange range);`
Removes any established `CasePolicy` for the given range of addresses.

Two functions also provide access to the global case policy ruling any and all connections for the current Voyager ORB:


```
public void setGlobalCasePolicy( CasePolicy casePolicy );  
public CasePolicy getGlobalCasePolicy();
```

Defining Policy Listeners

This section gives examples of code used to define the policy listeners.

- ◆ Any policy listeners wishing to listen to policy events from the TCP connection management system register with the current ManagedTcpTransport using the following code. The given listener will receive all future policy-changed events:

```
public void addPolicyListener( PolicyListener listener );
```

- ◆ Policy-applied events are not sent to the listener in the following example. The given listener will be deregistered and receive no more policy events:

```
public void removePolicyListener( PolicyListener listener );
```

- ◆ In the following example, the given listener will receive any policy-applied events for the given range and any wholly-included sub-ranges for which a CasePolicy is registered. If the range is the global range ("*:-"), then the given listener would receive all policy-applied events:

```
public void addCasePolicyListener( HostAddressRange range, PolicyListener listener );
```

- ◆ The following example deregisters the given listener from receiving policy-applied events on the given range.

```
public void removeCasePolicyListener( HostAddressRange range, PolicyListener listener );
```

Examples

Setting the Global CasePolicy

To set a Voyager ORB to limit the number of client connections to 25 and the idle time limit to 10 seconds:

```

IManageables manageables = Manageables.getManageables();
ManagedTcpPolicy policy = ( ManagedTcpPolicy )manageables.getPolicy(
IManagedTcpPolicyManager.MANAGEABLE_NAME );
policy.setGlobalCasePolicy( new CasePolicy( 0, 0, 0, 0, 25, 10000 ) );
manageables.setPolicy( IManagedTcpPolicyManager.MANAGEABLE_NAME, policy );

```

Setting Case Policies

To limit the number of client connections to the objectspace.com space to 10 with 5-second idle limits:

```

IManageables manageables = Manageables.getManageables();
ManagedTcpPolicy policy = ( ManagedTcpPolicy )manageables.getPolicy(
IManagedTcpPolicyManager.MANAGEABLE_NAME );
policy.setCasePolicy( new HostAddressRange( "*.objectspace.com" ), new CasePolicy( 0, 0, 0, 0, 10, 5000
));
manageables.setPolicy( IManagedTcpPolicyManager.MANAGEABLE_NAME, policy );

```

To prevent idle connections to the 10.2.10.* subnet:

```

IManageables manageables = Manageables.getManageables();
ManagedTcpPolicy policy = ( ManagedTcpPolicy )manageables.getPolicy(
IManagedTcpPolicyManager.MANAGEABLE_NAME );
policy.setCasePolicy( new HostAddressRange( "10.2.10.*" ), new CasePolicy( 0, 0, 0, 0, 0, 0 ) );
manageables.setPolicy( IManagedTcpPolicyManager.MANAGEABLE_NAME, policy );
To limit the number of server connections that will be accepted on port 8000 to 7:

```

```

IManageables manageables = Manageables.getManageables();
ManagedTcpPolicy policy = ( ManagedTcpPolicy )manageables.getPolicy(
IManagedTcpPolicyManager.MANAGEABLE_NAME );
policy.setCasePolicy( new HostAddressRange( "://8000" ), new CasePolicy( 0, 0, 7, 0, 0, 0 ) );
manageables.setPolicy( IManagedTcpPolicyManager.MANAGEABLE_NAME, policy );

```

To limit the total number of connections that will be allowed to the test.somewhere.org machine to 2:

```

IManageables manageables = Manageables.getManageables();
ManagedTcpPolicy policy = ( ManagedTcpPolicy )manageables.getPolicy(
IManagedTcpPolicyManager.MANAGEABLE_NAME );
policy.setCasePolicy( new HostAddressRange( "test.somewhere.org" ), new CasePolicy( 2, 0, 0, 0, 0, 0 ) );
manageables.setPolicy( IManagedTcpPolicyManager.MANAGEABLE_NAME, policy );

```

To set the socket retry limit to 3:

```
IManeables manageables = Manageables.getManageables();
ManagedTcpPolicy policy = ( ManagedTcpPolicy )manageables.getPolicy(
IMangedTcpPolicyManager.MANAGEABLE_NAME );
policy.setRetryLimit( 3 );
manageables.setPolicy( IMangedTcpPolicyManager.MANAGEABLE_NAME, policy );
```

Using the TCP Connection Management GUI

Since TCP Connection Management conforms to Voyager's Policy Management framework, the TCP Connection Management GUI will appear in the Voyager Management Console (VMC) for any Voyager ORB where TCP Connection Management is installed. This GUI provides convenient configuration of the TCP Connection Management policy.

Establishing Case Policies

Case policies can be examined from the Case Policies tab in the Connection Management GUI (see ["Figure 8"](#)). This tab displays the address ranges for which case policies have been established. Address ranges are of the format:

hostrange:portrange

Hostrange is an IP-based or name-based range of machines where an asterisk (*) indicates a match with any remaining characters, and portrange is a range of port numbers between 0 and 65535. A dash (-) indicates all ports.

The global case policy, *:-, is always present.

When a case policy is selected in the list, its current properties, the connection limits, and time-out values are displayed below the list. Unrestricted values will say "No limit."

Case policies may be added, removed, and edited by selecting the appropriate button. Any changes made will be displayed, but they will not take effect until the Apply button is selected. Changes may be undone by selecting the Cancel button. Concurrent management consoles do not modify each other, so the Update button can be used to acquire any changes to the current Connection Management policy from the Voyager ORB.



Figure 8

Adding and Editing Case Policies

If a case policy is to be added or edited, a dialog box, shown in "Figure 9", will open displaying the address range of the case policy and the possible restrictions. The address range will be blank for a new case policy; for an existing policy the address range cannot be changed.

TCP Address Range: *.objectspace.com

<input type="checkbox"/>	Live Connections:	
<input type="checkbox"/>	Idle Connections:	
<input checked="" type="checkbox"/>	Server Connections:	25
<input checked="" type="checkbox"/>	Server Idle Time:	30000 ms
<input type="checkbox"/>	Client Connections:	
<input type="checkbox"/>	Client Idle Time:	ms

OK Cancel

Figure 9

Each restriction has a checkbox to the left to indicate if the restriction is being enforced. If the checkbox is selected, then a value must be entered in the corresponding field to indicate the restricted value.

For instance, if the case policy for the range *.objectspace.com is to restrict the number of server connections to 25 and the idle time of server connections to 30 seconds, then the edit dialog should resemble the example in "Figure 9".

Note: If an address range is based on names rather than IP numbers, then significant delays could occur while internet names are resolved via DNS to their proper machines. If you notice significant time delays in your system, check your case policies. You may want to change any name references to IP number references to prevent the name lookups.

Setting the Retry Limit

The second tab in the Connection Management GUI, shown in "Figure 10", is used to set the socket retry limit. This number indicates how many times that Voyager should attempt to make a connection before giving up and admitting failure.

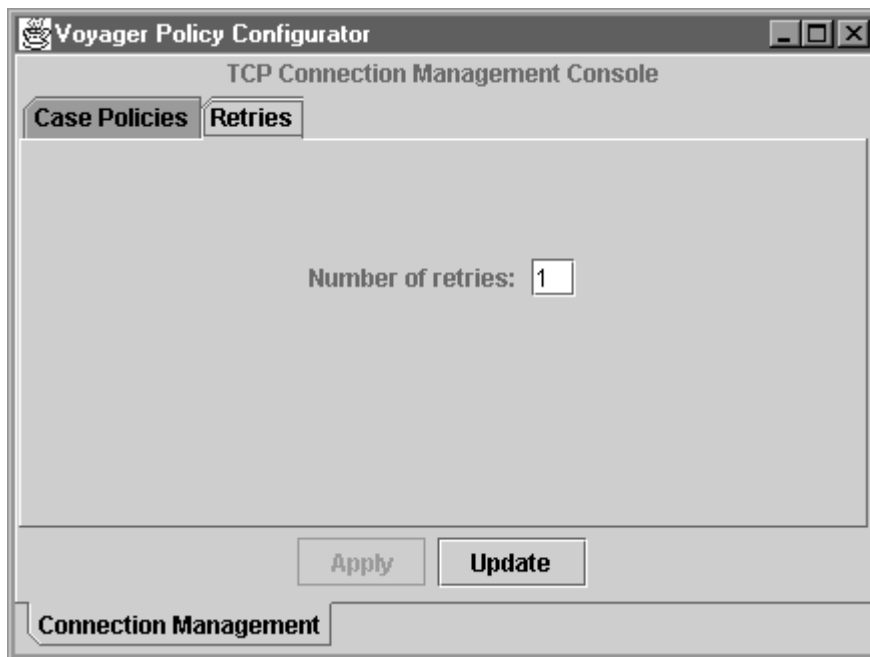


Figure 10

Appendices

Utilities **A**

In this chapter, you will learn to:

- ◆ use the voyager utility to start a Voyager server from the command line
- ◆ use the igen utility to generate a default interface from a class
- ◆ use the cgen utility to translate IDL to and from Java
- ◆ use the pgen utility to generate the source or bytecode form of a proxy for a given class

voyager

The voyager utility starts a Voyager server from the command line. To see usage information, type `voyager` with no arguments. A description of each argument follows:

- ◆ `xurl`

Include the URL that Voyager should use as one of the arguments. This is equivalent to `Voyager.startup(xurl)`.

- ◆ `-a (arguments) option`

Use `-a filename` to process lines in the given file as if they were additional command-line parameters.

- ◆ `-b (bootstrap) classname`

Use `-b classname` to load the specified class using the Voyager class loader and then execute its static `main(String[] args)` method. All subsequent class loading will occur via the Voyager class loader and will leverage Voyager's remote class loading capabilities. By default, `args` is equal to an empty array. To supply `args` to the method call, supply, specify the class within quotes followed by the individual arguments. For example, `voyager -b "mypackage.MyClass 42 hi"`.

- ◆ `-c (class loading) option`

Use `-c url` to enable network class loading from the specified URL. you can specify this option multiple times.



- ◆ `-d (directory configuration) option`

Use `-d url` to configure this Voyager server from the profile at the given directory URL.



- ◆ -f (file)

Use -f *file* with the -j option to specify a file in which directory information will be stored.

- ◆ -i (interpreter) option

By default, voyager is executed by java. To override this default, use the -i option followed by the name of the required interpreter. For example, if you are using the Microsoft development system, you can specify voyager -i jview



- ◆ -j (jndi directory server)

Use -j *root* to configure this server as a JNDI Voyager directory server. This argument must be used in conjunction with the -f argument.

- ◆ -l (log) level

Use -l *loglevel* (silent, exceptions, verbose) to turn on log output at the specified level.

- ◆ -m (map) option

Use -m *idlClass javaClass* to map an IDL entity to a Java class.



- ◆ -p (properties) option

Use -p *filename* to configure Voyager using the specified properties file after all the other command line options are processed.

- ◆ -q (quiet) option

By default, voyager displays a copyright notice. Use the -q option to disable this notice.

- ◆ -r (resource loading) option

Use -r to enable Voyager's built-in HTTP server and allow it to serve classes to other remote Voyager servers. This option is equivalent to `ClassManager.enableResourceServer()`.

- ◆ -s (security manager) option

Use this option to install a Voyager security manager. This option is equivalent to `System.setSecurityManager(new VoyagerSecurityManager())`.

- ◆ `-t` (thread pool size) option

Use `-t <int>` to set the maximum number of idle threads in Voyager's thread pool.

- ◆ `-v` (version) option

Use `-v` to print version information and exit.

- ◆ `-x` (extra parameters) option

Use `-x` to pass the remaining command line arguments through to the Java interpreter.

igen

The igen utility creates a default interface from a class. The interface has all the public methods of the original class, and is named to the original class name prefixed with “I.” For a list of the igen run-time options, run igen from the command line with no parameters.

To use the igen utility, specify the class name without the .class or .java extension. igen searches the directories, .zip files, and .jar files in the CLASSPATH for the specified file and generates an interface for the class. If the current directory contains the source or object code of the original class, typing the full class name is optional. You can generate interfaces for multiple classes by naming the classes separated by a space on the command line.

By default, igen places interfaces created from java.* classes into the related com.objectspace.java.* package. The igen utility reminds you of this behavior with a note each time you run igen on a java.* class.

For example, to create the interface IVector from java.util.Vector, execute the igen utility as shown:

```
>igen java.util.Vector
igen 3.0, copyright objectspace 1997-1999
note: java.* interfaces are placed into com.objectspace.java.*
>_
```

The igen utility generates an interface for all classes in the class’s hierarchy. For example, when igen is run on Hashtable, it generates IHashtable and IDictionary because Hashtable extends Dictionary.

A description of each igen argument follows:

- ◆ -d (directory) option

By default, igen places interfaces into the current working directory. To specify a root directory for igen to store the interfaces, use the -d option followed by the path of the directory you want to use. This option is analogous to the -d option of javac.

For example, to create the interface `\voyager\com\objectspace\java\util\Vector` from `java.util.Vector`, execute the `igen` utility as shown:

```
>igen java.util.Vector -d \voyager
igen 3.0, copyright objectspace 1997-1999
note: java.* interfaces are placed into com.objectspace.java.*
>_
```

- ◆ **-i (interpreter) option**

By default, `igen` is executed by `java`. To override this default, use the `-i` option followed by the name of the required interpreter. For example, if you are using the Microsoft development system, you can specify `igen -i jview`.

- ◆ **-q (quiet) option**

By default, `igen` displays a copyright notice. Use the `-q` option to disable this notice.

- ◆ **-r (remote exception) option**

Voyager supports two exception handling policies. It can throw Voyager-related exceptions, including network and class loading errors, as runtime exceptions (`com.objectspace.voyager.RuntimeRemoteException`) or as checked exceptions (`java.rmi.RemoteException`).

By default, `igen` generates each method to throw a runtime exception unless the original method explicitly throws a `java.rmi.RemoteException` or the `-r` option is specified.

- ◆ **-v (verbose) option**

By default, no status is printed as `igen` operates. Use the `-v` option to view status output.

- ◆ **-x (extra parameters) option**

Use `-x` to pass the remaining command line arguments through to the Java interpreter.

cgen

The cgen utility can translate IDL files to and from Java. For detailed information about this translation, consult the "[Java to IDL](#)" section in [Appendix B](#).

For a list of the cgen run-time options, run cgen from the command line with no parameters.

To translate an IDL file to Java, execute cgen with a list of the IDL files. The file names should end with a .idl extension.

To translate a Java file into IDL, execute cgen with a list of .java and/or .class files, omitting any extension. The cgen utility searches the directories, .zip files, and .jar files in the CLASSPATH for the specified file and generates IDL for the class. If the current directory contains the source or object code of the original class, typing the full class name is optional.

A description of each cgen option follows:

- ◆ -a (argument) option

Use -a *filename* to process lines in the given file as if they were additional command-line parameters.

- ◆ -d (directory) option

By default, cgen places interfaces created from .java files into the current working directory. The -d path option allows you to specify a different root directory in which to store the holder classes. This option is analogous to the -d option of javac.

- ◆ -f (flat) option

By default, cgen outputs IDL #include statements to match their module structure. Use the -f option to force a flat #include structure. For example, the IDL interface MyModule.MyInterface would be included using #include "MyInterface.idl" instead of #include "MyModuleMyInterface.idl".

- ◆ -h (holder) option

By default, cgen creates holders for enums, structs, unions, interfaces, and typedefs of sequences and arrays. Use the -h option to disable automatic holder creation.

- ◆ -i (interpreter) option

By default, cgen is executed by java. To override this default, use the -i option followed by the name of the required interpreter. For example, if you are using the Microsoft development system, you can specify `cgen -i jview`.

- ◆ -I (Include) option

Use the -I path option to add the specified path to the list of directories that cgen searches when looking for #include files that are relative. By default, cgen only searches relative to the current directory.

- ◆ -m (map) option

Use `-m idlClass javaClass` to map an IDL entity to and from a Java class.

- ◆ -q (quiet) option

By default, cgen displays a copyright notice. Use the -q option to disable this notice.

- ◆ -r (remote) option

Use the -r option to use `java.rmi.Remote` instead of `IRemote` for pass-by-reference.

- ◆ -t (transaction) option

Use -t to make the IDL interface extend `CosTransactions::TransactionalObject`.

- ◆ -v (verbose) option

By default, no status is printed as cgen operates. Use the -v option to view status output.

- ◆ -x (extra parameters) option

Use -x to pass the remaining command line arguments through to the Java interpreter.

pgen



The pgen utility generates the source or bytecode form of a proxy for a given class.

Manual Proxy Class Generation

Dynamic proxy generation is most useful as an aid to development. Eliminating manual proxy class generation, and the associated class synchronization problems, results in a significant increase in development speed. Use *manual* proxy generation when performance is critical and when proxy classes need to be post-processed.

Performance

Generation of the average proxy class typically takes about 250 milliseconds. After the proxy class for a given class has been generated, it never needs to be generated again for the lifetime of the VM process. Though this one-time hit is rarely noticeable over the lifetime of a proxy class, it can have a noticeable impact on the perceived system performance at proxy generation time, particularly when many classes are processed at once, such as in system startup. By generating proxy classes manually, proxy class loading becomes equivalent to loading of any class, and is unnoticeable.

Typically, you should use automatic proxy class generation during development, and manual proxy class generation for system deployment.

Post-Processing

Some designs require modifications of each proxy class to allow custom functionality. For example, you can modify each proxy method to print to a log when invoked. These modifications are impossible to make to proxies generated on the fly. By generating proxy source code, you can substitute your own logic for the default proxy logic, compile the proxy, and allow each Voyager application to load the custom proxy class.

pgen Utility Options

Use the pgen utility to generate the source or bytecode form of a proxy for a given class. For a list of the pgen run-time options, run pgen from the command line with no parameters.

To create the proxy class for java.util.Vector, execute the pgen utility as shown:

```
>pgen java.util.Vector
pgen 3.0, copyright objectspace 1997-1999
note: java.* proxy classes are placed into com.objectspace.java.*
>_
```

You can generate the proxy classes for multiple classes by naming the classes separated by a space on the command line.

- ◆ -d (directory) option

By default, pgen places proxy classes into the current working directory. The -d option allows you to specify a different directory in which to store the proxy class. This option is analogous to the -d option of javac.

- ◆ -i (interpreter) option

By default, pgen is executed by java. To override this default, use the -i option followed by the name of the required interpreter. For example, if you are using the Microsoft development system, you can specify pgen -i jview.

- ◆ -q (quiet) option

By default, pgen displays a copyright notice. Use the -q option to disable this notice.

- ◆ -v (verbose) option

By default, no status is displayed as pgen operates. Use the -v option to view status output.

- ◆ -s (source) option

By default, pgen generates executable Java .class files. However, to facilitate manual post-processing of proxies, pgen allows Java source code to be generated instead. Use -s option to generate the proxy classes in source form.

Examples **B**

This section contains an explanation, source code, and output for the examples in this manual.

Running the Examples

After you install Voyager, the source code for these examples is located in the examples directories. Each example description specifies the directory in which the example resides. The CLASSPATH must include voyage to run the examples.

Each example is presented as follows:

1. The command(s) used to prepare the example program for execution are presented. Commands that generate interfaces, generate holders, and compile Java source code belong in this category.
2. The command(s) used to run the example program are presented, followed by the program output.
3. The source code for the example programs is listed.

Commands the user types and the resulting output displayed are presented in a window as shown:.

>Command typed at prompt
Resulting output displayed to screen
>

Sometimes, not all output from a command displays to the screen at once. When subsequent output is presented in a window, the original command and output text are shaded gray, and new output is presented in bold. For example:

>Command typed at prompt
Resulting output displayed to screen
More output displayed to screen
>

Basics

The examples in this section demonstrate basic Voyager functions like messaging, remote construction, naming and lookup, and remote class loading.

Basics1 Example

The Basics1 example uses Voyager's remote construction mechanism to construct a Stockmarket object in a remote Voyager server. It then sends the object messages. The last message demonstrates how exceptions thrown on the server are transparently propagated to the client.

Note: The remote server does not terminate automatically. A Voyager server must be terminated explicitly, preferably by using `Voyager.shutdown()`.

From `examples\stockmarket`, compile the stock market files:

```
javac IStockmarket.java Stockmarket.java
```

From `examples\basics`, compile the example program:

```
javac Basics1.java
```

Start a Voyager server on port 8000 in one window. Run Basics1 in a second window.

Window1

```
>voyager 8000
voyager 3.1, copyright objectspace 1997-1999
construct stockmarket
news: Sun releases Java
```

Window2

```
>java examples.basics.Basics1  
sun share price = 103  
bought 10 shares of SUN for 960  
java.lang.IllegalArgumentException: share count < 0  
  
>
```

The source code for IStockmarket.java, Stockmarket.java, and Basics1.java follows. IStockmarket and Stockmarket are used in other examples throughout the guide, but their source is only listed here.

Interface examples\stockmarket\IStockmarket.java

```
// copyright 1997, 1998 objectspace
```

```
package examples.stockmarket;  
  
public interface IStockmarket  
{  
    int quote( String symbol );  
    int buy( int shares, String symbol );  
    int sell( int shares, String symbol );  
    void news( String announcement );  
}
```

Class examples\stockmarket\Stockmarket.java

```
// copyright 1997, 1998 objectspace
```

```
package examples.stockmarket;  
  
import java.util.*;  
import java.io.*;  
  
public class Stockmarket implements IStockmarket, Serializable  
{  
    static Random random = new Random();  
    private Hashtable prices = new Hashtable(); // symbol -> price
```



```

public Stockmarket()
{
    System.out.println( "construct stockmarket" );
}

public int quote( String symbol )
{
    Integer price = (Integer) prices.get( symbol ); // current price

    if( price == null )
    {
        // calculate random initial price 20..120
        price = new Integer( Math.abs( random.nextInt() ) % 100 + 20 );
    }
    else
    {
        // raise or lower the price by up to 20%
        double factor = 1.0 + (random.nextInt() % 20) / 100.0;
        price = new Integer( (int) (price.intValue() * factor) );
    }

    prices.put( symbol, price ); // store new price
    return price.intValue(); // return new price of stock
}

public int sell( int shares, String symbol )
{
    if( shares < 0 )
        throw new IllegalArgumentException( "share count < 0" );

    return shares * quote( symbol ); // return total
}

public int buy( int shares, String symbol )
{
    if( shares < 0 )
        throw new IllegalArgumentException( "share count < 0" );

    return shares * quote( symbol ); // return total
}

public void news( String announcement )

```

```

{
    System.out.println( "news: " + announcement ); // display news
}
}

```

Program examples\basics\Basics1.java

// copyright 1997-1999 objectspace

package examples.basics;

import examples.stockmarket.*;
import com.objectspace.voyager.*;

```

public class Basics1
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup(); // startup as client
            // create a remote stockmarket
            final String classname = "examples.stockmarket.Stockmarket";
            IStockmarket market = (IStockmarket) Factory.create( classname, "localhost:8000" );
            // send messages to the remote object
            market.news( "Sun releases Java" );
            System.out.println( "sun share price = " + market.quote( "SUN" ) );
            int buyprice = market.buy( 10, "SUN" );
            System.out.println( "bought 10 shares of SUN for " + buyprice );
            market.sell( -4, "SUN" ); // cause an exception
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }

        Voyager.shutdown();
    }
}

```

Basics2A and Basics2B Examples

The Basics2A and Basics2B examples demonstrate sending a remote message to a proxy obtained by a Namespace lookup. The Basics2A program exports the object to a specified port and binds the associated proxy into the NameSpace. Basics2B looks up the object and sends it a message.

From examples\basics, compile the example program:

```
javac Basics2A.java Basics2B.java
```

Run Basics2A in one window. Run Basics2B in a second window.

Window1

```
>java examples.basics.Basics2A  
construct stockmarket  
news: Sun releases Java
```

Window2

```
>java examples.basics.Basics2B  
  
>
```

The source code for Basics2A.java and Basics2B.java follows:

Program examples\basics\Basics2A.java

```
// copyright 1997-1999 objectspace
```

```
package examples.basics;
```

```
import examples.stockmarket.*;  
import com.objectspace.voyager.*;
```

```
public class Basics2A
```

```

{
public static void main( String[] args )
{
    try
    {
        Voyager.startup();
        // export a stockmarket on port 9000
        IStockmarket market = (IStockmarket) Proxy.export( new Stockmarket(), "9000" );
        // bind "NASDAQ" to the stockmarket
        Namespace.bind( "9000/NASDAQ", market );
    }
    catch( Exception exception )
    {
        System.err.println( exception );
    }
}
}

```

Program examples\basics\Basics2B.java

// copyright 1997-1999 objectspace

package examples.basics;

import examples.stockmarket.*;
import com.objectspace.voyager.*;

```

public class Basics2B
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            // obtain a proxy to the object on port 9000 with symbol "NASDAQ"
            IStockmarket market = (IStockmarket) Namespace.lookup( "//localhost:9000/NASDAQ" );
            // send a message to the remote object
            market.news( "Sun releases Java" );
        }
        catch( Exception exception )
        {

```

```
        System.err.println( exception );  
    }  
}  
}
```

Dynamic Aggregation

The examples in this section explain Voyager's dynamic aggregation framework.

Aggregation1A and Aggregation1B Examples

These examples demonstrate facet creation and remote access of facets. An Account facet is added to an Employee object in Aggregation1A. It is then accessed remotely in Aggregation1B.

Compile all .java files in examples\aggregation:

```
javac *.java
```

Run the Aggregation1A example in one window. Run Aggregation1B in a second window.

Window1

```
>java examples.aggregation.Aggregation1A  
primary = Employee( joe, 234-44-2678 )  
facet 0 = Account( 2000 )
```

Window2

```
>java examples.aggregation.Aggregation1B  
account = Account( 2000 )  
  
>
```

The source code for IEmployee.java, Employee.java, IAccount.java, Account.java, Aggregation1A.java, and Aggregation1B.java follows:

Interface examples\aggregation\IEmployee.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public interface IEmployee
{
    String getName();
    String getSSN();
}
```

Class examples\aggregation\Employee.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public class Employee implements IEmployee, com.objectspace.voyager.IRemote
{
    String name;
    String ssn;

    public Employee( String name, String ssn )
    {
        this.name = name;
        this.ssn = ssn;
    }

    public String toString()
    {
        return "Employee( " + name + ", " + ssn + " )";
    }

    public String getName()
    {
        return name;
    }

    public String getSSN()
    {
        return ssn;
    }
}
```

Interface examples\aggregation\IAccount.java

// copyright 1997-1999 objectspace

package examples.aggregation;

```
public interface IAccount
{
    void deposit( int amount );
    void withdraw( int amount );
    int getBalance();
}
```

Class examples\aggregation\Account.java

// copyright 1997-1999 objectspace

package examples.aggregation;

public class Account implements IAccount

```
{
    int balance;

    public String toString()
    {
        return "Account( " + balance + " )";
    }
}
```

```
public void deposit( int amount )
{
    balance += amount;
}
```

```
public void withdraw( int amount )
{
    if( amount > balance )
        throw new IllegalArgumentException( "only have $" + amount );

    balance -= amount;
}
```

```
public int getBalance()
```



```

    {
        return balance;
    }
}

```

Program examples\aggregation\Aggregation1A.java

// copyright 1997-1999 objectspace

```

package examples.aggregation;

import com.objectspace.lib.facets.*;
import com.objectspace.voyager.*;

public class Aggregation1A
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup( "8000" );
            IEmployee employee = new Employee( "joe", "234-44-2678" );
            IFacets facets = Facets.of( employee );
            IAccount account = (IAccount) facets.of( "examples.aggregation.IAccount" );
            account.deposit( 2000 );
            System.out.println( "primary = " + facets.getPrimary() );
            Object[] array = facets.getFacets();

            for( int i = 0; i < array.length; i++ )
                System.out.println( "facet " + i + " = " + array[ i ] );

            Namespace.bind( "Joe", employee );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }
    }
}

```

Program examples\aggregation\Aggregation1B.java

```
// copyright 1997-1999 objectspace

package examples.aggregation;

import com.objectspace.lib.facets.*;
import com.objectspace.voyager.*;

public class Aggregation1B
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            IEmployee employee = (IEmployee) Namespace.lookup( "//localhost:8000/Joe" );
            IAccount account = (IAccount) Facets.of( employee ).of( "examples.aggregation.IAccount" );
            System.out.println( "account = " + account );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }

        Voyager.shutdown();
    }
}
```

Aggregation2A and Aggregation2B Examples

These examples differ from the ["Aggregation1A and Aggregation1B Examples"](#) because they demonstrate the use of the `of()` and `get()` convenience methods. Not only do they present a cleaner API for frequently used aggregation operations, they also ensure compile-time checking of facet class types, as opposed to run-time parsing of String names.

Assuming the files have been compiled for the previous example, run the Aggregation2A example in one window. Run Aggregation2B in a second window.

Window1

```
>java examples.aggregation.Aggregation2A  
employee = Employee( ted, 224-55-1567 )  
security = Security( 1, putty )
```

Window2

```
>java examples.aggregation.Aggregation2B  
security = Security( 1, putty )  
  
>
```

The source code for ISecurity.java, Security.java, Aggregation2A.java and Aggregation2B.java follows:

Interface examples\aggregation\ISecurity.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public interface ISecurity  
{  
    int getClearance();  
    void setClearance( int clearance );  
    String getCode();  
    void setCode( String code );  
}
```

Class examples\aggregation\Security.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public class Security implements ISecurity
```

```

{
int clearance;
String code;

public int getClearance()
{
return clearance;
}

public void setClearance( int clearance )
{
this.clearance = clearance;
}

public String getCode()
{
return code;
}

public void setCode( String code )
{
this.code = code;
}

public String toString()
{
return "Security( " + clearance + ", " + code + " )";
}

static public ISecurity get( Object object )
{
// convenience method
return (ISecurity) com.objectspace.voyager.Facets.get( object, ISecurity.class );
}

static public ISecurity of( Object object ) throws ClassCastException
{
// convenience method
return (ISecurity) com.objectspace.voyager.Facets.of( object, ISecurity.class );
}
}

```

Program examples\aggregation\Aggregation2A.java

```
// copyright 1997-1999 objectspace

package examples.aggregation;

import com.objectspace.lib.facets.*;
import com.objectspace.voyager.*;

public class Aggregation2A
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup( "8000" );
            IEmployee employee = new Employee( "ted", "224-55-1567" );
            ISecurity security = Security.of( employee );
            security.setClearance( 1 );
            security.setCode( "putty" );
            System.out.println( "employee = " + employee );
            System.out.println( "security = " + security );
            Namespace.bind( "Ted", employee );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }
    }
}
```

Program examples\aggregation\Aggregation2B.java

```
// copyright 1997-1999 objectspace

package examples.aggregation;

import com.objectspace.voyager.*;

public class Aggregation2B
{
    public static void main( String[] args )
```

```

{
try
{
    Voyager.startup();
    IEmployee employee = (IEmployee) Namespace.lookup( "//localhost:8000/Ted" );
    ISecurity security = Security.of( employee );
    System.out.println( "security = " + security );
}
catch( Exception exception )
{
    System.err.println( exception );
}

    Voyager.shutdown();
}
}

```

Aggregation3 Example

This example demonstrates custom facet to class mapping. In the example, instances of class `Employee` get a custom facet of type `EmployeeBonusPlan`, whereas instances of `Programmer` get a custom facet of type `ProgrammerBonusPlan`. An attempt is made to get a `BonusPlan` facet for an instance of `String`. Because no custom facet class is found using the class-matching rules, and there is no default `BonusPlan` facet class, an exception is thrown.

Assuming the files have been compiled for the first aggregation example, run the `Aggregation3` example.

Window1

```

>java examples.aggregation.Aggregation3
plan1 = examples.aggregation.EmployeeBonusPlan@1f4585
plan2 = examples.aggregation.ProgrammerBonusPlan@1f3a24
java.lang.ClassCastException: java.lang.ClassNotFoundException:
examples.aggregation.BonusPlan

```

The source code for IProgrammer.java, Programmer.java, IBonusPlan.java, BonusPlan.java, EmployeeBonusPlan.java, ProgrammerBonusPlan.java, and Aggregation3.java follows:

Interface examples\aggregation\IProgrammer.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public interface IProgrammer extends IEmployee
{
    String getLanguage();
}
```

Class examples\aggregation\Programmer.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public class Programmer extends Employee implements IProgrammer
{
    String language;

    public Programmer( String name, String ssn, String language )
    {
        super( name, ssn );
        this.language = language;
    }

    public String getLanguage()
    {
        return language;
    }
}
```

Interface examples\aggregation\IBonusPlan.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public interface IBonusPlan
{
    int getBonus();
}
```

```
}
```

Class examples\aggregation\BonusPlan.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public class BonusPlan
{
    static public IBonusPlan get( Object object )
    {
        // convenience method
        return (IBonusPlan) com.objectspace.voyager.Facets.get( object, IBonusPlan.class );
    }

    static public IBonusPlan of( Object object ) throws ClassCastException
    {
        // convenience method
        return (IBonusPlan) com.objectspace.voyager.Facets.of( object, IBonusPlan.class );
    }
}
```

Class examples\aggregation\EmployeeBonusPlan.java

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
public class EmployeeBonusPlan implements IBonusPlan
{
    int bonus = 1000; // starting bonus

    public int getBonus()
    {
        int tmp = bonus;
        bonus += 1000; // bump by 1000 for next time
        return tmp;
    }
}
```

Class examples\aggregation\ProgrammerBonusPlan.java

```
// copyright 1997-1999 objectspace
```



```

package examples.aggregation;

public class ProgrammerBonusPlan implements IBonusPlan
{
    int bonus = 500; // initial bonus

    public int getBonus()
    {
        int tmp = bonus;
        bonus *= 2; // double for next time
        return tmp;
    }
}

```

Program examples\aggregation\Aggregation3.java

// copyright 1997-1999 objectspace

```

package examples.aggregation;

import com.objectspace.voyager.*;

public class Aggregation3
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            IEmployee employee = new Employee( "sandra", "234-33-7812" );
            IProgrammer programmer = new Programmer( "jeff", "211-45-1458", "java" );
            // get/add the appropriate facet for each employee
            IBonusPlan plan1 = BonusPlan.of( employee );
            System.out.println( "plan1 = " + plan1 );
            IBonusPlan plan2 = BonusPlan.of( programmer );
            System.out.println( "plan2 = " + plan2 );
            // exception since there is no matching facet type
            IBonusPlan plan3 = BonusPlan.of( "hi" );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }
    }
}

```

```

    Voyager.shutdown();
  }
}

```

Aggregation4 Example

This example demonstrates the facet-aware facet class `ManagerBonusPlan`. When an instance of `BonusPlan` is aggregated with an instance of `Manager`, the `ManagerBonusPlan` class is chosen. Because it implements `IFacet` and provides the correct constructor, it is constructed with the Facets object of the `Manager` primary object.

Assuming the files have been compiled for the first aggregation example, run the `Aggregation4` example.

Window1

```

>java examples.aggregation.Aggregation4
create manager bonus plan facet
primary = Employee( deborah, 622-45-8711 )
plan = examples.aggregation.ManagerBonusPlan@1f46f4

```

The source code for `IManager.java`, `Manager.java`, `ManagerBonusPlan.java`, and `Aggregation4.java` follows:

Interface `examples\aggregation\IManager.java`

```
// copyright 1997-1999 objectspace
```

```
package examples.aggregation;
```

```
import java.util.Vector;
```

```
public interface IManager extends IEmployee
{
    void addEmployee( IEmployee employee );
    Vector getEmployees();
}

```

Class examples\aggregation\Manager.java

```
// copyright 1997-1999 objectspace

package examples.aggregation;

import java.util.Vector;

public class Manager extends Employee implements IManager
{
    Vector employees = new Vector();

    public Manager( String name, String ssn )
    {
        super( name, ssn );
    }

    public void addEmployee( IEmployee employee )
    {
        employees.addElement( employee );
    }

    public Vector getEmployees()
    {
        return employees;
    }
}
```

Class examples\aggregation\ManagerBonusPlan.java

```
// copyright 1997-1999 objectspace

package examples.aggregation;

import com.objectspace.lib.facets.*;

public class ManagerBonusPlan implements IBonusPlan, IFacet
{
    public ManagerBonusPlan( IFacets facets )
    {
        System.out.println( "create manager bonus plan facet" );
        System.out.println( "primary = " + facets.getPrimary() );
    }
}
```

```

public int getBonus()
{
    return 0; // heh, heh!
}

public boolean isTransient()
{
    return true; // facet is stateless, so can discard
}
}

```

Program examples\aggregation\Aggregation4.java

// copyright 1997-1999 objectspace

package examples.aggregation;

import com.objectspace.voyager.*;

```

public class Aggregation4
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            IManager manager = new Manager( "deborah", "622-45-8711" );
            IBonusPlan plan = BonusPlan.of( manager );
            System.out.println( "plan = " + plan );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }

        Voyager.shutdown();
    }
}

```

Advanced Messaging

The examples in this section demonstrate more advanced forms of remote messaging such as one-way, future, remote static, and dynamic synchronous invocation.

Message1 Example

The Message1 example demonstrates dynamic invocation for remote static and instance method invocation. It creates an Alarm object in a remote Voyager server. It then invokes an instance method synchronously. Next, it invokes a static method on the Alarm class in the remote Voyager server.

From examples\message, compile the example program:

```
javac IAlarm.java Alarm.java Message1.java
```

Start a Voyager server on port 8000 in one window. Run Message1 in a second window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
delay( 5000 )  
done!  
wakeup( 5000 )  
done!
```

Window2

```
>java examples.message.Message1  
result1 = 5000  
result2 = 5000  
  
>
```

The source code for IAlarm.java, Alarm.java, and Message1.java follows:

Interface examples\message\IAAlarm.java

// copyright 1997-1999 objectspace

package examples.message;

```
public interface IAAlarm
{
    int delay( int milliseconds );
}
```

Class examples\message\Alarm.java

// copyright 1997-1999 objectspace

package examples.message;

```
public class Alarm implements IAAlarm
{
    public int delay( int milliseconds )
    {
        System.out.println( "delay( " + milliseconds + " )" );

        if( milliseconds < 0 )
            throw new IllegalArgumentException( "delay < 0" );

        try{ Thread.sleep( milliseconds ); } catch( InterruptedException exception ) {}
        System.out.println( "done!" );
        return milliseconds;
    }

    public static int wakeup( int milliseconds )
    {
        System.out.println( "wakeup( " + milliseconds + " )" );
        try{ Thread.sleep( milliseconds ); } catch( InterruptedException exception ) {}
        System.out.println( "done!" );
        return milliseconds;
    }

    public static Alarm newAlarm()
    {
        return new Alarm();
    }
}
```

```
}
```

Program examples\message\Message1.java

```
// copyright 1997-1999 objectspace
```

```
package examples.message;
```

```
import com.objectspace.voyager.*;
```

```
import com.objectspace.voyager.message.*;
```

```
public class Message1
```

```
{
```

```
    public static void main( String[] args )
```

```
    {
```

```
        try
```

```
        {
```

```
            Voyager.startup();
```

```
            // create a remote alarm
```

```
            IAlarm alarm = (IAlarm) Factory.create( "examples.message.Alarm", "://localhost:8000" );
```

```
            // invoke sync instance method
```

```
            Object[] delay = new Object[]{ new Integer( 5000 ) };
```

```
            Result result1 = Sync.invoke( alarm, "delay", delay );
```

```
            System.out.println( "result1 = " + result1.readInt() );
```

```
            // invoke sync class method
```

```
            Result result2 = Sync.invoke( "examples.message.Alarm", "wakeup", delay, "://localhost:8000" );
```

```
            System.out.println( "result2 = " + result2.readInt() );
```

```
        }
```

```
    catch( Exception exception )
```

```
    {
```

```
        System.err.println( exception );
```

```
    }
```

```
    Voyager.shutdown();
```

```
}
```

```
}
```

Message2 Example

The Message2 example demonstrates invocation of a one-way message. It creates an Alarm object in a remote Voyager server. It then invokes a one-way instance method on

the alarm. The example pauses one second to delay shutdown to allow time for the one-way invocation to fully flush from the program.

From examples\message, compile the example program:

```
javac Message2.java
```

Start a Voyager server on port 8000 in one window. Run Message2 in a second window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
delay( 5000 )  
done!
```

Window2

```
>java examples.message.Message2  
  
>
```

The source code for Message2.java follows:

Program examples\message\Message2.java

```
// copyright 1997-1999 objectspace
```

```
package examples.message;
```

```
import com.objectspace.voyager.*;  
import com.objectspace.voyager.message.*;
```

```
public class Message2  
{  
    public static void main( String[] args )  
    {  
        try
```



```

{
    Voyager.startup();
    // create remote alarm
    IAlarm alarm = (IAlarm) Factory.create( "examples.message.Alarm", "//localhost:8000" );
    // invoke oneway instance method
    OneWay.invoke( alarm, "delay", new Object[]{ new Integer( 5000 ) } );
    Thread.sleep( 1000 ); // allow message to flush
}
catch( Exception exception )
{
    System.err.println( exception );
}

Voyager.shutdown();
}
}

```

Message3 Example

The Message3 example demonstrates invocation of a future message. It creates an Alarm object in a remote Voyager server. It then invokes a future instance method on the alarm. Because the future invocation is asynchronous, the program can to execute other code while the message is being delivered. It then executes a blocking read on the result. Next, the program demonstrates how reading a result from a future can re-throw any exception that occurs during delivery or execution of the future message.

From examples\message, compile the example program:

```
javac Message3.java
```

Start a Voyager server on port 8000 in one window. Run Message3 in a second window.

Window1

```

>voyager 8000
voyager 3.1, copyright objectspace 1997-1999
delay( 5000 )
done!
delay( -1 )

```

Window2

```
>java examples.message.Message3
about to send delay( 5000 )
available = false
result = 5000
available = true
exception = false
about to send delay( -1 )
alarm.delay( -1 ) -> java.lang.IllegalArgumentException: delay < 0

>
```

The source code for Message3.java follows:

Program examples\message\Message3.java

```
// copyright 1997-1999 objectspace

package examples.message;

import com.objectspace.voyager.*;
import com.objectspace.voyager.message.*;

public class Message3
{
    public static void main( String args[] )
    {
        try
        {
            Voyager.startup();
            IAlarm alarm = (IAlarm) Factory.create( "examples.message.Alarm", "localhost:8000" );

            // demonstrate blocking reads
            try
            {
                System.out.println( "about to send delay( 5000 )" );
                Result result = Future.invoke( alarm, "delay", new Object[]{ new Integer( 5000 ) } );

                // check to see if the return value is available
                System.out.println( "available = " + result.isAvailable() );
            }
            catch ( Exception e )
            {
                System.out.println( "Exception: " + e );
            }
        }
        catch ( Exception e )
        {
            System.out.println( "Exception: " + e );
        }
    }
}
```

```

// execute a blocking read for the return value
int value = result.readInt();

// display the return value and the current status of the result
System.out.println( "result = " + value );
System.out.println( "available = " + result.isAvailable() );
System.out.println( "exception = " + result.isException() );
}
catch( Exception exception )
{
    System.err.println( "alarm.delay( 5000 ) -> " + exception );
}

// demonstrate a thrown exception
try
{
    System.out.println( "about to send delay( -1 )" );
    Result result = Future.invoke( alarm, "delay", new Object[]{ new Integer( -1 ) } );
    int value = result.readInt();
    System.out.println( "result = " + value );
}
catch( Exception exception )
{
    System.err.println( "alarm.delay( -1 ) -> " + exception );
}
}
catch( Exception exception )
{
    System.err.println( exception );
}

Voyager.shutdown();
}
}

```

Message4 Example

The Message4 example demonstrates invocation of a future message with listeners. It creates an Alarm object in a remote Voyager server. It then invokes a future instance

method on the alarm, passing in an array of listeners. These listeners receive a callback when the result of the future invocation is received.

From examples\message, compile the example program:

```
javac Message4.java
```

Start a Voyager server on port 8000 in one window. Run Message4 in a second window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
delay( 5000 )  
done!
```

Window2

```
>java examples.message.Message4  
send delay( 5000 )  
listener gets result event  
event source = Result( 5000 )  
object = 5000  
exception = false  
listener gets result event  
event source = Result( 5000 )  
object = 5000  
exception = false  
value = 5000  
  
>
```

The source code for Message4.java follows:

Program examples\message\Message4.java

```
// copyright 1997-1999 objectspace
```

```
package examples.message;
```

```

import com.objectspace.voyager.*;
import com.objectspace.voyager.message.*;

public class Message4
{
    public static void main( String args[] )
    {
        try
        {
            Voyager.startup();
            IAlarm alarm = (IAlarm) Factory.create( "examples.message.Alarm", "localhost:8000" );

            // create two listeners capable of getting a callback when the invocation completes
            ResultListener[] listeners = new ResultListener[ 2 ];
            listeners[ 0 ] = new MyResultListener();
            listeners[ 1 ] = new MyResultListener();

            // add as a listener to the result of the future invocation
            System.out.println( "send delay( 5000 )" );
            Result result = Future.invoke( alarm, "delay", new Object[]{ new Integer( 5000 ) }, false, 0, listeners );
            System.out.println( "value = " + result.readInt() );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }

        Voyager.shutdown();
    }
}

class MyResultListener implements ResultListener
{
    public void resultReceived( ResultEvent event )
    {
        System.out.println( "listener gets result event" );
        System.out.println( "event source = " + event.getSource() );
        System.out.println( "object = " + event.getObject() );
        System.out.println( "exception = " + event.isException() );
    }
}

```

Message5 Example

The Message5 example demonstrates invocation of a future message with two threads blocking on the result. When the result of the future invocation is received, both blocking threads read the return value.

From examples\message, compile the example program:

```
javac Message5.java
```

Start a Voyager server on port 8000 in one window. Run Message5 in a second window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
delay( 5000 )  
done!
```

Window2

```
>java examples.message.Message5  
about to send delay( 5000 )  
waiting...  
waiting...  
reader thread gets 5000  
reader thread gets 5000  
  
>
```

The source code for Message5.java follows:

Program examples\message\Message5.java

```
// copyright 1997-1999 objectspace
```

```
package examples.message;
```

```
import com.objectspace.voyager.*;  
import com.objectspace.voyager.message.*;
```

```

public class Message5
{
    public static void main( String args[] )
    {
        try
        {
            Voyager.startup();
            IAlarm alarm = (IAlarm) Factory.create( "examples.message.Alarm", "localhost:8000" );
            System.out.println( "about to send delay( 5000 )" );
            Result result = Future.invoke( alarm, "delay", new Object[] { new Integer( 5000 ) } );

            // simulate two different application threads blocking on the same future invocation
            Thread thread1 = new ReaderThread( result );
            thread1.start();
            Thread thread2 = new ReaderThread( result );
            thread2.start();
            thread1.join();
            thread2.join();
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }

        Voyager.shutdown();
    }
}

class ReaderThread extends Thread
{
    Result result;

    ReaderThread( Result result )
    {
        this.result = result;
    }

    public void run()
    {
        try
        {

```

```

        System.out.println( "waiting..." );
        int value = result.readInt();
        System.out.println( "reader thread gets " + value );
    }
    catch( Exception exception )
    {
        System.err.println( exception );
    }
}
}

```

Message6 Example

The Message6 example demonstrates invocation of a future message with a timeout. The message is designed to take longer than the timeout value. Consequently, a `TimeoutException` is thrown.

From `examples\message`, compile the example program:

```
javac Message6.java
```

Start a Voyager server on port 8000 in one window. Run Message6 in a second window.

Window1

```

>voyager 8000
voyager 3.1, copyright objectspace 1997-1999
delay( 5000 )
done!

```

Window2

```

>java examples.message.Message6
send delay( 5000 )
com.objectspace.voyager.message.TimeoutException: future timed out after 3000ms

>

```

The source code for Message6.java follows:

Program examples\message\Message6.java

// copyright 1997-1999 objectspace

```
package examples.message;

import com.objectspace.voyager.*;
import com.objectspace.voyager.message.*;

public class Message6
{
    public static void main( String args[] )
    {
        try
        {
            Voyager.startup();
            IAlarm alarm = (IAlarm) Factory.create( "examples.message.Alarm", "///localhost:8000" );
            System.out.println( "send delay( 5000 )" );
            // invoke method with 3 second timeout
            Result result = Future.invoke( alarm, "delay", new Object[] { new Integer( 5000 ) }, false, 3000, null );
            int value = result.readInt();
            System.out.println( "value = " + value );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }

        Voyager.shutdown();
    }
}
```

Message7 Example

The Message7 example demonstrates a special form of dynamic invocation that returns results by reference instead of by value. The program invokes a static method on the Alarm class in a remote Voyager server. This method returns a new Alarm object; however, the invocation uses return by reference, so the client receives a proxy to the remote alarm object instead of the alarm object itself.

From examples\message, compile the example program:

```
javac Message7.java
```

Start a Voyager server on port 8000 in one window. Run Message7 in a second window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
delay( 5000 )  
done!
```

Window2

```
>java examples.message.Message7  
  
>
```

The source code for Message7.java follows:

Program examples\message\Message7.java

```
// copyright 1997-1999 objectspace
```

```
package examples.message;
```

```
import com.objectspace.voyager.*;  
import com.objectspace.voyager.message.*;
```

```
public class Message7  
{  
    public static void main( String args[] )  
    {  
        try  
        {  
            Voyager.startup();  
            // request a proxy to the result  
            Result result = Sync.invoke( "examples.message.Alarm", "newAlarm", null, "localhost:8000", true );
```

```
// read the proxy to the return result
IAlarm alarm = (IAlarm) result.readObject();
// alarm is a proxy to the remote result
alarm.delay( 5000 );
}
catch( Exception exception )
{
    System.err.println( exception );
}

Voyager.shutdown();
}
}
```

Multicast and Publish/Subscribe

The examples in this section demonstrate Voyager's multicast and publish/subscribe features.

Space1 Example

The Space1 example demonstrates constructing and populating a space. It first constructs a subspace in a remote Voyager server on port 8000 and populates it with Consumer objects constructed in the same server. It then constructs another subspace in a remote Voyager server on port 9000 and populates it with Consumer objects. Finally, the two subspaces are connected to form a single distributed space.

Compile all .java files in examples\space:

```
javac *.java
```

Start a Voyager server on port 8000 in one window and a Voyager server on port 9000 in a second window. Run Space1 in a third window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
created Consumer(jack)  
created Consumer(sasha)
```

Window2

```
>voyager 9000  
voyager 3.1, copyright objectspace 1997-1999  
created Consumer(simon)  
created Consumer(galina)
```

Window3

```
>java examples.space.Space1
```

```
>
```

The source code for `IConsumer.java`, `Consumer.java`, and `Space1.java` follows. See the ["Space2 Example"](#) for the `NewsListener` source code.

Interface examples\space\IConsumer.java

```
// copyright 1997-1999 objectspace
```

```
package examples.space;
```

```
public interface IConsumer extends NewsListener
{
    void news( NewsEvent event );
    void news( String string );
}
```

Class examples\space\Consumer.java

```
// copyright 1997-1999 objectspace
```

```
package examples.space;
```

```
public class Consumer implements IConsumer
{
    String name;

    public Consumer( String name )
    {
        this.name = name;
        System.out.println( "created " + this );
    }

    public String toString()
    {
```

```

    return "Consumer( " + name + " )";
}

public void news( NewsEvent event )
{
    System.out.println( this + " gets news " + event );
}

public void news( String string )
{
    System.out.println( this + " gets news " + string );
}
}

```

Program examples\space\Space1.java

// copyright 1997-1999 objectspace

package examples.space;

import com.objectspace.voyager.*;
import com.objectspace.voyager.space.*;

```

public class Space1
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();

            ISubspace subspace1 = (ISubspace) Factory.create( "com.objectspace.voyager.space.Subspace",
"//localhost:8000/Subspace1" );
            IConsumer consumer1 = (IConsumer) Factory.create( "examples.space.Consumer", new Object[]{
"jack" }, "//localhost:8000/Jack" );
            IConsumer consumer2 = (IConsumer) Factory.create( "examples.space.Consumer", new Object[]{
"sasha" }, "//localhost:8000/Sasha" );
            subspace1.add( consumer1 );
            subspace1.add( consumer2 );

            ISubspace subspace2 = (ISubspace) Factory.create( "com.objectspace.voyager.space.Subspace",
"//localhost:9000/Subspace2" );

```

```

        IConsumer consumer3 = (IConsumer) Factory.create( "examples.space.Consumer", new Object[]{
"simon" }, "localhost:9000/Simon" );
        IConsumer consumer4 = (IConsumer) Factory.create( "examples.space.Consumer", new Object[]{
"galina" }, "localhost:9000/Galina" );
        subspace2.add( consumer3 );
        subspace2.add( consumer4 );

        subspace1.connect( subspace2 );
    }
    catch( Exception exception )
    {
        System.err.println( exception );
    }

    Voyager.shutdown();
}
}

```

Space2 Example

The Space2 example demonstrates two forms of distributed multicast. The first form demonstrates multicast of standard Java method invocations. The second form demonstrates distributed JavaBeans-style event multicasting. Both multicasts are sent into the space constructed in Space1 by using one of the subspaces as a gateway.

Compile all .java files in examples\space:

```
javac *.java
```

Start a Voyager server on port 8000 in one window and a Voyager server on port 9000 in a second window. Run Space1 in a third window. Leave the two Voyager servers running. Then run Space2 in the third window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
created Consumer( jack )  
created Consumer( sasha )  
Consumer( jack ) gets news newsflash 2!  
Consumer( sasha ) gets news newsflash 2!  
Consumer( jack ) gets news newsflash 1!  
Consumer( sasha ) gets news newsflash 1!  
Consumer( jack ) gets news NewsEvent( newsflash 3! )  
Consumer( sasha ) gets news NewsEvent( newsflash 3! )
```

Window2

```
>voyager 9000  
voyager 3.1, copyright objectspace 1997-1999  
created Consumer( simon )  
created Consumer( galina )  
Consumer( simon ) gets news newsflash 2!  
Consumer( simon ) gets news newsflash 1!  
Consumer( galina ) gets news newsflash 1!  
Consumer( galina ) gets news newsflash 2!  
Consumer( simon ) gets news NewsEvent( newsflash 3! )  
Consumer( galina ) gets news NewsEvent( newsflash 3! )
```

Window3

```
>java examples.space.Space1  
  
>java examples.space.Space2  
  
>
```


The source code for NewsEvent.java, NewsListener.java, Reporter.java, and Space2.java follows:

Class examples\space\NewsEvent.java

```
// copyright 1997-1999 objectspace

package examples.space;

import java.util.EventObject;

public class NewsEvent extends EventObject
{
    String news;

    public NewsEvent( String news )
    {
        super( news );
        this.news = news;
    }

    public NewsEvent( Object source, String news )
    {
        super( source );
        this.news = news;
    }

    public String toString()
    {
        return "NewsEvent( " + news + " )";
    }
}
```

Interface examples\space\NewsListener.java

```
// copyright 1997-1999 objectspace

package examples.space;

import java.util.EventListener;

public interface NewsListener extends EventListener
{
}
```

```
void news( NewsEvent event );  
}
```

Class examples\space\Reporter.java

```
// copyright 1997-1999 objectspace  
  
package examples.space;  
  
import java.util.Vector;  
  
public class Reporter  
{  
    Vector listeners = new Vector();  
  
    synchronized public void addNewsListener( NewsListener listener )  
    {  
        listeners.addElement( listener );  
    }  
  
    synchronized public void news( String message )  
    {  
        NewsEvent event = new NewsEvent( this, message );  
  
        for( int i = 0; i < listeners.size(); i++ )  
            ((NewsListener) listeners.elementAt( i )).news( event );  
    }  
}
```

Program examples\space\Space2.java

```
// copyright 1997-1999 objectspace  
  
package examples.space;  
  
import com.objectspace.voyager.*;  
import com.objectspace.voyager.space.*;  
import com.objectspace.voyager.space.multicasting.*;  
  
public class Space2  
{  
    public static void main( String[] args )
```

```

{
try
{
    Voyager.startup();
    ISubspace subspace1 = (ISubspace) Namespace.lookup( "//localhost:8000/Subspace1" );
    IConsumer consumer1 = (IConsumer) subspace1.getMulticastProxy( "examples.space.IConsumer" );
    consumer1.news( "newsflash 1!" );
    Multicast.invoke( subspace1, "news", new Object[] { "newsflash 2!" }, "examples.space.IConsumer" );
    Reporter reporter = new Reporter();
    NewsListener listener = (NewsListener) subspace1.getMulticastProxy( "examples.space.NewsListener"
);
    reporter.addNewsListener( listener );
    reporter.news( "newsflash 3!" );
    try{ Thread.sleep( 2000 ); } catch( Exception exception ) {} // allow oneway messages to drain
    }
catch( Exception exception )
{
    exception.printStackTrace();
    System.err.println( exception );
}

    Voyager.shutdown();
}
}

```

Space3 Example

The Space3 example demonstrates Voyager's publish/subscribe mechanism. It uses the space that is built by the Space1 example. The program creates a subscriber for three of the consumers created in Space1 and subscribes each to a given topic. The subscriber for a given consumer is added to the subspace local to the consumer. It can therefore receive messages that are published to that subspace. The program uses a ConsumerAdapter to allow filtering of published events without coupling the Consumer class to the details of the publish/subscribe mechanism. Next, the program publishes messages to each of the three topics.

Compile all .java files in examples\space:

```
javac *.java
```

Start a Voyager server on port 8000 in one window and a Voyager server on port 9000 in a second window. Run Space1 in a third window. Leave the two Voyager servers running. Then run Space3 in the third window.

Window1

```
>voyager 8000
voyager 3.1, copyright objectspace 1997-1999
created Consumer( jack )
created Consumer( sasha )
adapter gets NewsEvent( news flash 4! ) on news.general
Consumer( jack ) gets news NewsEvent( news flash 4! )
adapter gets NewsEvent( news flash 5! ) on news.special
Consumer( jack ) gets news NewsEvent( news flash 5! )
adapter gets NewsEvent( news flash 5! ) on news.special
Consumer( sasha ) gets news NewsEvent( news flash 5! )
adapter gets NewsEvent( news flash 6! ) on news.*
Consumer( jack ) gets news NewsEvent( news flash 6! )
adapter gets NewsEvent( news flash 6! ) on news.*
Consumer( sasha ) gets news NewsEvent( news flash 6! )
```

Window2

```
>voyager 9000
voyager 3.1, copyright objectspace 1997-1999
created Consumer( simon )
created Consumer( galina )
adapter gets NewsEvent( news flash 4! ) on news.general
Consumer( galina ) gets news NewsEvent( news flash 4! )
adapter gets NewsEvent( news flash 6! ) on news.*
Consumer( galina ) gets news NewsEvent( news flash 6! )
```

Window3

```
>java examples.space.Space1  
  
>java examples.space.Space3  
  
>
```

The source code for ConsumerAdapter.java and Space3.java follows:

Class examples\space\ConsumerAdapter.java

```
// copyright 1997-1999 objectspace
```

```
package examples.space;
```

```
import java.io.Serializable;  
import java.util.EventObject;  
import com.objectspace.voyager.space.publishing.*;
```

```
public class ConsumerAdapter implements PublishedEventListener, Serializable  
{  
    IConsumer consumer;  
  
    public ConsumerAdapter( IConsumer consumer )  
    {  
        this.consumer = consumer;  
    }  
  
    public void publishedEvent( EventObject event, Topic topic )  
    {  
        System.out.println( "adapter gets " + event + " on " + topic );  
  
        if( event instanceof NewsEvent )  
            consumer.news( (NewsEvent) event ); // forward to consumer  
    }  
}
```

Program examples\space\Space3.java

```
// copyright 1997-1999 objectspace
```

```

package examples.space;

import com.objectspace.voyager.*;
import com.objectspace.voyager.space.*;
import com.objectspace.voyager.space.publishing.*;

public class Space3
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            final String subscriberclass = "com.objectspace.voyager.space.publishing.Subscriber";
            ISubspace subspace1 = (ISubspace) Namespace.lookup( "//localhost:8000/Subspace1" );
            IConsumer jack = (IConsumer) Namespace.lookup( "//localhost:8000/Jack" );
            ISubscriber subscriber1 = (ISubscriber) Factory.create( subscriberclass, "//localhost:8000" );
            subscriber1.subscribe( new Topic( "news.*" ) );
            subscriber1.setListener( new ConsumerAdapter( jack ) );
            subspace1.add( subscriber1 );

            IConsumer sasha = (IConsumer) Namespace.lookup( "//localhost:8000/Sasha" );
            ISubscriber subscriber2 = (ISubscriber) Factory.create( subscriberclass, "//localhost:8000" );
            subscriber2.subscribe( new Topic( "news.special" ) );
            subscriber2.setListener( new ConsumerAdapter( sasha ) );
            subspace1.add( subscriber2 );

            ISubspace subspace2 = (ISubspace) Namespace.lookup( "//localhost:9000/Subspace2" );
            IConsumer galina = (IConsumer) Namespace.lookup( "//localhost:9000/Galina" );
            ISubscriber subscriber3 = (ISubscriber) Factory.create( subscriberclass, "//localhost:9000" );
            subscriber3.subscribe( new Topic( "news.general" ) );
            subscriber3.setListener( new ConsumerAdapter( galina ) );
            subspace2.add( subscriber3 );

            Publish.invoke( subspace1, new NewsEvent( "news flash 4!" ), new Topic( "news.general" ) );
            Publish.invoke( subspace1, new NewsEvent( "news flash 5!" ), new Topic( "news.special" ) );
            Publish.invoke( subspace1, new NewsEvent( "news flash 6!" ), new Topic( "news.*" ) );

            try{ Thread.sleep( 2000 ); } catch( Exception exception ) {} // allow oneway messages to drain
        }
        catch( Exception exception )

```

```
{  
    System.err.println( exception );  
}  
  
Voyager.shutdown();  
}  
}
```

Mobility

The examples in this section demonstrate how any serializable object can be moved around the network.

Mobility1 Example

The Mobility1 example demonstrates mobility and messaging. An object is constructed, moved, and sent messages. The movement of the object is transparent to the client. The client's reference to the moving object is valid whether the object is local, remote, or in the process of moving.

From examples\mobility, compile the example program:

```
javac IDrone.java Drone1.java Mobility1.java
```

Start a Voyager server on port 8000 in one window. Run Mobility1 in a second window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
drone prints 1  
drone prints 3
```

Window2

```
>java examples.mobility.Mobility1  
drone prints 0  
drone prints 2  
>
```

The source code for IDrone.java, Drone1.java, and Mobility1.java follows:

Interface examples\mobility\IDrone.java

```
// copyright 1997-1999 objectspace
```

```
package examples.mobility;
```

```
public interface IDrone
{
    void print( int i );
}
```

Class examples\mobility\Drone1.java

```
// copyright 1997-1999 objectspace
```

```
package examples.mobility;
```

```
import java.io.*;
```

```
public class Drone1 implements IDrone, Serializable
{
    public void print( int i )
    {
        System.out.println( "drone prints " + i );
    }
}
```

Program examples\mobility\Mobility1.java

```
// copyright 1997-1999 objectspace
```

```
package examples.mobility;
```

```
import com.objectspace.voyager.*;
import com.objectspace.voyager.mobility.*;
import com.objectspace.lib.util.*;
```

```
public class Mobility1
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup( "7000" );
        }
    }
}
```

```

// create a local drone
IDrone drone = (IDrone) Factory.create( "examples.mobility.Drone1" );
// get/add mobility facet
IMobility mobility = Mobility.of( drone );

// move the drone to and from the remote program
for( int i = 0; i < 4; i++ )
{
    drone.print( i ); // display integer at current location

    if( i % 2 == 0 )
        mobility.moveTo( "//localhost:8000" ); // move remote
    else
        mobility.moveTo( "//localhost:7000" ); // move local
    }
}
catch( Exception exception )
{
    System.err.println( exception );
}

Voyager.shutdown();
}
}

```

Mobility2 Example

The Mobility2 example demonstrates receiving mobility callbacks using the IMobile interface. An object is constructed and moved around. The various move callbacks are executed on the object throughout the operation.

From examples\mobility, compile the example program:

```
javac Drone2.java Mobility2.java
```

Start a Voyager server on port 8000 in one window. Run Mobility2 in a second window.

Window1

```
>voyager 8000
voyager 3.1, copyright objectspace 1997-1999
0 preArrival()
0 postArrival()
drone prints 1
1 preDeparture( tcp://homer1:8000, //localhost:7000 )
1 postDeparture()
2 preArrival()
2 postArrival()
drone prints 3
3 preDeparture( tcp://homer1:8000, //localhost:7000 )
3 postDeparture()
```

Window2

```
>java examples.mobility.Mobility2
drone prints 0
0 preDeparture( tcp://homer1:7000, //localhost:8000 )
0 postDeparture()
1 preArrival()
1 postArrival()
drone prints 2
2 preDeparture( tcp://homer1:7000, //localhost:8000 )
2 postDeparture()
3 preArrival()
3 postArrival()

>
```

The source code for Drone2.java and Mobility2.java follows:

Class examples\mobility\Drone2.java

```
// copyright 1997-1999 objectspace

package examples.mobility;

import java.io.*;
```

```

import com.objectspace.voyager.mobility.*;

public class Drone2 implements IDrone, IMobile, Serializable
{
    int n;

    public void print( int i )
    {
        n = i;
        System.out.println( "drone prints " + i );
    }

    public void preDeparture( String source, String destination )
    {
        System.out.println( n + " preDeparture( " + source + ", " + destination + " )" );
    }

    public void preArrival()
    {
        System.out.println( n + " preArrival()" );
    }

    public void postArrival()
    {
        System.out.println( n + " postArrival()" );
    }

    public void postDeparture()
    {
        System.out.println( n + " postDeparture()" );
    }
}

```

Program examples\mobility\Mobility2.java

```

// copyright 1997-1999 objectspace

package examples.mobility;

import com.objectspace.voyager.*;
import com.objectspace.voyager.mobility.*;

public class Mobility2

```

```

{
public static void main( String[] args )
{
try
{
Voyager.startup( "7000" );
// create a local drone
IDrone drone = (IDrone) Factory.create( "examples.mobility.Drone2" );
// get/add mobility facet
IMobility mobility = Mobility.of( drone );

// move the drone to and from the remote program
for( int i = 0; i < 4; i++ )
{
drone.print( i ); // display integer at current location

if( i % 2 == 0 )
mobility.moveTo( "//localhost:8000" ); // move remote
else
mobility.moveTo( "//localhost:7000" ); // move local
}
}
catch( Exception exception )
{
System.err.println( exception );
}

Voyager.shutdown();
}
}

```

Agents

The example in this section demonstrates how any serializable object can use Voyager's dynamic aggregation feature to become a mobile, autonomous agent.

Agents1 Example

The Agents1 example demonstrates mobile autonomous agents. An object uses dynamic aggregation to access its Agent facet. This allows the object to move itself around the network. When the agent has completed its tasks, it disables its autonomy, allowing the agent to be garbage collected.

Note that messaging speeds are greatly improved after the agent co-locates itself with its target.

If the Stockmarket example classes from the Basics1 example have not already been compiled, do so now. From `examples\agents`, compile the example program:

```
javac ITrader.java Trader.java Agents1.java
```

Start a Voyager server on port 8000 in one window. Run Agents1 in a second window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999  
construct stockmarket  
at remote market  
start trade  
stop trade  
time = 380ms
```

Window2

```
>java examples.agents.Agents1
construct trader
remote trade
start trade
stop trade
time = 4917ms
local trade

>
```

The source code for ITrader.java, Trader.java, and Agents1.java follows:

Interface examples\agents\ITrader.java

```
// copyright 1997-1999 objectspace
```

```
package examples.agents;
```

```
import examples.stockmarket.*;
```

```
public interface ITrader
```

```
{
    void work( IStockmarket market );
}
```

Class examples\agents\Trader.java

```
// copyright 1997-1999 objectspace
```

```
package examples.agents;
```

```
import java.io.*;
```

```
import com.objectspace.lib.timer.*;
```

```
import com.objectspace.voyager.*;
```

```
import com.objectspace.voyager.agent.*;
```

```
import examples.stockmarket.*;
```

```
public class Trader implements ITrader, Serializable
{
```

```

public Trader()
{
    System.out.println( "construct trader" );
}

public void finalize()
{
    System.out.println( "finalize trader" );
}

public void work( IStockmarket market )
{
    System.out.println( "remote trade" );
    tradeAt( market ); // trade with remote market
    System.out.println( "local trade" );

    try
    {
        Agent.of( this ).moveTo( market, "atMarket" ); // move to market
    }
    catch( Exception exception )
    {
        System.err.println( exception );
    }
}

public void atMarket( IStockmarket market )
{
    System.out.println( "at remote market, home = " + Agent.of( this ).getHome() );
    tradeAt( market ); // trade with local market
    Agent.of( this ).setAutonomous( false ); // allow myself to be gc'ed
}

private void tradeAt( IStockmarket market )
{
    System.out.println( "start trade" );
    Stopwatch watch = new Stopwatch();
    watch.start();

    for( int i = 0; i < 1000; i++ ) // do 1000 trades
        market.buy( 100, "SUN" );
}

```



```

        watch.stop();
        System.out.println( "stop trade" );
        System.out.println( "time = " + watch.getTotalTime() + "ms" );
    }
}

```

Program examples\agents\Agents1.java

// copyright 1997-1999 objectspace

```
package examples.agents;
```

```
import examples.stockmarket.*;
import com.objectspace.voyager.*;
```

```

public class Agents1
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            // create remote stockmarket
            String marketClass = Stockmarket.class.getName();
            IStockmarket market = (IStockmarket) Factory.create( marketClass, "localhost:8000" );
            // create trader agent
            ITrader trader = (ITrader) Factory.create( Trader.class.getName() );
            // trade from local machine, then trade on remote machine
            trader.work( market );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }

        Voyager.shutdown();
    }
}

```

Naming Service

The examples in this section demonstrate Voyager's federated directory system and pluggable naming service.

Naming1 Example

The Naming1 example demonstrates the federated directory system. It first creates a directory in a remote Voyager server into which it places a few items. It then creates another directory on a different remote Voyager server and binds the first into the second. Next, it demonstrates traversal of names across the federated directory service.

From examples\naming, compile the example program:

```
javac Naming1.java
```

Start a Voyager server on port 8000 in one window and a Voyager server on port 9000 in a second window. Run Naming1 in a third window.

Window1

```
>voyager 8000  
voyager 3.1, copyright objectspace 1997-1999
```

Window2

```
>voyager 9000  
voyager 3.1, copyright objectspace 1997-1999
```

Window3

```
>java examples.naming.Naming1  
CA -> calcium  
AU -> gold  
AU -> null  
  
>
```

The source code for Naming1.java follows:

Program examples\naming\Naming1.java

```
// copyright 1997-1999 objectspace
```

```
package examples.naming;
```

```
import com.objectspace.voyager.*;  
import com.objectspace.voyager.directory.*;
```

```
public class Naming1
```

```
{  
    public static void main( String[] args )  
    {  
        try  
        {  
            Voyager.startup();
```

```
            // create and populate a remote directory of chemical symbols  
            String dirClass = Directory.class.getName();  
            IDirectory symbols = (IDirectory) Factory.create( dirClass, "localhost:8000" );  
            symbols.put( "CA", "calcium" );  
            symbols.put( "AU", "gold" );
```

```
            // link a root directory in a different program to the symbols  
            IDirectory root = (IDirectory) Factory.create( dirClass, "localhost:9000" );  
            root.put( "symbols", symbols );
```

```
            // access the symbols from the root directory  
            System.out.println( "CA -> " + root.get( "symbols/CA" ) );  
            System.out.println( "AU -> " + root.get( "symbols/AU" ) );  
            root.remove( "symbols/AU" );
```

```

        System.out.println( "AU -> " + root.get( "symbols/AU" ) );
    }
    catch( Exception exception )
    {
        System.err.println( exception );
    }

    Voyager.shutdown();
}
}

```

Naming2 Example

The Naming2 example demonstrates Voyager's naming service. It first creates an object in a remote Voyager server. It then binds that object to a name in that server's name space. After the object is bound to a name, the program is able to look up that object by name. The program then unbinds the object and demonstrates that lookup with the old name will no longer succeed.

Compile the Stockmarket example classes from the Basics1 example. From examples\naming, compile the example program:

```
javac Naming2.java
```

Start a Voyager server on port 8000 in one window. Run Naming2 in a second window.

Window1

```

>voyager 8000
voyager 3.1, copyright objectspace 1997-1999
construct stockmarket

```

Window2

```
>java examples.naming.Naming2
//localhost:8000/NASDAQ -> examples.stockmarket.Stockmarket@1f3587
com.objectspace.voyager.NamespaceException: no object bound to the name
vdir://localhost:8000/NASDAQ;
```

The source code for Naming2.java follows:

Program examples\naming\Naming2.java

```
// copyright 1997-1999 objectspace
```

```
package examples.naming;
```

```
import examples.stockmarket.*;
import com.objectspace.voyager.*;
```

```
public class Naming2
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            // create a remote stockmarket
            String marketClass = Stockmarket.class.getName();
            String marketName = "//localhost:8000/NASDAQ";
            IStockmarket market1 = (IStockmarket) Factory.create( marketClass, "//localhost:8000" );
            // bind the stockmarket to the symbol "NASDAQ"
            Namespace.bind( marketName, market1 );
            // lookup the stockmarket
            IStockmarket market2 = (IStockmarket) Namespace.lookup( marketName );
            System.out.println( marketName + " -> " + market2 );
            // unbind the symbol
            Namespace.unbind( marketName );
            // lookup the stockmarket again
            IStockmarket market3 = (IStockmarket) Namespace.lookup( marketName );
            System.out.println( marketName + " -> " + market3 );
        }
        catch( Exception exception )
```

```

    {
        System.err.println( exception );
    }

    Voyager.shutdown();
}
}

```

Naming3 Example

The Naming3 example demonstrates the use of JNDI to access objects in a Voyager server. First, an initial context is created with the service provider URL `"//localhost:8000/state"`. After it populates the context by binding names to string objects, the program is able to look up a string object by name. Then it lists the contents of the context which will be the name and string objects added to the context. The program then overwrites the binding with name `"CA"` using the `rebind` method and remove binding with name `"IL"` using the `unbind` method. The updated contents of the context is listed again to show the effect of the `rebind` and `unbind`. The program then creates a subcontext under the initial context and populates the subcontext. After showing the bindings of the subcontext, the subcontext is destroyed.

From `Voyager\examples\naming`, compile the example program:

```
Javac Naming3.java
```

Start a Voyager server on port 8000 in one window, specify the root and storage file for the directory server, and enable resource serving. Run Naming3 in a second window.

Window 1

```

C:\>voyager 8000 -r -j state -f c:\store
voyager orb professional (directory server) 3.1, copyright objectspace 1997-1999

```

Window 2

```
C:\>java examples.naming.Naming3
TX -> Texas
context listing:
IL: java.lang.String:Illinois
TX: java.lang.String:Texas
CA: java.lang.String:California
ON: java.lang.String:Ontario
updated context listing:
TX: java.lang.String:Texas
CA: java.lang.String:New California
ON: java.lang.String:Ontario
subcontext bindings:
DAL: Dallas
TOR: Toronto
```

The source code for Naming3.java follows:

Program examples\naming\Naming3.java

```
// copyright 1997-1999 objectspace

package examples.naming;

import javax.naming.*;
import java.util.Properties;

public class Naming3
{
    public static void main( String[] args )
    {
        try
        {
            // create initial context
            Properties env = new Properties();
            env.put( "java.naming.factory.initial",
                    "com.objectspace.voyager.jndi.spi.VoyagerContextFactory" );
            env.put( "java.naming.provider.url", "///localhost:8000/state" );
            Context ctx = new InitialContext( env );
```

```

// populate the context
ctx.bind( "TX", "Texas" );
ctx.bind( "CA", "California" );
ctx.bind( "IL", "Illinois" );
ctx.bind( "ON", "Ontario" );

// lookup the string object "Texas"
String tx = (String)ctx.lookup( "TX" );
System.out.println( "TX -> " + tx );

// list the contents of the context
System.out.println( "context listing:" );
NamingEnumeration list = ctx.list( "" );

while( list.hasMore() )
{
    NameClassPair nc = (NameClassPair)list.next();
    System.out.println( nc );
}

// overwrite "CA" binding
ctx.rebind( "CA", "New California" );

// remove "IL" binding
ctx.unbind( "IL" );

// list the contents of the updated context
System.out.println( "updated context listing:" );
NamingEnumeration updatedList = ctx.list( "" );

while( updatedList.hasMore() )
{
    NameClassPair nc = (NameClassPair)updatedList.next();
    System.out.println( nc );
}

// create and populate subcontext "city"
Context cityctx = ctx.createSubcontext( "city" );
cityctx.bind( "DAL", "Dallas" );
cityctx.bind( "TOR", "Toronto" );

```



```

// list the bindings in the "city" context
System.out.println( "subcontext bindings:" );
NamingEnumeration bindinglist = ctx.listBindings( "city" );

while( bindinglist.hasMore() )
{
    Binding bd = (Binding)bindinglist.next();
    System.out.println( bd.getName() + ": " + bd.getObject() );
}

// destroy subcontext "city"
ctx.destroySubcontext( "city" );
}
catch( Exception exception )
{
    exception.printStackTrace();
}
}
}

```

Activation

The examples in this section explain Voyager's activation framework. they also demonstrate a Library service capable of looking up activatable Book objects.

This example provides a file-based activator that uses serialization. you can replace this persistent store with custom implementations. Refer to the Voyager website for implementations that use PSE and JBDC that you can plug in this example to demonstrate how to adapt the activation framework to any persistent store.

Activation1A and Activation1B Example

The Activation1A and Activation 1B examples demonstrate how activating proxies are able to fault in their associated remote objects. First, the library server is started with a FileLibrary. Then, an activatable proxy is obtained to a remote book object. The server is then restarted, which removes the book object from memory. Then the client messages the book, which activates the book back into memory on the server. These examples illustrate the server restart. However, activation can also occur if the client persists the proxy, or its external form, and the distributed garbage collector removes the book from memory. In this case, if the client's reference is restored and messaged, the book will also be activated on the server.

Compile all .java files in examples\activation:

```
javac *.java
```

Run the library service, which is started from Activation1A. The extra flag instructs the program to populate the database. After the Activation1A is running, run Activation1B in a second window from the examples\activation directory. Cycle Activation1A as instructed, but do not specify the startup flag when restarting.

Window1

```
>java examples.activation.Activation1A examples.activation.FileLibrary startup  
library service online  
press enter to kill the library service  
get memento of Book( Two Heads are Better Than One )  
  
>java examples.activation.Activation1A examples.activation.FileLibrary  
library service online  
press enter to kill the library service  
activating 0-201-63452-X from File Library
```

Window2

```
>java examples.activation.Activation1B  
book = Book( Two Heads are Better Than One ), ISBN = 0-201-63452-X  
enabling book for activation...  
sleeping for 20 seconds. please cycle Activation1A  
display the book again  
book = Book( Two Heads are Better Than One ), ISBN = 0-201-63452-X  
  
>
```

The source code for IBook.java, Book.java, ILibrary.java, FileLibrary.java, BookActivator.java, Activation1A.java, and Activation1B.java follows.

Interface examples\activation\IBook.java

```
// copyright 1997-1999 objectspace
```

```
package examples.activation;
```

```
public interface IBook  
{  
    String getISBN();  
    String getTitle();  
    String getAuthor();  
}
```

Class examples\activation\Book.java

```
// copyright 1997-1999 objectspace

package examples.activation;

import java.io.Serializable;
import com.objectspace.voyager.IRemote;

/**
 * Book object that has a title, an author, and an ISBN.
 */

public class Book implements Serializable, IRemote, IBook
{
    String title;
    String author;
    String isbn;

    public Book( String title, String author, String isbn )
    {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
    }

    public String getISBN()
    {
        return isbn;
    }

    public String getTitle()
    {
        return title;
    }

    public String getAuthor()
    {
        return author;
    }

    public int hashCode()
```

```

    {
        return isbn.hashCode();
    }

    public boolean equals( Object object )
    {
        return ( object.getClass() == getClass() ) ? ((Book) object).isbn.equals( isbn ) : false;
    }

    public String toString()
    {
        return "Book( " + title + " )";
    }
}

```

Interface examples\activation\ILibrary.java

```

// copyright 1997-1999 objectspace

package examples.activation;

import java.io.IOException;

public interface ILibrary
{
    void addBook( IBook book ) throws IOException;
    IBook getBook( String isbn ) throws IOException;
    void close();
}

```

Class examples\activation\FileLibrary.java

```

// copyright 1997-1999 objectspace

package examples.activation;

import java.io.*;
import java.util.*;

/**
 * Simple Library implementation that serializes the book to a
 * file named after the ISBN of the book.
 */

```

```

public class FileLibrary implements ILibrary, com.objectspace.voyager.IRemote
{
    Hashtable cache = new Hashtable();

    public FileLibrary()
    {
    }

    public void addBook( IBook book ) throws IOException
    {
        String filename = book.getISBN() + ".dat";
        ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream( filename ) );

        try
        {
            out.writeObject( book );
        }
        finally
        {
            out.close();
        }
    }

    /**
     * Note that since Book implements IRemote, this method will return the
     * book "by reference". That is, a proxy to the book is returned. Therefore,
     * this book object remains on the server, and the client actually receives a proxy.
     */
    public IBook getBook( String isbn ) throws IOException
    {
        IBook book = (IBook) cache.get( isbn );

        if( book != null )
            return book;

        ObjectInputStream in = new ObjectInputStream( new FileInputStream( isbn + ".dat" ) );

        try
        {
            book = (IBook) in.readObject();
            cache.put( isbn, book );
        }
    }
}

```

```

        return book;
    }
    catch( ClassNotFoundException exception )
    {
        throw new IOException( exception.toString() );
    }
    finally
    {
        in.close();
    }
}

public void close()
{
    // do nothing
}

public String toString()
{
    return "File Library";
}
}

```

Class examples\activation\BookActivator.java

// copyright 1997-1999 objectspace

```
package examples.activation;
```

```
import java.io.IOException;
import java.util.Hashtable;
import com.objectspace.voyager.*;
import com.objectspace.voyager.activation.*;
```

```
/**
 * Simple activator capable of activating instances of Book. Note that this activator
 * is in no way coupled to the persistence mechanism. Any persistence mechanism can be
 * plugged in by specifying a custom implementation of ILibrary.
 */
```

```
public class BookActivator implements IActivator
{
    ILibrary library;
```

```

BookActivator( ILibrary library )
{
    this.library = library;
}

/**
 * If this method returns null, then the activation manager will ask the remaining
 * Activators registered with it (if any) for the memento. If this method
 * returns a string value, then the activation manager will assume this Activator
 * is handling the object to memento conversion and the string will be used
 * in the activating proxy.
 *
 * This method can use any heuristic to determine whether or not to handle
 * the object to memento conversion. This simple example merely shows one
 * such heuristic: handle the conversion if the object is an instance of Book
 * that is in the library.
 */
public String getMemento( Proxy proxy ) throws ActivationException
{
    try
    {
        System.out.println( "get memento of " + proxy );
        Object object = Snapshot.of( proxy ).getObject();
        return object instanceof Book ? ((IBook) object).getISBN() : null;
    }
    catch( Exception exception )
    {
        throw new ActivationException( exception );
    }
}

/**
 * This example uses no aggregation or custom exportation and therefore ignores the
 * facets and properties of the Snapshot object. More sophisticated
 * applications may need to persist all of the Snapshot's data. This data would be
 * used in this method to fully reconstruct the Snapshot.
 */
public Proxy activate( String memento ) throws ActivationException
{
    try
    {

```



```

        System.out.println( "activating " + memento + " from " + library );
        IBook book = library.getBook( memento );
        return Snapshot.from( book, null, null ).restore();
    }
    catch( Exception exception )
    {
        throw new ActivationException( exception );
    }
}
}
}

```

Program examples\activation\Activation1A.java

// copyright 1997-1999 objectspace

```
package examples.activation;
```

```
import java.io.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.activation.*;
```

```
public class Activation1A
```

```

{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup( "8000" );
            ILibrary library = (ILibrary) Class.forName( args[ 0 ] ).newInstance();

            if( args.length > 1 ) // stock the library
            {
                IBook book1 = new Book( "Know Where Your Towel Is", "Ford Prefect", "0-201-63451-1" );
                library.addBook( book1 );
                IBook book2 = new Book( "Two Heads are Better Than One", "Zaphod Beeblebrox", "0-201-63452-X" );
                library.addBook( book2 );
            }

```

```

            BookActivator activator = new BookActivator( library );
            Activation.register( activator );
            Namespace.bind( "Library", library );
            System.out.println( "library service online" );
            System.out.println( "press enter to kill the library server" );

```

```

        System.in.read(); // block for <Enter> key
        library.close(); // allow the library to do any cleanup necessary
        Voyager.shutdown();
    }
    catch( Exception exception )
    {
        exception.printStackTrace();
    }
}
}

```

Program examples\activation\Activation1B.java

// copyright 1997-1999 objectspace

```
package examples.activation;
```

```

import java.io.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.activation.*;

public class Activation1B
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            ILibrary library = (ILibrary) Namespace.lookup( "//localhost:8000/Library" );
            IBook book = library.getBook( "0-201-63452-X" );
            System.out.println( "book = " + book + ", ISBN = " + book.getISBN() );
            System.out.println( "enabling book for activation..." );
            Activation.enable( book );
            System.out.println( "sleeping for 20 seconds. please cycle Activation1A" );
            try{ Thread.sleep( 20000 ); } catch( Exception exception ) {}
            System.out.println( "display the book again" );
            System.out.println( "book = " + book + ", ISBN = " + book.getISBN() );
        }
        catch( Exception exception )
        {
            exception.printStackTrace();
        }
    }
}

```

```
Voyager.shutdown();  
}  
}
```

Security

The example in this section demonstrates how Voyager's security mechanism can restrict the actions of objects whose classes have been loaded from a remote source.

Security1 Example

The Security1 program sends a Visitor object to two remote locations: the server started in the Native program and the server started in the Foreign. The Native program is in the same directory as the Security1 program. Because '.' is in the CLASSPATH, the classloader in the Native program can load the Visitor class without using a resource loader, and any Visitor objects will be trusted. However, because the Foreign program is in a different directory, it must use a resource loader to load the Visitor class and will therefore not trust instances of Visitor. Though both remote programs have security managers installed, only the Foreign program will disallow the Visitor's thread manipulations.

From examples\security\native, compile the example program:

```
javac IVisitor.java Visitor.java Native.java Security1.java
```

From examples\security\foreign, compile the example program:

```
javac Foreign.java
```

From examples\security\foreign, run the Foreign program. Run the Native program in a second window from examples\security\native. Finally, run Security1 in a third window from examples\security\native.

Window1

```
>java Foreign  
creating thread
```

Window2

```
>java Native  
creating thread  
thread created. starting  
thread started  
new thread is counting...  
0  
1  
2  
3
```

Window3

```
>java Security1  
java.lang.SecurityException: foreign objects/messages may not manipulate a thread group
```

The source code for IVisitor.java, Visitor.java, Library.java, Foreign.java, Native.java, and Security1.java follows:

Interface examples\security\native\IVisitor.java

// Copyright (c) 1997, 1998 ObjectSpace, Inc.

```
import java.io.Serializable;  
  
public interface IVisitor  
{  
    void loopUsingThread();  
}
```

Program examples\security\native\Visitor.java

// Copyright (c) 1997, 1998 ObjectSpace, Inc.

```
import java.io.Serializable;  
import com.objectspace.voyager.*;
```

```

public class Visitor implements IVisitor, Serializable, Runnable
{
    public synchronized void loopUsingThread()
    {
        System.out.println( "creating thread" );
        Thread thread = new Thread( this );
        System.out.println( "thread created. starting" );
        thread.start();
        System.out.println( "thread started" );

        try
        {
            wait(); // wait for thread to complete
        }
        catch( InterruptedException exception )
        {
        }
    }

    public synchronized void run()
    {
        System.out.println( "new thread is counting..." );

        for( int i = 0; i < 4; i++ )
            System.out.println( i );

        notify(); // notify that thread has completed
    }
}

```

Program examples\security\foreign\Foreign.java

// Copyright (c) 1997, 1998 ObjectSpace, Inc.

```

import com.objectspace.voyager.*;
import com.objectspace.voyager.loader.*;
import com.objectspace.voyager.security.*;

public class Foreign
{
    public static void main( String args[] )
    {

```

```

try
{
    System.setSecurityManager( new VoyagerSecurityManager() );
    Voyager.startup( "7000" );
    // load class bytes from this server if not local
    VoyagerClassLoader.addURLResource( "http://localhost:9000/" );
}
catch( Exception exception )
{
    System.err.println( exception );
}
}
}

```

Program examples\security\native\Native.java

// Copyright (c) 1997, 1998 ObjectSpace, Inc.

```

import com.objectspace.voyager.*;
import com.objectspace.voyager.security.*;

public class Native
{
    public static void main( String args[] )
    {
        try
        {
            System.setSecurityManager( new VoyagerSecurityManager() );
            Voyager.startup( "8000" );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }
    }
}

```

Program examples\security\native\Security1.java

// Copyright (c) 1997, 1998 ObjectSpace, Inc.

```

import com.objectspace.voyager.*;

public class Security1

```

```

{
public static void main( String args[] )
{
    try
    {
        ClassManager.enableResourceServer(); // allow remote vm's to load class bytes from local classpath
        Voyager.startup( "9000" );

        IVisitor visitor1 = (IVisitor)Factory.create( "Visitor", "8000" ); // Visitor in remote vm classpath
        visitor1.loopUsingThread();

        IVisitor visitor2 = (IVisitor)Factory.create( "Visitor", "7000" ); // Visitor not in remote vm classpath
        visitor2.loopUsingThread();

        Voyager.shutdown();
    }
    catch( Exception exception )
    {
        System.err.println( exception );
    }
}
}

```

Applets and Servlets

The examples in this section demonstrate how Voyager works with applets and servlets.

Note: Because Netscape does not fully implement the Java 1.1 specification, if you are using Netscape you must either use the Java PlugIn or put the Java 1.1 classes.zip file from the JRE or JDK in Netscape's classpath.

CalcApplet Example

The CalcApplet example demonstrates an applet that uses Voyager. The CalcServer program merely exports a calculator object capable of adding two numbers. The applet receives a reference to this calculator object and uses it to add numbers entered by the user.

From examples\calculator, compile the Calculator utility classes program:

```
javac ICalculator.java Calculator.java CalcServer.java
```

From examples\applets, compile the example program:

```
javac CalcApplet.java
```

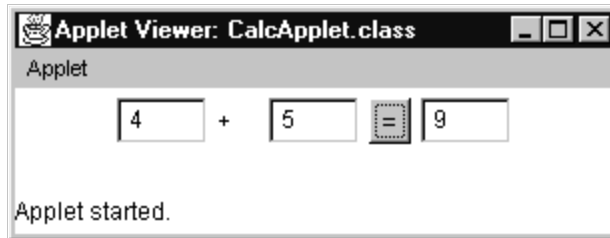
Run the CalcServer program in one window from examples\calculator. Run appletviewer with the HTML file in a second window from examples\applet.

Window1

```
>java -Dclass_based_proxies=true examples.calculator.CalcServer  
4 + 5 = 9
```

Window2

>appletviewer http://<Voyager Server:port>/Calculator.html



The source code for ICalculator.java, Calculator.java, CalcServer.java, CalcApplet.java and Calculator.html follows:

Interface examples\calculator \ICalculator.java

// copyright 1997-1999 objectspace

```
package examples.calculator;
```

```
public interface ICalculator
{
    int add( int x, int y );
}
```

Class examples\ calculator \Calculator.java

// copyright 1997-1999 objectspace

```
package examples.calculator;
```

```
public class Calculator implements ICalculator
{
    public int add( int x, int y )
    {
        int result = x + y;
        System.out.println( x + " + " + y + " = " + result );
    }
}
```

```

        return result;
    }
}

```

Program examples\calculator \CalcServer.java

// copyright 1997-1999 objectspace

```
package examples.calculator;
```

```
import com.objectspace.voyager.*;
```

```
public class CalcServer
```

```
{
    static ICalculator calculator;
```

```
    public static void main( String args[] )
```

```
    {
        try
        {
            // enables remote classloading from this program
            ClassManager.enableResourceServer();

```

```
            // starts Voyager on port 8000
            Voyager.startup( "8000" );

```

```
            // construct the calculator
            calculator = (ICalculator) Factory.create( "examples.calculator.Calculator" );
            Namespace.rebind( "MyCalculator", calculator );
        }

```

```
    catch( Exception exception )
    {
        System.err.println( exception );
    }
}
}

```

Applet examples\applets\CalcApplet.java

// copyright 1997-1999 objectspace

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

```

import com.objectspace.voyager.*;
import com.objectspace.voyager.router.*;
import examples.calculator.*;

public class CalcApplet extends Applet
{
    TextField operand1 = new TextField( "", 3 );
    Label plus = new Label( "+" );
    TextField operand2 = new TextField( "", 3 );
    Button add = new Button( "=" );
    TextField result = new TextField( "", 3 );
    ICalculator calculator;

    public void init()
    {
        setLayout( new FlowLayout() );
        add( operand1 );
        add( plus );
        add( operand2 );
        add( add );
        add( result );
        add.addActionListener( new ActionListener()
        { public void actionPerformed( ActionEvent event ) { add(); } } );
    }

    public void start()
    {
        try
        {
            // initialize voyager using the current security sandbox boundaries
            Voyager.startup( this, null );

            // connect to remote calculator in server
            calculator = (ICalculator) Namespace.lookup( Routing.getRouterAddress() + "/MyCalculator" );
        }
        catch( Exception exception )
        {
            System.err.println( exception );
        }
    }

    public void stop()

```

```

{
try
{
    Voyager.shutdown();
}
catch( Exception exception )
{
}
}

void add()
{
    int x = Integer.parseInt( operand1.getText() );
    int y = Integer.parseInt( operand2.getText() );

    try
    {
        result.setText( Integer.toString( calculator.add( x, y ) ) );
    }
    catch( Exception exception )
    {
        result.setText("Error");
    }
}
}

```

HTML examples\applets\Calculator.html

```

<html>
<head><title>Calculator Applet</title></head>
<body><hr><p>
<applet code="CalcApplet" align="baseline" width="300" height="50">
<param name="routerAddress" value="8000">
</applet>
</p><hr></body>
</html>

```

CalcServlet Example

The CalcServlet example demonstrates using servlets with Voyager. This example requires that install the Servlets SDK before compiling the example. You can download

the Servlets SDK from Sun's Java website <http://java.sun.com>. It also requires that you have a Servlet compatible webserver to run the example. Sun's Java website also has pointers to Servlet compatible webserver.

Compile the CalcServer program from the CalcApplet example now if not compiled. From examples\servlets, compile the example program:

```
javac CalcServlet.java
```

The source code for CalcServlet.java and Calculator.html follows:

HTML examples\servlets\Calculator.html

```
<html>
<head><title>Calculator Servlet</title></head>
<body><hr><p>
<form method="get" action="/Calculator" >
<input type="text" size=5 name="lhs"> + <input type="text" size=5 name="rhs"> <input type="submit" value=" = ">
</form>
</p><hr></body>
</html>
```

Servlet examples\servlets\CalcServlet.java

```
// copyright 1997-1999 objectspace
```

```
package examples.servlet;
```

```
import java.io.*;
import java.util.*;
```

```
import javax.servlet.*; // requires servlet developers kit from javasoft
import javax.servlet.http.*;
```

```
import com.objectspace.voyager.*;
```

```
import examples.calculator.*;
```

```
public class CalcServlet extends HttpServlet
{
```

```
    protected ICalculator calculator;
```

```
    public void init( ServletConfig config ) throws ServletException
```

```

{
    super.init( config );

    try
    {
        if( !Voyager.isStarted() )
            Voyager.startup( this, null ); // start voyager with current classloader
    }
    catch( Exception exception )
    {
        System.out.println( exception.toString() );
    }
}

public void doGet ( HttpServletRequest request, HttpServletResponse response )
    throws ServletException, IOException
{
    ServletOutputStream out = response.getOutputStream(); // output stream to write response
    String message; // outgoing message
    String queryString = request.getQueryString(); // the data to calculate

    if( queryString == null )
    {
        message = "<p><h2>no data</h2>";
    }
    else
    {
        Hashtable parameters = HttpUtils.parseQueryString( queryString );
        String[] lhs = (String[]) parameters.get( "lhs" );
        String[] rhs = (String[]) parameters.get( "rhs" );

        try
        {
            calculator = (ICalculator) Namespace.lookup( "//localhost:8000/MyCalculator" );
            int value = calculator.add( Integer.parseInt( lhs[0] ), Integer.parseInt( rhs[0] ) );
            message = "<P><hr><b>" + lhs[0] + "</b> + <b>" + rhs[0] + "</b> = <b><i>" + value + "</b></i><hr>";
        }
        catch( Exception exception )
        {
            System.out.println( exception.toString() );
            message = "<h2>an error occured: <i>" + exception + "</i></h2>";
        }
    }
}

```

```

    }

    // set content type and other response header fields first
    response.setContentType( "text/html" );

    // then write the data of the response
    out.println( "<HTML><HEAD><TITLE> CalcServlet Output </TITLE></HEAD><BODY>" );
    out.println( "<h1> CalcServlet Output </h1>" );
    out.println( message );
    out.println( "</BODY></HTML>" );
    out.close();
}

public String getServletInfo()
{
    return "A simple calculator servlet";
}
}

```

CORBA

The examples in this section demonstrate Voyager's CORBA features.

HOLDERS

Files

- ◆ ILottery.idl
- ◆ Lotter.java
- ◆ Person.idl
- ◆ Ticket.idl
- ◆ Server.java
- ◆ Client.java

To compile the example, change to \voyager\examples\corba\holder, and type:

```
cgen ILottery.idl Person.idl Ticket.idl -v
javac *.java
```

Window1

```
% java Server
lottery IOR = IOR:000000000000001149444c3a494c6f7474657279
CORBA server is ready
money before purchase = 30
money after $10 purchase = 20
set ticket to Ticket( 23, 42, 11 )
```

Window2

```
% java Client
bank IOR = IOR:000000000000001149444c3a494c6f747465727
money before purchase = 30
money after purchase = 20
ticket = Ticket( 23, 42, 11 )
```

Arrays

Files

- ◆ IMath.idl
- ◆ Math.java
- ◆ Server.java
- ◆ Client.java

To compile the example, change to \voyager\examples\corba\array, and type:

```
cgen IMath.idl -v
javac *.java
```

Window1

```
% java Server
math IOR = IOR:000000000000000e49444c3a494d6174683a312
CORBA server is ready
INVOKE addLongArray
INVOKE addLongSequence
INVOKE multiMatrix
multiply 2 by 3 array
```

Window1

```
% java Client
math IOR = IOR:000000000000000e49444c3a494d6174683a312e3000000
2 + 6 + 8 = 16
1 + 5 + 6 + 8 = 20
matrix before = { 2 3 1 } { 4 6 7 }
matrix after = { 4 6 2 } { 8 12 14 }
```

Enums

Files

- ◆ Color.idl
- ◆ Enum.java

To compile the example, change to \voyager\examples\corba\enum, and type:

```
cgen Color.idl -v
javac *.java
```

Window1

```
% java Enum
Color( red ), Color( green ), Color( blue )
Color._blue = 2
Color.blue.value() = 2
Color.from_int( 2 ) = Color( blue )
```

Structs

Files

- ◆ Address.idl
- ◆ Person.idl
- ◆ IPostOffice.idl

- ◆ PostOffice.java
- ◆ Server.java
- ◆ Client.java

To compile the example, change to \voyager\examples\corba\struct, and type:

```
cgen Address.idl Person.idl IPostOffice.idl -h -v
javac *.java
```

Window1

```
% java Client
post office IOR = IOR:000000000000001449444c3a49506f73744f666666963653a312e3000
Address of Person( Graham, 36 ) is Address( 123 arkwright road, TX, 75248 )
```

Window2

```
% java Server
postoffice IOR = IOR:000000000000001449444c3a49506f73744f6666
CORBA server is ready
INVOKE getAddress
getAddress( Person( Graham, 36 ) )
name = Graham
age = 36
return Address( 123 arkwright road, TX, 75248 )
```

Unions

Files

- ◆ Winnings.idl
- ◆ Union.java

To compile the example, change to \voyager\examples\corba\union, and type:

```
cgen Winnings.idl -v
javac *.java
```

Window1

```
% java Union  
regular winnings = 200  
jackpot winnings = 420000  
booby winnings = 0
```

TypeCodes

Files

- ◆ Name.idl
- ◆ IPrinter.idl
- ◆ Printer.java
- ◆ Server.java
- ◆ Client.java

To compile the example, change to \voyager\examples\corba\typecode, and type:

```
cgen IPrinter.idl Name.idl -v  
javac *.java
```

Window1

```
% java Client  
printer IOR = IOR:000000000000001149444c3a495072696e74465723a312
```

Window2

```
% java Server
printer IOR = IOR:000000000000001149444c3a495072696e7465723a312e
CORBA server is ready
any.getObject() = 146
typecode.kind() = 3
long
any.getObject() = voyager
typecode.kind() = 18
string, max length = 0
any.getObject() = Name( james, t, kirk )
typecode.kind() = 15
struct Name
id = IDL:Name:1.0
member 0 name = "first" type = string, max length = 0
member 1 name = "middle" type = char
member 2 name = "last" type = string, max length = 0
```

Attributes

Files

- ◆ ICar.idl
- ◆ Car.java
- ◆ Server.java
- ◆ Client.java

To compile the example, change to \voyager\examples\corba\attribute, and type:

```
cgen ICar.idl -v
javac *.java
```

Window1

```
% java Server  
car IOR = IOR:000000000000000d49444c3a494361723a312e3000000  
CORBA server is ready
```

Window2

```
% java Client  
car IOR = IOR:000000000000000d49444c3a494361723a312e30000  
make = lotus esprit  
speed = 0  
speed = 140
```

Narrowing

Files

- ◆ IX.idl
- ◆ IY.idl
- ◆ ILibrary.idl
- ◆ Library.java
- ◆ Server.java
- ◆ Client.java

To compile the example, change to \voyager\examples\corba\narrow, and type:

```
cgen IX.idl IY.idl ILibrary.idl -v  
javac *.java
```

Window1

```
% java Server  
library IOR = IOR:000000000000001149444c3a494c6962726172  
CORBA server is ready  
printX()  
printY()
```

Window2

```
% java Client  
library IOR = IOR:000000000000001149444c3a494c6962726172793a3
```

Dynamic

Files

- ◆ ITimer.idl
- ◆ Timer.java
- ◆ Server.java
- ◆ Client.java

To compile the example, change to `\voyager\examples\corba\dynamic`, and type:

```
cgen ITimer.idl -v  
javac *.java
```

Window1

```
% java Server  
timer IOR = IOR:00000000000000f49444c3a4954696d65723a3  
CORBA server is ready  
wait( 6000 )  
wait( 6000 )
```


Window2

```
% java Client
timer IOR = IOR:000000000000000f49444c3a4954696d65723a312e3000
invoke OneWay wait( 6000 )
invoke Sync wait( 6000 )
result = 6000
invoke Future wait( 6000 )
result = false
result = 6000
```

Pragma

Files

- ◆ X.idl
- ◆ Pragma.java

To compile the example, change to \voyager\examples\corba\pragma, and type:

```
cgen X.idl -d . -v
javac *.java
```

Window1

```
% java Pragma
A id = IDL:p.q/X/IA:1.0
B id = IDL:s.t/X/IB:3.0
C id = IDL:x.y/IZ:3.0
```

Java to IDL

Files in \voyager\examples\corba\bank

- ◆ IAccount.java
- ◆ IBank.java

- ◆ Account.java
- ◆ Bank.java
- ◆ OverdrawnException.java
- ◆ BatchException.java
- ◆ Server.java
- ◆ Client.java

Files in \voyager\examples\corba\javatoidl

- ◆ IBank.idl
- ◆ IAccount.idl
- ◆ OverdrawnException.idl
- ◆ BatchException.idl
- ◆ IBank.java
- ◆ IAccount.java
- ◆ Client.java
- ◆ OverdrawnException.java
- ◆ BatchException.java

To compile the non-CORBA part of the example, change to \voyager\examples\corba\bank, and type:

```
javac *.java  
cgen IBank IAccount OverdrawnException BatchException -v -d ..\javatoidl
```

To compile the CORBA part of the example, change to \voyager\examples\corba\javatoidl, and type:

```
cgen IBank.idl IAccount.idl OverdrawnException.idl BatchException.idl -v  
javac *.java
```

Window1

```
% cd \voyager\examples\corba\bank
% java Server
bank IOR = IOR:0000000000000000e49444c3a4942616e6b3a31
server is ready
open account
```

Window2

```
% cd \voyager\examples\corba\bank
% java Client
start REGULAR client
account balance = 1000
account balance = 500
OverdrawnException( only have $500, 1500 )
```

Window3

```
% cd \voyager\examples\corba\javatoidl
% java Client
start CORBA client
bank IOR = IOR:0000000000000000e49444c3a4942616e6b3a312e3000000000000001
account balance = 1000
account balance = 500
BatchException( only have $500, 400, 2 )
```

Naming Service

Files

- ◆ IElement.java
- ◆ Element.java
- ◆ Client.java

◆ Server.java

To compile the example, change to \voyager\examples\corba\naming, and type:

```
javac *.java
```

Window1

```
% voyager 8000  
voyager orb professional 3.1, copyright objectspace 1997-1999
```

Window2

```
% java Server  
server ready
```

Window3

```
% java Client  
Elements/CA -> Calcium  
Elements/AU -> Gold  
got 2 bindings  
binding 0: NameComponent( ZN, )  
binding 1: NameComponent( AU, )  
iterator returns: NameComponent( AG, )  
iterator returns: NameComponent( CA, )
```

RMI

The examples in this section demonstrate Voyager's RMI features.

Note: Examples 1 and 2 will require that RMI stubs be generated for the Stockmarket classes from the first examples. If this has not yet been done, run `rmic examples.stockmarket.RmiStockmarket` at the command line and make sure that the resulting stub and skeleton file are placed in the same directory as the `RmiStockmarket` class.

RMI Example 1

An object is registered by RMI with an `rmiregistry`. A Voyager client is able to lookup the object and invoke a method on it.

Start `rmiregistry` running on port 8000 in one window. In a second window, start `RmiServer1`. It will register a stockmarket with the `rmiregistry` on port 8000. In a third window, run `VoyagerClient1`. It will look up the registered stockmarket and interact with it.

Window 1

```
> rmiregistry 8000
```

Window 2

```
> java examples.rmi.RmiServer1  
construct stockmarket, export the object with UnicastRemoteObject  
Buying 200 of SUNW  
news: Voyager can be an RMI client!  
Selling 200 of CSC0
```

Window 3

```
> java examples.rmi.VoyagerClient1
```

The source code for RmiServer1.java and VoyagerClient1.java follows.

Interface examples\rmi\RmiServer1.java

```
// copyright 1997 - 1999 objectspace
```

```
package examples.rmi;
```

```
import java.util.*;
import java.rmi.*;
import examples.stockmarket.RmiStockmarket;

public class RmiServer1
{
    public static void main( String[] args )
    {
        try
        {
            // true => export the object through UnicastRemoteObject
            Naming.bind( "rmi://:8000/NASDAQ", new RmiStockmarket( true ) );
        }
        catch( Exception exception )
        {
            exception.printStackTrace();
        }
    }
}
```

Interface examples\rmi\VoyagerClient1.java

```
// copyright 1997 - 1999 objectspace
```

```
package examples.rmi;
```

```
import com.objectspace.voyager.*;
import examples.stockmarket.IRmiStockmarket;
```

```
public class VoyagerClient1
```

```

{
public static void main( String[] args )
{
try
{
Voyager.startup();

// lookup the object in the RMI registry running on port 8000
IRmiStockmarket market = (IRmiStockmarket) Namespace.lookup( "rmi://localhost:8000/NASDAQ" );

market.buy( 200, "SUNW" );
market.news( "Voyager can be an RMI client!" );
market.sell( 200, "CSCO" );

Voyager.shutdown();
}
catch( Exception exception )
{
exception.printStackTrace();
}
}
}

```

RMI Example 2

In this example, Voyager will export a stockmarket to an external rmiregistry. A pure-RMI client looks up the stockmarket and interacts with it.

In one window, start rmiregistry on its default port (1099). In a second window, start VoyagerServer1. It will register a Stockmarket object with the rmiregistry. In a third window, start RmiClient1. The client will look up the Stockmarket object in the registry and invoke some of its methods.

Note: If you are using Java2 (JDK 1.2) to run this example, then you will have to set up your .java.policy file to allow the Voyager classes and the example classes permission to run this code. Refer to the documentation for Java2 Security for more detail.

Window 1

```
> rmiregistry
```

Window 2

```
> java examples.rmi.VoyagerServer1  
construct stockmarket  
BOUND!  
Buying 200 of SUNW  
news: Voyager can be an RMI server!  
Selling 200 of CSCO
```

Window 3

```
> java examples.rmi.RmiClient1  
BINDING...  
BOUND!
```

The source code for VoyagerServer1.java and RmiClient1.java follows.

Interface examples\rmi\VoyagerServer1.java

```
// copyright 1997 - 1999 objectspace  
package examples.rmi;  
  
import com.objectspace.voyager.*;  
import com.objectspace.voyager.rmi.*;  
import examples.stockmarket.RmiStockmarket;  
  
public class VoyagerServer1  
{  
    public static void main( String[] args )  
    {  
        try
```



```

{
    Voyager.startup( "8000" );

    // enable the resource server, so this instance of Voyager
    // can serve the classes to the RMI registry
    ClassManager.enableResourceServer();

    // set the server's codebase to point to this instance of Voyager.
    RmiRegistry.setServerCodebase( "http://localhost:8000" );

    // bind stockmarket to RMI registry running on port 1099.
    // voyager generates the stubs dynamically so rmic never has to be run.
    // false => do not export object through UnicastRemoteObject
    Namespace.bind( "rmi:NASDAQ", new RmiStockmarket( false ) );

    System.out.println( "BOUND!" );
}
catch( Exception exception )
{
    {
        exception.printStackTrace();
    }
}
}
}

```

Interface examples\rmi\RmiClient1.java

// copyright 1997 - 1999 objectspace

```

package examples.rmi;

import java.rmi.*;
import examples.stockmarket.IRmiStockmarket;

public class RmiClient1
{
    public static void main( String[] args )
    {
        try
        {
            // install RMI security manager
            System.setSecurityManager( new RMISecurityManager() );

            // lookup object in the RMI registry running on port 1099

```

```

System.out.println( "BINDING..." );
IRmiStockmarket market = (IRmiStockmarket) Naming.lookup( "rmi:NASDAQ" );
System.out.println( "BOUND!" );

market.buy( 200, "SUNW" );
market.news( "Voyager can be an RMI server!" );
market.sell( 200, "CSCO" );
}
catch( Exception exception )
{
    exception.printStackTrace();
}
}
}

```

RMI Example 3

In this example, the Voyager server takes on the duties of rmiregistry itself. It creates and exports a Stockmarket object. A pure-RMI client is able to query the Voyager server using standard RMI lookups, access the Stockmarket object, and invoke methods on it. Best of all, the Voyager server is capable of generating the RMI stub classes when needed, so `rmic` is not required to generate the stub and skeleton classes beforehand.

In one window, run `VoyagerServer2`. The server will provide rmiregistry behavior on port 8000, and it will create a Stockmarket object bound in that registry. In a second window, `RmiClient2` will look up that object through pure RMI calls and invoke methods on the Stockmarket object.

Note: If you are using Java2 (JDK 1.2) to run this example, then you will have to set up your `.java.policy` file to allow the Voyager classes and the example classes permission to run this code. Refer to the documentation for Java2 Security for more detail.

Window 1

```
> java examples.rmi.VoyagerServer2  
construct stockmarket  
BOUND!  
Buying 200 of SUNW  
news: Voyager can host an RMI registry!  
Selling 200 of CSC0
```

Window 2

```
> java examples.rmi.RmiClient2  
BINDING...  
BOUND!
```

The source code for VoyagerServer2.java and RmiClient2.java follows.

Interface examples\rmi\VoyagerServer2.java

```
// copyright 1997 - 1999 objectspace  
package examples.rmi;  
  
import com.objectspace.voyager.*;  
import com.objectspace.voyager.rmi.*;  
import examples.stockmarket.RmiStockmarket;  
  
public class VoyagerServer2  
{  
    public static void main( String[] args )  
    {  
        try  
        {  
            Voyager.startup( "8000" );  
  
            com.objectspace.lib.util.Console.setLogLevel( "verbose" );  
            // enable the resource server, so this instance of Voyager  
            // can serve the classes to the RMI registry  
            ClassManager.enableResourceServer();  
        }  
    }  
}
```

```

// set the server's codebase to point to this instance of Voyager
RmiRegistry.setServerCodebase( "http://gglass1.objectspace.com:8000" );

// bind stockmarket to RMI registry running on port 8000.
// Voyager generates the stubs dynamically so rmic never has to be run.
// false => do not export object through UnicastRemoteObject
Namespace.bind( "rmi://localhost:8000/NASDAQ", new RmiStockmarket( false ) );

    System.out.println( "BOUND!" );
}
catch( Exception exception )
{
    exception.printStackTrace();
}
}
}

```

Interface examples\rmi\RmiClient2.java

```

// copyright 1997 - 1999 objectspace

package examples.rmi;

import java.rmi.*;
import examples.stockmarket.IRmiStockmarket;

public class RmiClient2
{
    public static void main( String[] args )
    {
        try
        {
            // install RMI security manager
            System.setSecurityManager( new RMISecurityManager() );

            // lookup object in the RMI registry running on port 8000
            System.out.println( "BINDING..." );
            IRmiStockmarket market = (IRmiStockmarket) Naming.lookup( "rmi://:8000/NASDAQ" );
            System.out.println( "BOUND!" );

            market.buy( 200, "SUNW" );
            market.news( "Voyager can host an RMI registry!" );

```

```
        market.sell( 200, "CSCO" );  
    }  
    catch( Exception exception )  
    {  
        exception.printStackTrace();  
    }  
}  
}
```

DCOM



The examples in this section demonstrate Voyager's COM/DCOM features.

DCOM Example 1

An object is registered in a Voyager namespace. A Visual Basic client is able to look up the object and invoke methods on it.

Step 1. Start the `examples.calculator.CalcServer` class in one window. This will start a Voyager server at port 8000 and bind a Calculator object into its namespace as `MyCalculator`. In a second window, run the `VBCalcClient` program.

Step 2. From `examples\calculator`, compile the calculator files:

```
javac ICalculator.java Calculator.java CalcServer.java
```

Step 3. Run `CalcServer` in one window as follows:

Window 1

```
java examples.calculator.CalcServer
```

Step 4. To build the `VBCalcClient` application, open the Visual Basic project `examples\dcom\VBCalcClient.vbp` file in Visual Basic 6.0. Go to the Project menu and select References. In the dialog box that pops up, click on the Browse button. A file selection dialog box will appear. Traverse to the Voyager installation directory and select `bin\VoyagerBridge.dll`. Close all dialog boxes. Go to the File menu and select Make `VBCalcClient.exe`.

Step 5. To run the client application, the Microsoft Java Virtual Machine (MSVM) must be present on the client machine. Also, the system `CLASSPATH` on the client machine must include the Voyager installation directory and the `Voyager.jar` file.

Step 6. Now run the client application in another window as follows, specifying the name of machine running the CalcServer:

Window 2

VBCalcClient *calcServerMachineName*

A message box will pop up with the message:

3 + 4 = 7

The same message will also appear in the first window.

Window 1

> java examples.calculator.CalcServer

3 + 4 = 7

The source code for VBCalcClient follows.

Module VBCalcClient.bas

```
Public Sub Main()  
    Dim namespace As VoyagerLib.INamespace  
    Dim adder As Object  
    Dim x As Long  
    Dim y As Long  
    Dim sum As Long  
  
    ' Create namespace bridge  
    Set namespace = CreateObject("VoyagerLib.Bridge")  
  
    ' Lookup calculator object in the server namespace  
    Set adder = namespace.Lookup("//" & Command & ":8000/MyCalculator")  
    x = 3  
    y = 4  
  
    ' Method invocation  
    sum = adder.Add(x, y)  
  
    MsgBox ("3 + 4 = " & sum)  
End Sub
```

The source code for the Java classes follows.

```
Interface examples\calculator\ICalculator.java
// copyright 1997-1999 objectspace
```

```
package examples.calculator;
```

```
public interface ICalculator
{
    int add( int x, int y );
}
```

```
Class examples\calculator\Calculator.java
// copyright 1997-1999 objectspace
```

```
package examples.calculator;
```

```
public class Calculator implements ICalculator
{
    public int add( int x, int y )
    {
        int result = x + y;
        System.out.println( x + " + " + y + " = " + result );
        return result;
    }
}
```

Program examples\calculator\CalcServer.java

```
// copyright 1997-1999 objectspace
```

```
package examples.calculator;
```

```
import com.objectspace.voyager.*;
```

```
public class CalcServer
{
    static ICalculator calculator;

    public static void main( String args[] )
    {
        try
        {
            // enables remote classloading from this program

```



```

ClassManager.enableResourceServer();

// starts Voyager on port 8000
Voyager.startup( "8000" );

// construct the calculator
calculator = (ICalculator) Factory.create( "examples.calculator.Calculator" );
Namespace.rebind( "MyCalculator", calculator );
}
catch( Exception exception )
{
    System.err.println( exception );
}
}
}

```

DCOM Example 2

A COM object is registered in a Voyager namespace. A Voyager client is able to look up the object and invoke methods on it.

The Calculator object from the previous example is implemented as a COM object in this example. Tools supplied with the Microsoft Platform SDK and the Microsoft SDK for Java 3.2 are required to build and run this example.

The Calculator object is packaged as a COM component in the file `examples\dcom\Calculator.dll`. The details of how the DLL file was created are beyond the scope of this example

To use the COM class from Java, run the DLL file through the `jactivex.exe` tool supplied with the Microsoft SDK for Java:

```
jactivex -d ..\.. -p:b -p examples.dcom Calculator.dll
```

This will produce a java interface `examples.dcom.ICalculator` and a java class `examples.dcom.Calculator`. These are java wrappers for the original COM interface and object, respectively.

Compile all the java classes in the `examples\dcom` directory using the Microsoft `jvc` compiler available with the Microsoft SDK for Java.

```
jvc /x- *.java
```

Before running the example, register the Calculator.dll file in examples\dcom directory using the regsvr32.exe tool included with the Microsoft Platform SDK as follows:

```
regsvr32 Calculator.dll
```

Run the server class examples.dcom.CalcServer in one window using the MSVM jview. Run the client class examples.dcom.CalcClient in another window using jview.

Window 1

```
jview examples.dcom.CalcServer  
3 + 4 = 7
```

Window 2

```
jview examples.dcom.CalcClient  
3 + 4 = 7
```

The source code for the client and the server classes follows.

Program examples\dcom\CalcServer.java

```
// copyright 1997-1999 objectspace  
  
package examples.dcom;  
  
import com.objectspace.voyager.*;  
  
public class CalcServer  
{  
    static ICalculator calculator;  
  
    public static void main( String args[] )  
    {  
        try  
        {  
            // starts Voyager on port 8000  
            Voyager.startup( "8000" );  
  
            // Binds calculator factory into the namespace  
            Namespace.bind( "MyCalculator",  
                new com.objectspace.voyager.directory.ClassFactory( "examples.dcom.Calculator" ) );  
        }  
        catch( Exception exception )
```

```

    {
        System.err.println( exception );
    }
}
}

```

Program examples\dcom\CalcClient.java

// copyright 1997-1999 objectspace

```
package examples.dcom;
```

```
import com.objectspace.voyager.*;
```

```
public class CalcClient
```

```
{
    public static void main( String[] args )
```

```
{
    try
    {
        // start up Voyager
        Voyager.startup();
```

```
        // look up a calculator object
        ICalculator calc = (ICalculator)Namespace.lookup( "8000/MyCalculator" );
```

```
        // send message
        int sum = calc.add( 3, 4 );
        System.out.println( "3 + 4 = " + sum );
```

```
        // shutdown voyager
        Voyager.shutdown();
    }
    catch( Exception exception )
    {
        exception.printStackTrace();
    }
}
}

```

Ultra-Light Client

t



The examples in this section demonstrate how Voyager works with ultra-light clients.

Ultra-Light Client Example

The LightCalcClient example is a simple example of an applet that uses Voyager. The CalcServer program exports a calculator object capable of adding two numbers. The applet receives a reference to this calculator object and uses it to add numbers entered by the user.

From examples\calculator, compile the Calculator utility classes program:

```
javac ICalculator.java Calculator.java CalcServer.java
```

From examples\applets, compile the example program:

```
javac LightCalcApplet.java
```

Run an instance of Voyager server with enabled resource server on port 9000 in a second window. Modify LightCalculator.html so the codebase tag points to the Voyager server you just started. Make sure lightclient.jar is available in the directory where LightCalculator.html is located. Start a web browser, and load LightCalculator.html.

Note: The codebase tag in LightCalculator.html must contain the URL pointing to the CalcServer. For example, if CalcServer is running on host dallas:8000 then the URL should be `http://dallas:8000/`. In addition, lightclient.jar should be in the classpath of CalcServer.

Window1

```
java -Dclass_based_proxies=true examples.calculator.CalcServer  
4 + 5 = 9
```

Window2

```
>appletviewer http://<Voyager Server:port>/LightCalculator.html
```

The source code for ICalculator.java, Calculator.java, CalcServer.java, CalcApplet.java and Calculator.html follows:

Applet examples\applets\LightCalcApplet.java

```
// copyright 1997-1999 objectspace
```

```
import java.applet.*;  
import java.awt.*;  
import java.awt.event.*;  
import com.objectspace.voyager.*;  
import examples.calculator.*;
```

```
public class LightCalcApplet extends Applet  
{  
    TextField operand1 = new TextField( "", 3 );  
    Label plus = new Label( "+" );  
    TextField operand2 = new TextField( "", 3 );  
    Button add = new Button( "=" );  
    TextField result = new TextField( "", 3 );  
    ICalculator calculator;  
  
    public void init()  
    {
```

```

setLayout( new FlowLayout() );
add( operand1 );
add( plus );
add( operand2 );
add( add );
add( result );
add.addActionListener( new ActionListener()
{
    public void actionPerformed( ActionEvent event )
    {
        add();
    }
});
}

public void start()
{
    try
    {
        Namespace.setServerURL( this );
        calculator = (ICalculator) Namespace.lookup( "MyCalculator" );
    }
    catch( Exception exception )
    {
        exception.printStackTrace();
    }
}

void add()
{
    int x = Integer.parseInt( operand1.getText() );
    int y = Integer.parseInt( operand2.getText() );

    try
    {
        result.setText( Integer.toString( calculator.add( x, y ) ) );
    }
    catch( Exception exception )
    {
        result.setText("Error");
    }
}

```

}

HTML examples\applets\LightCalculator.html

```
<html>
<head><title>Calculator Applet</title></head>
<body><hr><p>
<applet codebase="http://<address of the Voyager server goes here>" code="LightCalcApplet"
align="baseline" width="300" height="50" archive="lightclient.jar" VIEWASTEXT>
</applet>
</p><hr></body>
</html>
```

Timers

The examples in this section demonstrate Voyager's Timing services.

Stopwatch1 Example

The Stopwatch1 example demonstrates use of the Stopwatch class to clock time intervals.

From examples\timer, compile the program:

```
javac Stopwatch1.java
```

Run the Stopwatch1 program from examples\timer.

Window1

```
>java examples.timer.Stopwatch1
start the stopwatch and then sleep for 2 seconds
stop the stopwatch
lap count = 1
lap time = 2022
total time = 2022
pause for 4 seconds
start the stopwatch and then sleep for 3 seconds
stop the stopwatch
lap count = 2
lap time = 3004
total time = 5037
average lap time = 2518.5
lap times = [ 2023 3014 ]

>
```

The source code for Stopwatch1.java follows:

Program examples\timer\Stopwatch1.java

```
// copyright 1997-1999 objectspace
```

```
package examples.timer;
```

```

import com.objectspace.lib.timer.*;

public class Stopwatch1
{
    public static void main( String args[] )
    {
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.setRecordLapTimes( true ); // record individual lap times

        System.out.println( "start the stopwatch and then sleep for 2 seconds" );
        stopwatch.start();
        try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}
        System.out.println( "stop the stopwatch" );
        stopwatch.stop();
        System.out.println( "lap count = " + stopwatch.getLapCount() );
        System.out.println( "lap time = " + stopwatch.getLapTime() );
        System.out.println( "total time = " + stopwatch.getTotalTime() );

        System.out.println( "pause for 4 seconds" );
        try{ Thread.sleep( 4000 ); } catch( InterruptedException exception ) {}

        System.out.println( "start the stopwatch and then sleep for 3 seconds" );
        stopwatch.start();
        try{ Thread.sleep( 3000 ); } catch( InterruptedException exception ) {}
        System.out.println( "stop the stopwatch" );
        stopwatch.stop();
        System.out.println( "lap count = " + stopwatch.getLapCount() );
        System.out.println( "lap time = " + stopwatch.getLapTime() );
        System.out.println( "total time = " + stopwatch.getTotalTime() );
        System.out.println( "average lap time = " + stopwatch.getAverageLapTime() );
        long[] times = stopwatch.getLapTimes();
        System.out.print( "lap times = [ " );
        for( int i = 0; i < times.length; i++ )
            System.out.print( times[ i ] + " " );
        System.out.println( "]" );
    }
}

```

Timer1 Example

The Timer1 example demonstrates TimerListeners listening for an alarm using shared thread notification. The first listener's event notification completes before the second listener's event notification begins.

From examples\timer, compile the program:

```
javac SleepyHead.java Timer1.java
```

Run the Timer1 program from examples\timer.

Window1

```
>java examples.timer.Timer1
notify after 10 seconds using shared thread
main program sleeps for 20 seconds...
enter timerExpired( TimerEvent( 903768890498 ) )
sleep 2 seconds...
exit timerExpired( TimerEvent( 903768890498 ) )
enter timerExpired( TimerEvent( 903768890498 ) )
sleep 2 seconds...
exit timerExpired( TimerEvent( 903768890498 ) )
main program terminates

>
```

The source code for SleepyHead.java and Timer1.java follows:

Class examples\timer\SleepyHead.java

```
// copyright 1997-1999 objectspace

package examples.timer;

import com.objectspace.lib.timer.*;

public class SleepyHead implements TimerListener
{
    public void timerExpired( TimerEvent event )
    {
        System.out.println( "enter timerExpired( " + event + " )");
        System.out.println( "sleep 2 seconds..." );
    }
}
```

```

    try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}
    System.out.println( "exit timerExpired( " + event + " )" );
  }
}

```

Program examples\timer\Timer1.java

// copyright 1997-1999 objectspace

```
package examples.timer;
```

```
import com.objectspace.lib.timer.*;
```

```

public class Timer1
{
    public static void main( String args[] )
    {
        SleepyHead sleepy1 = new SleepyHead();
        SleepyHead sleepy2 = new SleepyHead();

        System.out.println( "notify after 10 seconds using shared thread" );
        Timer timer = new Timer();
        timer.addTimerListener( sleepy1 );
        timer.addTimerListener( sleepy2 );
        timer.alarmAfter( 10000 );

        System.out.println( "main program sleeps for 20 seconds..." );
        try{ Thread.sleep( 20000 ); } catch( InterruptedException exception ) {}
        System.out.println( "main program terminates" );
    }
}

```

Timer2 Example

The Timer2 example demonstrates TimerListeners listening from within different timer groups. The event notifications are interlaced due to the multi-threaded listening.

From examples\timer, compile the program:

```
javac Timer2.java
```

Run the Timer2 program from examples\timer.

Window1

```
>java examples.timer.Timer2
timer1 is member of default timer group
timer1 will wake up sleepy1 every 10 seconds
timer2 is member of explicit timer group
timer2 will wake up sleepy2 every 10 seconds
main program sleeps for 25 seconds...
enter timerExpired( TimerEvent( 903769450483 ) )
sleep 2 seconds...
enter timerExpired( TimerEvent( 903769450413 ) )
sleep 2 seconds...
exit timerExpired( TimerEvent( 903769450483 ) )
exit timerExpired( TimerEvent( 903769450413 ) )
enter timerExpired( TimerEvent( 903769460417 ) )
sleep 2 seconds...
enter timerExpired( TimerEvent( 903769460487 ) )
sleep 2 seconds...
exit timerExpired( TimerEvent( 903769460417 ) )
exit timerExpired( TimerEvent( 903769460487 ) )
main program terminates
```

>

The source code for Timer2.java follows:

Program examples\timer\Timer2.java

```
// copyright 1997-1999 objectspace
```

```
package examples.timer;
```

```
import com.objectspace.lib.timer.*;
```

```
public class Timer2
{
    public static void main( String args[] )
    {
        SleepyHead sleepy1 = new SleepyHead();
        SleepyHead sleepy2 = new SleepyHead();
```

```
        System.out.println( "timer1 is member of default timer group" );
```

```

Timer timer1 = new Timer(); // default timer group, Thread.NORM_PRIORITY
timer1.addTimerListener( new TimerListenerThread( sleepy1 ) );
System.out.println( "timer1 will wake up sleepy1 every 10 seconds" );
timer1.alarmEvery( 10000 );

System.out.println( "timer2 is member of explicit timer group" );
TimerGroup group = new TimerGroup( Thread.MAX_PRIORITY );
Timer timer2 = new Timer( group ); // explicit timer group
timer2.addTimerListener( new TimerListenerThread( sleepy2 ) );
System.out.println( "timer2 will wake up sleepy2 every 10 seconds" );
timer2.alarmEvery( 10000 );

System.out.println( "main program sleeps for 25 seconds..." );
try{ Thread.sleep( 25000 ); } catch( InterruptedException exception ) {}
System.out.println( "main program terminates" );
}
}

```

Timer3 Example

The Timer3 example demonstrates TimerListeners listening for an alarm using separate thread notification. The event notifications are interlaced due to the multi-threaded listening.

From examples\timer, compile the program:

```
javac Timer3.java
```

Run the Timer3 program from examples\timer.

Window1

```
>java examples.timer.Timer3
notify after 10 seconds with separate threads
main program sleeps for 20 seconds...
enter timerExpired( TimerEvent( 903769946226 ) )
sleep 2 seconds...
enter timerExpired( TimerEvent( 903769946226 ) )
sleep 2 seconds...
exit timerExpired( TimerEvent( 903769946226 ) )
exit timerExpired( TimerEvent( 903769946226 ) )
main program terminates

>
```

The source code for Timer3.java follows:

Program examples\timer\Timer3.java

```
// copyright 1997-1999 objectspace

package examples.timer;

import com.objectspace.lib.timer.*;

public class Timer3
{
    public static void main( String args[] )
    {
        SleepyHead sleepy1 = new SleepyHead();
        SleepyHead sleepy2 = new SleepyHead();

        System.out.println( "notify after 10 seconds with separate threads" );
        Timer timer = new Timer();
        timer.addTimerListener( new TimerListenerThread( sleepy1 ) );
        timer.addTimerListener( new TimerListenerThread( sleepy2 ) );
        timer.alarmAfter( 10000 );

        System.out.println( "main program sleeps for 20 seconds..." );
        try{ Thread.sleep( 20000 ); } catch( InterruptedException exception ) {}
        System.out.println( "main program terminates" );
    }
}
```

}
}

Replication Examples



Replication Example 1

This example shows how to create a replication cluster from the command line and use the cluster once it is created. In this example, we will be storing our cluster peer list in a directory server that is not a part of the cluster. In this example, we insert a value into the replication peer at port 8000 and attempt to retrieve the value from the peer running on port 7000.

```
package examples.replication;

import com.objectspace.voyager.*;

public class ReplAddGet
{
    public static void main( String[] argv )
    {
        try
        {
            Voyager.startup();
            Namespace.bind( "//localhost:8000/foo", "Value put into 8000" );
            System.out.println( "8000: " + (String) Namespace.lookup( "//localhost:8000/foo" ) );
            System.out.println( "7000: " + (String) Namespace.lookup( "//localhost:7000/foo" ) );
        }
        catch( Exception exception )
        {
            exception.printStackTrace();
        }

        Voyager.shutdown();
    }
}
```

examples/replication/ReplAddGet.java

To run this example, follow these steps:

Step 1. Start the directory server that will maintain the list of replication peers.

```
voyager 10000 -f peers.db
```

It is good practice to store the list of peers in a persistent file, as in this case.

It is also good practice to keep the list of replication peers in a separate directory from the replication cluster. You may use a directory server that is a part of the cluster to maintain this list; however, the list will not be replicated, so if you need to reference the directory server host containing the peer list (for instance, to start additional servers), you must bypass the load balancing mechanism by referencing the directory server host by its IP address.

Step 2. Start the directory peers.

```
voyager 8000 -replication 10000  
voyager 7000 -replication 10000
```

Refer to [“Using Replication” on page 177](#) for a description of the usage and valid formats for the replication peer list host URL.

Step 3. Run the example.

```
java examples.replication.ReplAddGet
```

Output:

```
8000: Value put into 8000  
7000: Value put into 8000
```

As the example shows, the value put into the peer running on port 8000 may be retrieved from either of the replicated peers in the cluster.

Replication Example 2

This example shows how to add a local directory server to a replication cluster programmatically.

The code for this example is as follows:

```
package examples.replication;  
  
import com.objectspace.voyager.*;  
import com.objectspace.voyager.replication.Replication;
```

```

public class ExplicitPeer
{
    public static void main( String[] argv )
    {
        try
        {
            Voyager.startup();

            // Add this server's directory server to the replication cluster, the list is maintained
            // in the directory server at 10000.
            Replication.install( "10000" );
            Namespace.bind( "foo", "foo put into local namespace" );
            System.out.println( "local: " + (String) Namespace.lookup( "foo" ) );
            System.out.println( "8000: " + (String) Namespace.lookup( "//localhost:8000/foo" ) );

            // remove this server's directory server from the cluster.
            Replication.uninstall();
            Namespace.bind( "bar", "bar put into local namespace" );
            System.out.println( "local: " + (String) Namespace.lookup( "bar" ) );
            System.out.println( "8000: " + (String) Namespace.lookup( "//localhost:8000/bar" ) );
        }
        catch( Exception exception )
        {
            exception.printStackTrace();
        }

        Voyager.shutdown();
    }
}

```

examples/replication/ExplicitPeer.java

To run this example, follow these steps:

Step 1. Start the directory server that will maintain the list of replication peers.

```
voyager 10000 -f peers.db
```

Step 2. Start the directory peers.

```
voyager 8000 -replication 10000
```

Step 3. Run the example program.

```
java examples.replication.ExplicitPeer
```

Output:

```
local: foo put into local namespace
8000: foo put into local namespace
local: bar put into local namespace
com.objectspace.voyager.NamespaceException: no object bound to the name
vdir://localhost:8000/bar;
    at
com.objectspace.voyager.directory.DirectoryNamingService.lookup(DirectoryNamingService.java:79)
    at com.objectspace.voyager.CompoundNamespace.lookup(CompoundNamespace.java:73)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.objectspace.voyager.Namespace.invokeNonStatic(Namespace.java:352)
    at com.objectspace.voyager.Namespace.lookup(Namespace.java:113)
    at examples.replication.ExplicitPeer.main(ExplicitPeer.java:25)
```

When the local directory server is a part of the cluster, bind messages to the local namespace are replicated to the other peers in the cluster. Once replication has been uninstalled, messages are no longer replicated to the peers.

Load Balancing Examples



Load Balancing Example 1

This example illustrates how to create a load balanced directory server from the command line and how binds and lookups to a load balanced directory differ from binds and lookups to a non-load balanced directory.

The code for this example is the following:

```
package examples.loadbalance;

import com.objectspace.voyager.*;

public class LBTestA
{
    public static void main( String[] argv )
    {
        try
        {
            Voyager.startup();
            // Servers: 8000 - load balanced directory server
            // 6000 & 7000 - servers that will host our remote objects.

            // First, we try to bind two non-proxy objects to the same name.
            try
            {
                Namespace.bind( "//localhost:8000/foo", "str1" );
                Namespace.bind( "//localhost:8000/foo", "str2" );

                // This will fail because these String objects are serialized rather than proxied
                // to the remote namespace. Only multiple non-directory Proxy objects from distinct hosts
                // may be bound to the same name.
            }
            catch( NamespaceException nsException )
```

```

    {
        System.err.println( "Both strings not allowed: " + nsException.toString() );
    }

    // Second, we try to bind two proxy objects to the same name.
    try
    {
        Proxy remote6000 = Factory.create( "java.lang.String", new Object[]{ "remote - 6000" }, "6000" );
        Namespace.bind( "//localhost:8000/bar", remote6000 );
        Proxy remote7000 = Factory.create( "java.lang.String", new Object[]{ "remote - 7000" }, "7000" );
        Namespace.bind( "//localhost:8000/bar", remote7000 );
        System.err.println( "Both proxies are bound at the same name." );

        // When this succeeds, we retrieve the object at that name and print out what
        // we got. We do this a few times to show that different objects are returned.

        for( int i=0; i < 5; i++ )
            System.out.println( "Retrieved: " + Namespace.lookup( "//localhost:8000/bar" ) );
    }
    catch( NamespaceException nsException )
    {
        System.err.println( "Both proxies not allowed: " + nsException.toString() );
    }
}
catch( Exception exception )
{
    exception.printStackTrace();
}

Voyager.shutdown();
}
}

```

examples/loadbalance/LBTestA.java

To run this example:

Step 1. Start the load balanced directory server.

voyager 8000 -lb

See the load balancing documentation for exact usage of the -lb option.

Step 2. Start the voyager object (application) servers on ports 6000 and 7000.

```
voyager 6000  
voyager 7000
```

These voyager servers will host the objects that we create in our example.

Step 3. Run the example code

```
java examples.loadbalance.LBTestA
```

Output:

Both strings not allowed: com.objectspace.voyager.NamespaceException: name is already used by another object

Both proxies are bound at the same name.

Retrieved: remote - 6000

Retrieved: remote - 7000

Retrieved: remote - 6000

Retrieved: remote - 7000

Retrieved: remote - 6000

Load Balancing Example 2

This example shows how to create a load balanced directory within your application code.

The code for this example is the following:

```
package examples.loadbalance;
```

```
import com.objectspace.voyager.*;
```

```
import com.objectspace.voyager.loadbalance.LoadBalancedDirectory;
```

```
public class LBTestB
```

```
{
```

```
    public static void main( String[] argv )
```

```
    {
```

```
        try
```

```
        {
```

```
            Voyager.startup();
```

```
            // Servers: 6000 & 7000 - servers that will host our remote objects.
```

```
            // First, create a load balanced directory and bind it to a name in our local
```

```
            // namespace
```

```

Namespace.bind( "loadbalance", new LoadBalancedDirectory() );

// Then we try to bind two non-proxy objects to the same name in that directory.
try
{
    Namespace.bind( "loadbalance/foo", "str1" );
    Namespace.bind( "loadbalance/foo", "str2" );

    // This will fail because these String objects are serialized rather than proxied
    // to the remote namespace. Only multiple non-directory Proxy objects from distinct hosts
    // may be bound to the same name.
}
catch( NamespaceException nsException )
{
    System.err.println( "Both strings not allowed: " + nsException.toString() );
}

// Now we try to bind two proxy objects to the same name.
try
{
    Proxy remote6000 = Factory.create( "java.lang.String", new Object[]{ "remote - 6000" }, "6000" );
    Namespace.bind( "loadbalance/bar", remote6000 );
    Proxy remote7000 = Factory.create( "java.lang.String", new Object[]{ "remote - 7000" }, "7000" );
    Namespace.bind( "loadbalance/bar", remote7000 );
    System.err.println( "Both proxies are bound at the same name." );

    // When this succeeds, we retrieve the object at that name and print out what
    // we got. We do this a few times to show that different objects are returned.

    for( int i=0; i < 5; i++ )
        System.out.println( "Retrieved: " + Namespace.lookup( "loadbalance/bar" ) );
}
catch( NamespaceException nsException )
{
    System.err.println( "Both proxies not allowed: " + nsException.toString() );
}
}
catch( Exception exception )
{
    exception.printStackTrace();
}

```



```
Voyager.shutdown();  
}  
}
```

examples/loadbalance/LBTestB.java

To run this example:

Step 1. Start the voyager object (application) servers on ports 6000 and 7000.

voyager 6000

voyager 7000

These voyager servers will host the objects that we create in our example.

Step 2. Run the example code.

```
java examples.loadbalance.LBTestB
```

Output:

Both strings not allowed: com.objectspace.voyager.NamespaceException: name is already used by another object

Both proxies are bound at the same name.

Retrieved: remote - 6000

Retrieved: remote - 7000

Retrieved: remote - 6000

Retrieved: remote - 7000

Retrieved: remote - 6000

Load Balancing Example 3

This example shows how to correctly add a new subdirectory to a load balanced directory.

The code for this example is the following:

```
package examples.loadbalance;
```

```
import com.objectspace.voyager.*;
```

```
import com.objectspace.voyager.directory.*;
```

```

public class PutDirectory
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup();
            // we explicitly create our directory
            IDirectory dir = new Directory();
            // bind our new directory into the load balanced namespace.
            Namespace.bind( "8000/dir", dir );

            // first, we'll add values directly to the directory (incorrect).
            try
            {
                Proxy remote6000 = Factory.create( "java.lang.String", new Object[]{ "remote - 6000" }, "6000" );
                dir.add( "foo", remote6000 );
                Proxy remote7000 = Factory.create( "java.lang.String", new Object[]{ "remote - 7000" }, "7000" );
                dir.add( "foo", remote7000 );
                System.out.println( "both values added" );
            }
            catch( DirectoryException exception )
            {
                // We should get an exception because this directory isn't load balanced.
                System.out.println( "both values not added" );
                exception.printStackTrace();
            }

            // clear out whatever is there.
            dir.clear();
            // Now we get a reference to the load balanced version of this directory.
            dir = (IDirectory) Namespace.lookup( "8000/dir" );

            // Both values are now allowed because the directory is now load balanced.
            try
            {
                Proxy remote6000 = Factory.create( "java.lang.String", new Object[]{ "remote - 6000" }, "6000" );
                dir.add( "foo", remote6000 );
                Proxy remote7000 = Factory.create( "java.lang.String", new Object[]{ "remote - 7000" }, "7000" );
                dir.add( "foo", remote7000 );
                System.out.println( "both values added" );
            }
        }
    }
}

```

```

        catch( DirectoryException exception )
        {
            System.out.println( "both values not added" );
            exception.printStackTrace();
        }
    }
    catch( Exception exception )
    {
        exception.printStackTrace();
    }

    Voyager.shutdown();
}
}

```

examples/loadbalance/PutDirectory.java

To run this example:

Step 1. Start the load balanced directory server on port 8000.

```
voyager 8000 -lb
```

See the load balancing documentation for exact usage of the `-lb` option.

Step 2. Start the voyager object (application) servers on ports 6000 and 7000.

```
voyager 6000
```

```
voyager 7000
```

These voyager servers will host the objects that we create in our example.

Step 3. Run the example code.

```
java examples.loadbalance.PutDirectory
```

Output:

```
both values not added
```

```
com.objectspace.voyager.directory.DirectoryException: name is already used by another object
    at com.objectspace.voyager.directory.Directory.add(Directory.java:160)
    at examples.loadbalance.PutDirectory.main(PutDirectory.java:24)
```

```
both values added
```

As the output shows, once the directory object created is correctly set, it will behave as a load balanced directory and allow multiple binds to the same name.

Load Balancing Example 4

This example illustrates how to create a custom load balancing policy.

```
package examples.loadbalance;

import com.objectspace.voyager.loadbalance.ILoadBalancePolicy;
import com.objectspace.voyager.loadbalance.HostList;
import com.objectspace.voyager.management.IManageable;

/**
 * A simple load balancing policy that always returns the first object in the list.
 */
public class SimplePolicy implements ILoadBalancePolicy
{
    public Object chooseObject( HostList list )
    {
        // Returns the first proxy in the list.
        return list.proxies().nextElement();
    }

    public HostList createList()
    {
        // We don't need any special processing, so we'll just use the standard HostList.
        return new HostList();
    }

    public IManageable getManager()
    {
        // to satisfy the IPolicy interface
        return null;
    }
}
```

examples/loadbalance/SimplePolicy.java

To run this example:

Step 1. Start the load balanced directory server.

```
> voyager 8000 -lb -lb_policy examples.loadbalance.SimplePolicy
```

Step 2. Start the voyager object (application) servers on ports 6000 and 7000.

```
voyager 6000  
voyager 7000
```

These voyager servers will host the objects that we create in our example.

Step 3. Run the example code.

```
java examples.loadbalance.LBTestA
```

Output:

```
Both strings not allowed:  
com.objectspace.voyager.NamespaceException: name is already used by another  
object  
Both proxies are bound at the same name.  
Retrieved: remote - 6000  
Retrieved: remote - 6000  
Retrieved: remote - 6000  
Retrieved: remote - 6000  
Retrieved: remote - 6000
```

Load Balancing Example 5

This example demonstrates the affinity characteristic of the Voyager Load Balanced Directory Services.

The code for examples 5 and 6 is the same, and you can find it in `examples/loadbalance/Affinity.java`.

```
package examples.loadbalance;  
  
import com.objectspace.voyager.*;  
  
public class Affinity  
{  
    public static void main( String[] args )  
    {  
        try  
        {  
            Voyager.startup();
```

```

// Put a remote proxy into the directory.
Namespace.bind( "8000/foo", Factory.create( "java.lang.String", new Object[]{ "remote" }, "7000" ) );
// Put a local proxy into the directory.
Namespace.bind( "8000/foo", Proxy.of( "local" ) );

// Iterate through and see what we get with the lookup.
for( int i=0; i < 5; i++ )
    System.out.println( Namespace.lookup( "8000/foo" ).toString() );

}
catch( Exception exception )
{
    exception.printStackTrace();
}

Voyager.shutdown();
}
}

```

examples/loadbalance/Affinity.java

This example creates a remote object proxy and places it in the load balanced directory. Then a local object is placed into the directory, bound to the same name. Finally, we repeatedly retrieve the object bound to that name and see what is returned.

To run this example:

Step 1. Start a voyager server at port 7000.

```
voyager 7000
```

Step 2. Start a load balanced directory server on port 8000.

```
voyager 8000 -lb
```

Step 3. Compile and run the Affinity.java example, and you should see the following output:

```
java examples.loadbalance.Affinity
```

Output:

```

local
local
local

```

```
local  
local
```

Load Balancing Example 6

This example demonstrates the behavior of the Voyager Load Balanced Directory Service when affinity is disabled.

To run this example:

Step 1. Start a voyager server on port 7000.

```
voyager 7000
```

Step 2. Start a load balanced directory service on port 8000 with affinity disabled. (Note: This example assumes you use the Sun JDK.)

```
voyager 8000 -lb -x -Dobjectspace.loadbalance.affinity_disabled=true
```

Step 3. Compile and run the Affinity.java example and you should now see the following output:

```
java examples.loadbalance.Affinity
```

Output:

```
remote  
local  
remote  
local  
remote
```

Configuration and Management



The examples in this section demonstrate extending Voyager's configuration and management frameworks and using them in a deployed system by converting VoyagerDB into a simple, configurable Voyager service. Use the VoyagerDB add-on to run this example.

From examples\management, compile all of the Java files:

```
javac *.java
```

Start a Voyager Directory Server in one window on port 8000 with its root at dir and its directory file at 8000.db.

Window 1

```
> voyager 8000 -j dir -f 8000.db  
voyager orb professional (directory server) 3.1, copyright objectspace 1997-1999
```

In a second window, create the serialized configuration object for DbService.

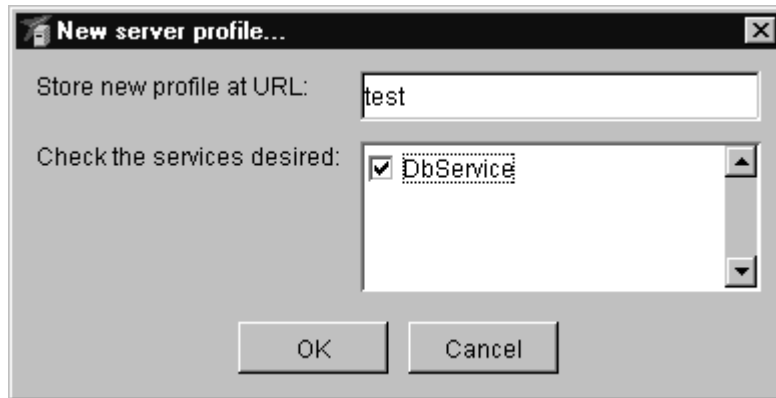
Window 2

```
> java examples.management.ConfigurationFactory dbsvc.ser  
>
```

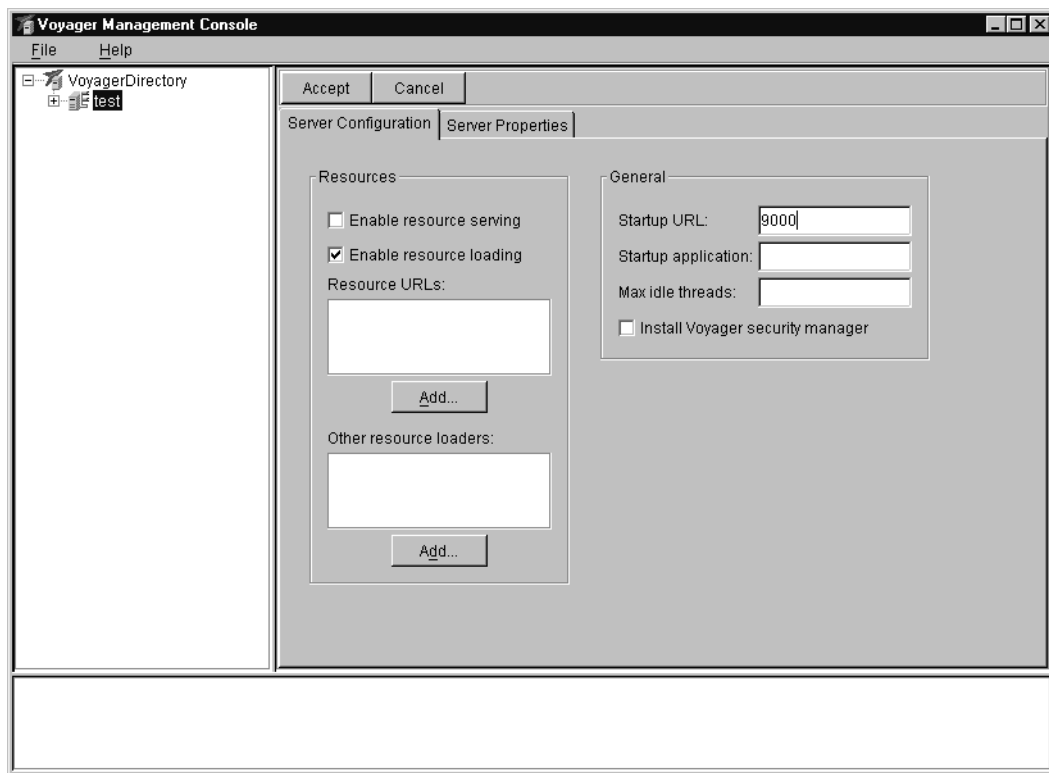
The configuration object is written to the file dbsvc.ser.

Start the Voyager Management Console, using the console utility, and log into the directory server by entering 8000/dir in the *URL* field and pressing the *OK* button. The console will inform you that the directory server has not been initialized. Press the *OK* button to clear this dialog. Next, from the File menu choose *Initialize directory...* Install the DbService into the directory by choosing Install service from the file menu and selecting dbsvc.ser.

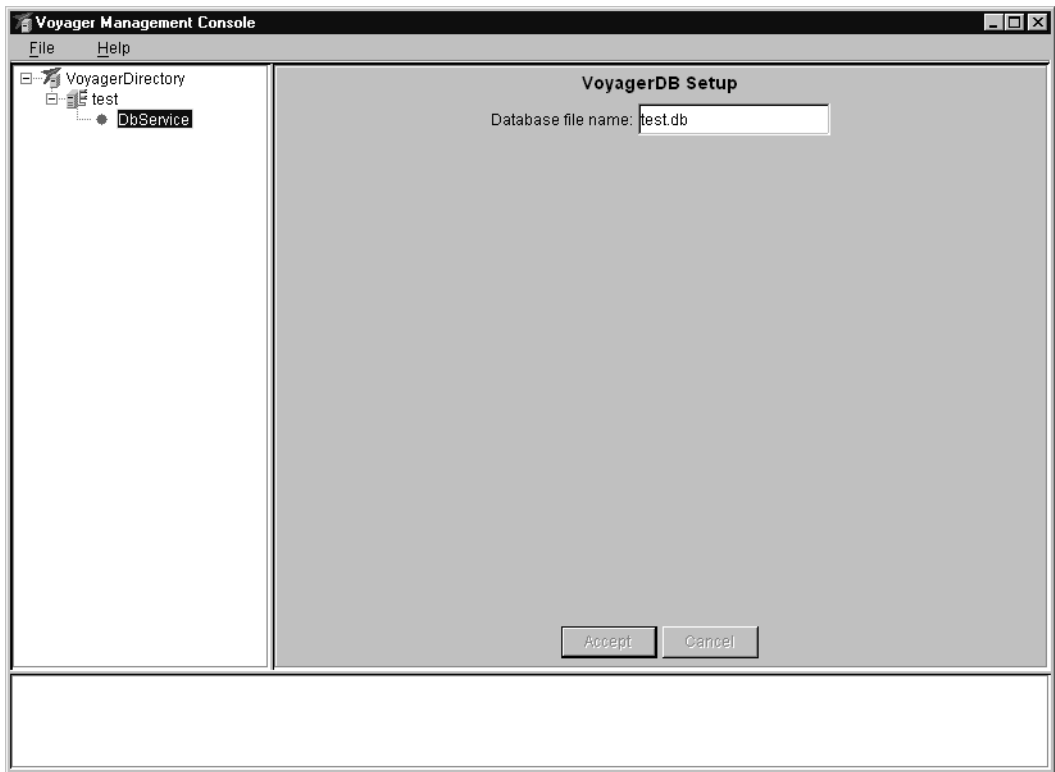
Create a new server profile by choosing New server profile... from the file menu. Enter test in the *URL* field, and check DbService in the *services list*. Press *OK* when finished.



A node called *test* appears in the *tree view* on the left. Highlight the *test* node. In the *Startup URL* field, replace 8000 with 9000, and press the *Accept* button.



Expand the test node, and highlight the DbService child node. Enter test.db in the Database name field.



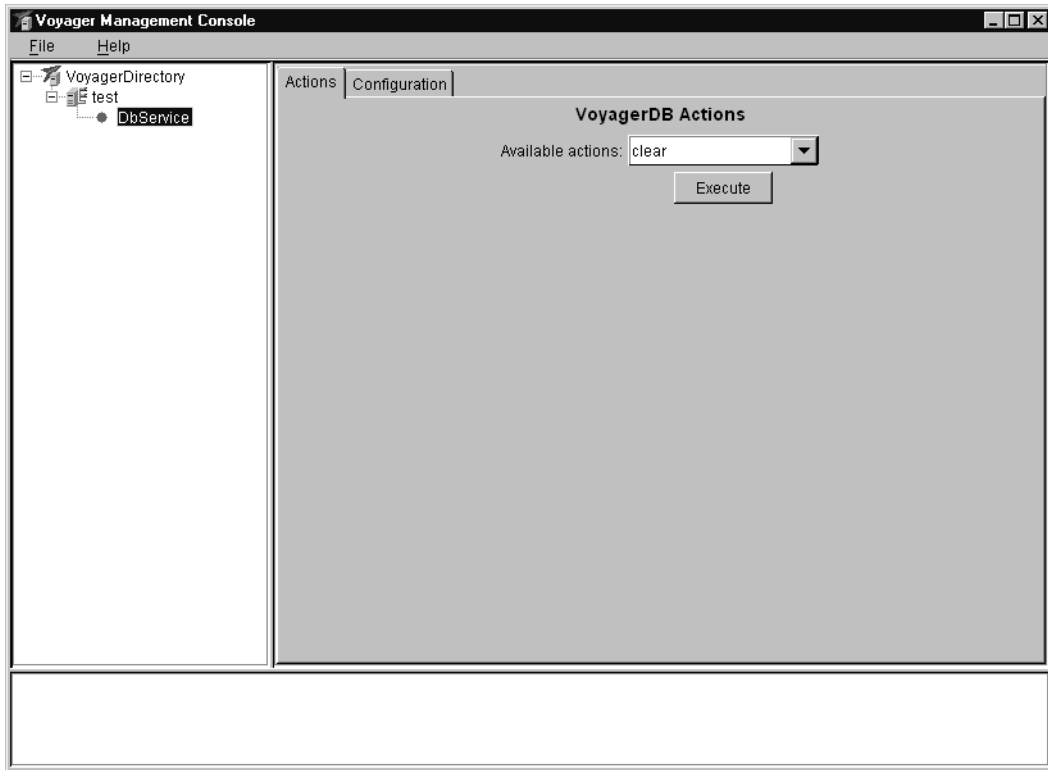
Shut down the console. Start a Voyager server, and associate it with the test entry in the directory, using the -d option.

Window 3

```
> voyager -d 8000/dir/test  
voyager orb professional 3.1, copyright objectspace 1997-1999  
DbService OPENED
```

Start the console again and log into the directory server the same as before. Expand the test node, and highlight the DbService child node. It now has two tabs, one for

configuration and one for actions. You are now interacting with the service via the management framework.



The source code for DbConfig.java, DbMonitor.java, DbService.java, and MonitorTool.java follows. DbConfig is an implementation of IConfiguration, DbMonitor is an implementation of IManagementAgent, DbService is the class being configured and managed, and MonitorTool is an implementation of ITool.

Class examples\management\DbConfig.java

```
// copyright 1999 objectspace

package examples.management;

import com.objectspace.voyager.management.*;
import com.objectspace.voyager.db.DatabaseException;
import com.objectspace.voyager.*;
```

```

import java.util.Properties;

public class DbConfig implements IConfiguration
{

    private String dbFileName = null;

    public DbConfig( String dbFileName )
    {
        setDbFileName( dbFileName );
    }

    // ACCESSING

    public void setDbFileName( String dbFileName )
    {
        this.dbFileName = dbFileName;
    }

    public String getDbFileName()
    {
        return this.dbFileName;
    }

    // ICONFIGURATION BEHAVIOR

    public void install( String url ) throws ManagementException
    {
        initialize();
        installAgent( url );
    }

    // SUPPORT

    void initialize()
    {
        try
        {
            DbService.initialize( getDbFileName() );
        }
    }
}

```

```

    }
    catch( DatabaseException e )
    {
        throw new DbServiceException( "could not initialize database: " + e );
    }
}

private void installAgent( String url )
{
    IManagementAgent agent = getAgent();
    DbService.setAgentInstaller( new Installer( url, agent ) );
}

private IManagementAgent getAgent()
{
    try
    {
        return DbMonitor.getMonitor();
    }
    catch( Exception e )
    {
        throw new DbServiceException( "could not create management agent" );
    }
}

}

```

Class examples\management\DbMonitor.java

// copyright 1999 objectspace

```
package examples.management;
```

```
import com.objectspace.voyager.management.*;
import com.objectspace.voyager.*;
import javax.naming.*;
import com.objectspace.voyager.db.DatabaseException;
import java.util.Properties;
```

```
public class DbMonitor extends ManagementAgent implements IConstants, IRemote
{

```

```

private static IManagementAgent soleInstance = null;

public static IManagementAgent getMonitor()
{
    try
    {
        if( soleInstance == null )
            soleInstance = (IManagementAgent) Factory.create( DbMonitor.class.getName() );
    }
    catch( Exception e )
    {
        throw new DbServiceException( "could not create management agent" );
    }

    return soleInstance;
}

/**
 * @return the managed objects
 */

public Object[] getManagedObjects()
{
    return new Object[] { Proxy.of( this ) };
}

/**
 * @return an array of strings identifying these actions
 */

public String[] getActions()
{
    return actions;
}

/**
 * @param action a string identifying the desired action
 */

public void performAction( String action ) throws ManagementException

```

```

{
    if( action.equals( CLEAR_ACTION ) == true )
        doClear();

    else if( action.equals( STARTUP_ACTION ) == true )
        restart();

    else if( action.equals( SHUTDOWN_ACTION ) == true )
        shutdown( SHUTDOWN_GRACEFULLY );

    else
        throw new ManagementException( "action \"" + action + "\" not supported" );
}

/**
 * @return an array of strings identifying the event types
 */

public String[] getEventTypes()
{
    return events;
}

/**
 * updates the configuration information for this service
 * @param url the location where this agent and its configuration are bound
 * @param configuration the new configuration object
 */

public void update( String url, IConfiguration configuration )
    throws ManagementException
{
    shutdown( SHUTDOWN_GRACEFULLY );

    if( configuration instanceof DbConfig )
        ( (DbConfig) configuration ).initialize();

    updateConfiguration( url, configuration );
    restart();
}

/**

```



```

* ask the service to restart
*/

public void restart()
{
    try
    {
        DbService.open();
        notify( STARTUP_EVENT, null );
    }
    catch( DatabaseException e )
    {
        notify( EXCEPTION_EVENT, new Object[] { e } );
    }
}

/**
* ask the service to shut down
* @param mode SHUTDOWN_GRACEFULLY, SHUTDOWN_IMMEDIATELY, or user value
*/

public void shutdown( int mode )
{
    try
    {
        DbService.close();
        notify( SHUTDOWN_EVENT, null );
    }
    catch( DatabaseException e )
    {
        notify( EXCEPTION_EVENT, new Object[] { e } );
    }
}

private void doClear()
{
    try
    {
        DbService.clear();
        notify( CLEAR_EVENT, null );
    }
    catch( DatabaseException e )

```

```

    {
        notify( EXCEPTION_EVENT, new Object[] { e } );
    }
}

private void updateConfiguration( String url, IConfiguration config )
{
    new Installer( url, config ).run();
}

}

```

Class examples\management\DbService.java

```

// copyright 1999 objectspace

package examples.management;

import com.objectspace.voyager.db.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.persistence.Persistence;
import com.objectspace.voyager.system.IService;
import java.io.PrintStream;

public class DbService implements IService
{

    private IVoyagerDb database = null;
    private static Runnable agentInstaller = null;
    private static DbService soleInstance = null;

    private DbService( String fileName ) throws DatabaseException
    {
        this.database = new VoyagerDb( fileName, false );
        Persistence.setDatabase( this.database );
    }

    // STATIC

    public static void initialize( String fileName ) throws DatabaseException
    {

```

```

try
{
    if( DbService.soleInstance != null )
        Voyager.deregisterService( DbService.soleInstance );

    DbService.soleInstance = new DbService( fileName );
    Voyager.registerService( DbService.soleInstance );
}
catch( DatabaseException e )
{
    throw new DbServiceException( "could not initialize database: " + e );
}
}

```

// ACCESSING

```

public static IVoyagerDb getDatabase() throws DatabaseException
{
    return getSoleInstance().getDb();
}

public static void setAgentInstaller( Runnable agentInstaller )
{
    DbService.agentInstaller = agentInstaller;
}

private static DbService getSoleInstance()
{
    if( DbService.soleInstance == null )
        throw new DbServiceException( "DbService has not been initialized" );

    return soleInstance;
}

private IVoyagerDb getDb()
{
    return this.database;
}

private static Runnable getAgentInstaller()
{

```

```
return DbService.agentInstaller;  
}
```

// EXPOSED VOYAGERDB BEHAVIOR

```
public static void open() throws DatabaseException  
{  
    getDatabase().open();  
    System.out.println( "DbService OPENED" );  
}
```

```
public static void close() throws DatabaseException  
{  
    getDatabase().close();  
    System.out.println( "DbService CLOSED" );  
}
```

```
public static void clear() throws DatabaseException  
{  
    getDatabase().clear();  
    System.out.println( "DbService CLEARED" );  
}
```

// ISERVICE BEHAVIOR

```
public void startup() throws StartupException  
{  
    if( getAgentInstaller() != null )  
        getAgentInstaller().run();  
}
```

```
try  
{  
    DbService.open();  
}  
catch( DatabaseException e )  
{  
    throw new StartupException( e );  
}  
}
```

```

public void shutdown()
{
    try
    {
        DbService.close();
    }
    catch( DatabaseException e )
    {
        throw new DbServiceException( "could not close database: " + e );
    }
}

public String getId()
{
    return "VoyagerDB";
}

public String[] getPrerequisites()
{
    return new String[0];
}

public void dumpStatus( PrintStream printStream )
{
    try
    {
        String status = ( getDatabase().isOpen() == true ) ? "open" : "closed";
        printStream.println( "DbService: VoyagerDB is " + status );
    }
    catch( Exception e )
    {
        printStream.println( "DbService: error printing status: " + e );
    }
}
}

```

Class examples\management\MonitorTool.java

// copyright 1999 objectspace

package examples.management;

```

import com.objectspace.workshop.explorer.ui.BasicUIPanel;
import com.objectspace.voyager.management.*;
import com.objectspace.voyager.*;
import com.objectspace.lib.ui.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class MonitorTool extends BasicUIPanel implements IEventListener, IConstants
{

    public static final String TOOL_NAME = "Actions";

    private JComboBox actions = null;
    private JLabel actionsLabel = null;
    private JButton execute = null;
    private JLabel title = null;

    // ITOOL BEHAVIOR

    public Image getIcon()
    {
        return null;
    }

    public boolean isDestroyable()
    {
        return true;
    }

    public boolean isStoppable()
    {
        return true;
    }

    public void init()
    {
        addAsEventListener( DbMonitor.EXCEPTION_EVENT );
        addAsEventListener( DbMonitor.STARTUP_EVENT );
        addAsEventListener( DbMonitor.SHUTDOWN_EVENT );
    }
}

```

```
addAsEventListener( DbMonitor.CLEAR_EVENT );
```

```
actions = new JComboBox();  
actions.setPreferredSize( FIELD_SIZE );  
actionsLabel = new JLabel( "Available actions:" );  
title = new JLabel( "VoyagerDB Actions" );  
title.setFont( new Font( "dialog", Font.BOLD, 14 ) );  
execute = new JButton( "Execute" );  
execute.addActionListener( getActionListener() );
```

```
GridBagHelper grid = new GridBagHelper( this );
```

```
grid.insets = grid.FULL_CUSHION;  
grid.gridwidth = 2;  
grid.add( title, 0, 0 );
```

```
grid.insets = grid.NORTH_CUSHION;  
grid.gridwidth = 1;  
grid.add( actionsLabel, 0, 1 );  
grid.add( actions, 1, 1 );
```

```
grid.insets = grid.NO_CUSHION;  
grid.add( execute, 1, 2 );  
grid.weighty = 1;
```

```
grid.add( new JPanel(), 0, 3 );
```

```
refreshValues();  
}
```

```
public void start()  
{  
    // no op  
}
```

```
public void stop()  
{  
    // no op  
}
```

```
public void destroy()  
{
```

```

removeAsEventListener( DbMonitor.EXCEPTION_EVENT );
removeAsEventListener( DbMonitor.STARTUP_EVENT );
removeAsEventListener( DbMonitor.SHUTDOWN_EVENT );
removeAsEventListener( DbMonitor.CLEAR_EVENT );
}

```

```

public String getName()
{
    return TOOL_NAME;
}

```

// SUPPORT

```

private void refreshValues()
{
    String[] available = getManagementAgent().getActions();

```

```

    if( actions.getItemCount() > 0 )
        actions.removeAllItems();

```

```

    for( int i = 0; i < available.length; i++ )
        actions.addItem( available[ i ] );

```

```

    if( available.length > 0 )
        actions.setSelectedIndex( 0 );

```

```

    validate();
}

```

```

private void execute()
{
    if( actions.getSelectedIndex() != -1 )
    {
        String selected = (String) actions.getSelectedItem();
        executeAction( selected );
    }
    else
        showNotSelected();
}

```

```

private void executeAction( String action )

```



```

{
try
{
    getManagementAgent().performAction( action );
}
catch( ManagementException e )
{
    handleException( e );
}
}

private void showNotSelected()
{
    JOptionPane.showMessageDialog(
        this,
        "No action is selected.\nPlease make a selection.",
        "No selection",
        JOptionPane.INFORMATION_MESSAGE );
}

private IManagementAgent getManagementAgent()
{
    return (IManagementAgent) getObject();
}

private void handleException( Exception e )
{
    getContext().showStatus( "MonitorTool exception: " + e );
}

```

// EVENTS

```

private ActionListener getActionListener()
{
    return new ActionListener()
    {
        public void actionPerformed( ActionEvent e )
        {
            if( e.getSource() == execute )
                execute();
        }
    }
}

```

```
};  
}
```

// IEVENTLISTENER BEHAVIOR

```
public void notify( String eventType, Object args[] )  
{  
    getContext().showStatus( "EVENT - " + eventType );  
}
```

// EVENTLISTENER HELPERS

```
private void addAsEventListener( String eventType )  
{  
    try  
    {  
        getManagementAgent().addEventListener( (IEventListener) Proxy.of( this ), eventType );  
    }  
    catch( ManagementException e )  
    {  
        handleException( e );  
    }  
}  
  
private void removeAsEventListener( String eventType )  
{  
    try  
    {  
        getManagementAgent().removeEventListener( (IEventListener) Proxy.of( this ), eventType );  
    }  
    catch( ManagementException e )  
    {  
        handleException( e );  
    }  
}  
  
}
```

Configuration



The examples in this section demonstrate user-customized configuration of Voyager properties.

Configuration1 Example

The Configuration1 example demonstrates configuration of a property in a Voyager server. The server is started on port 8000 with a properties file that changes the Console log level from SILENT to VERBOSE. The output to this server's console shows the modification.

Note: The displayed exception indicates that the Configuration1 program terminated. This information is reported to the Console in the form of the SocketException.

From examples\configuration, compile the example program:

```
javac IDisplay.java Display.java Configuration1.java
```

Start a Voyager server on port 8000 with the properties file in one window. Run Configuration1 in a second window.

Window1

```
>voyager 8000 -p configuration1.properties
voyager 3.1, copyright objectspace 1997-1999
tcp accepting connections on tcp://homer1:8000
[thread: TransportConnection(tcp://10.2.5.194:8000->tcp://10.2.5.194:1229)] DGC cycle START
[thread: Thread-2] will display because log level set to verbose
[thread: TransportConnection(tcp://10.2.5.194:8000->tcp://10.2.5.194:1229)]
java.net.SocketException: Connection reset by peer
    at java.net.SocketInputStream.read(Compiled Code)
    at java.io.BufferedInputStream.fill(Compiled Code)
    at java.io.BufferedInputStream.read(Compiled Code)
    at com.objectspace.voyager.vrmp.VrmpRequestHandler.readFully(Compiled Code)
    at com.objectspace.voyager.vrmp.VrmpRequestHandler.canProcess(Compiled Code)
    at com.objectspace.voyager.tcp.RequestManager.process(Compiled Code)
    at com.objectspace.voyager.tcp.RequestManager.process(Compiled Code)
    at com.objectspace.voyager.tcp.RequestManager$1.run(Compiled Code)
    at com.objectspace.lib.thread.ReusableThread.run(Compiled Code)
[thread: TransportConnection(tcp://10.2.5.194:8000->tcp://10.2.5.194:1229)] Closing connection
TransportConnection(tcp://10.2.5.194:8000->tcp://10.2.5.194:1229)
```

Window2

```
>java examples.configuration.Configuration1

>
```

The source code for configuration1.properties, IDisplay.java, Display.java, and Configuration1.java follows:

Properties examples\configuration\configuration1.properties

```
# This will set the Console log level to VERBOSE so that the
# display agent will successfully display its message
lib.util.Console.setLogLevel=verbose
```

Interface examples\configuration\IDisplay.java

// copyright 1997-1999 objectspace

package examples.configuration;

```
public interface IDisplay
{
    void display( String message );
}
```

Class examples\configuration\Display.java

// copyright 1997-1999 objectspace

package examples.configuration;

```
import java.io.Serializable;
import com.objectspace.lib.util.Console;
```

```
public class Display implements Serializable, IDisplay
{
    public Display()
    {
    }

    public void display( String message )
    {
        Console.log( message, Console.VERBOSE );
    }
}
```

Program examples\configuration\Configuration1.java

// copyright 1997-1999 objectspace

package examples.configuration;

```
import com.objectspace.voyager.Proxy;
import com.objectspace.voyager.Voyager;
import com.objectspace.voyager.agent.Agent;
```

```
public class Configuration1
{
}
```

```

public static void main( String[] args )
{
    try
    {
        Voyager.startup();
        IDisplay agent = (IDisplay) Proxy.of( new Display() );

        // won't display locally
        agent.display( "will not display since log level is silent by default" );

        // will display remotely
        Agent.of( agent ).moveTo( "//localhost:8000", "display", new Object[]{ "will display because log level set
to verbose" } );
        Voyager.shutdown();
    }
    catch( Exception ex )
    {
        ex.printStackTrace();
    }
}

```

Configuration2 Example

The Configuration2 example demonstrates configuration of multiple properties in a custom Voyager application. The thread pool's max idle thread count is set, as is the routing address. The router is configured to send all remote communication through the server on 8000. The output in this server's window shows confirmation of the router.

From examples\configuration, compile the example program:

```
javac Configuration2.java
```

Start a Voyager server on port 8000 with verbose logging in one window, and a Voyager server on port 7000 with verbose logging in a second window. Run Configuration2 in a third window.

Window1

```
>voyager 8000 -l verbose
voyager 3.1, copyright objectspace 1997-1999
tcp accepting connections on tcp://homer1:8000
[thread: TransportConnection(tcp://10.2.5.194:8000->tcp://10.2.5.194:1237)]
java.net.SocketException: Connection reset by peer
    at java.net.SocketInputStream.read(Compiled Code)
    at java.io.BufferedInputStream.fill(Compiled Code)
    at java.io.BufferedInputStream.read(Compiled Code)
    at com.objectspace.voyager.router.RouteRequestHandler.readFully(Compiled Code)
    at com.objectspace.voyager.router.RouteRequestHandler.canProcess(Compiled Code)
    at com.objectspace.voyager.tcp.RequestManager.process(Compiled Code)
    at com.objectspace.voyager.tcp.RequestManager.process(Compiled Code)
    at com.objectspace.voyager.tcp.RequestManager$1.run(Compiled Code)
    at com.objectspace.lib.thread.ReusableThread.run(Compiled Code)
[thread: TransportConnection(tcp://10.2.5.194:8000->tcp://10.2.5.194:1237)] Closing connection
TransportConnection(tcp://10.2.5.194:8000->tcp://10.2.5.194:1237)
```

Window2

```
>voyager 7000 -l verbose
voyager 3.1, copyright objectspace 1997-1999
tcp accepting connections on tcp://homer1:7000
[thread: TransportConnection(tcp://10.2.5.194:7000->tcp://10.2.5.194:1238)] DGC cycle START
[thread: TransportConnection(tcp://10.2.5.194:7000->tcp://10.2.5.194:1238)] routed message
```

Window3

```
>java examples.configuration.Configuration2
max idle threads set to: 10
router address set to: //homer1:8000

>
```

The source code for configuration2.properties and Configuration2.java follows:

Properties examples\configuration\configuration2.properties

```
# Change the max idle threads from Integer.MAX_VALUE to 10
voyager.ThreadManager.setMaxIdleThreads=10

# Route all communication through localhost:8000
voyager.router.Routing.setRouterAddress=//localhost:8000
```

Program examples\configuration\Configuration2.java

```
// copyright 1997-1999 objectspace

package examples.configuration;

import com.objectspace.lib.configuration.PropertyLoader;
import com.objectspace.voyager.Proxy;
import com.objectspace.voyager.ThreadManager;
import com.objectspace.voyager.Factory;
import com.objectspace.voyager.Voyager;
import com.objectspace.voyager.router.Routing;

public class Configuration2
{
    public static void main( String[] args )
    {
        try
        {
            // configure router and threadmanager
            new PropertyLoader( "configuration2.properties" ).load();
            Voyager.startup();
            System.out.println( "max idle threads set to: " + ThreadManager.getMaxIdleThreads() );
        }
    }
}
```



```

System.out.println( "router address set to: " + Routing.getRouterAddress() );

IDisplay d = (IDisplay) Factory.create( "examples.configuration.Display", "///localhost:7000" );
d.display( "routed message" );
Voyager.shutdown();
}
catch( Exception ex )
{
    ex.printStackTrace();
}
}
}

```

Configuration3 Example

The Configuration3 example demonstrates configuration of a *multi-property*, that is, a property that can take multiple values. The property demonstrated is the URL resource property. Setting these two properties, allows Voyager to load classes from the otherwise hidden folders `./hidden1` and `./hidden2`.

From `examples\configuration`, compile the example program:

```
javac Configuration3.java
```

From `examples\configuration\hidden1`, compile the first hidden class:

```
javac Hidden1.java
```

From `examples\configuration\hidden2`, compile the second hidden class:

```
javac Hidden2.java
```

Run Configuration3 from `examples\configuration`.

Window1

```

>java examples.configuration.Configuration3
found class Hidden1
found class Hidden2

>

```

The source code for configuration3.properties and Configuration3.java follows:

Properties examples\configuration\configuration3.properties

```
# Without these two lines, Voyager would not be able to find the
# classes Hidden1 and Hidden2. These two lines use relative
# syntax for specifying "file:" URLs.
voyager.loader.VoyagerClassLoader.addURLResource[1]=./hidden1/
voyager.loader.VoyagerClassLoader.addURLResource[2]=./hidden2/
```

Program examples\configuration\Configuration3.java

```
// copyright 1997-1999 objectspace

package examples.configuration;

import com.objectspace.lib.configuration.PropertyLoader;
import com.objectspace.voyager.ClassManager;
import com.objectspace.voyager.Voyager;

public class Configuration3
{
    public static void main( String[] args )
    {
        try
        {
            // add multiple resource loaders
            new PropertyLoader( "configuration3.properties" ).load();
            Voyager.startup();
            Class c = ClassManager.getClass( "Hidden1" );
            System.out.println( "found " + c );
            c = ClassManager.getClass( "Hidden2" );
            System.out.println( "found " + c );
            Voyager.shutdown();
        }
        catch( Exception ex )
        {
            ex.printStackTrace();
        }
    }
}
```

Universal Gateway



There are two Voyager servers: one on port 8000 and the other one on port 10000 (Voyager server on port 10000 must be started with `-r` option). The one on port 8000 has a CORBA object bound to `TravelEngine` in the `CorbaDirectory` directory. This Voyager server is started in `CorbaServer.class`. In order to get a proxy to a CORBA reference, the actual object must be remote to the server where the proxy is created.

To create this environment, the lookup call is sent to Voyager on port 10000: `"rmi://:10000/"`. This is an RMI call since there is an `rmi:` prefix; thus the returned object will be a proxy to an RMI object. The server will look up the actual object on port 10000, and it will use the remaining portion of the name: `cos://localhost:8000/CorbaDirectory/TravelEngine`. This routes to the Voyager server on port 8000 and returns a proxy to a CORBA object. (see `Client.java`)

From `examples/universalgateway`, compile the universal gateway example files:

```
javac *.java
```

Window1

```
>voyager 10000 -r
```

This starts a Voyager server on port 10000 with a resource loader enabled:

Window2

```
>java CorbaServer
```

Window2

```
>java Client
```

The source code for Client.java, CorbaServer.java, Flight.java, IFlight.java, PassengerInfo.java, IPassengerInfo.java, TravelEngine.java, ITravelEngine.java, and ITravelEngine.idl follows:

Program examples\universalgateway\Client.java

```
import java.rmi.*;

public class Client
{
    public static void main( String[] args )
    {
        try
        {
            System.setSecurityManager( new RMISecurityManager() );

            // This is a bit tricky:
            // There are two Voyager servers: one on port 8000 and the other one on port 10000 (Voyager server
            // on port 10000 must be started with -r option).
            // The one on port 8000 has a CORBA object bound to "TravelEngine" name in "CorbaDirectory"
            directory.
            // This Voyager server is started in CorbaServer.class. In order to get a proxy to a CORBA reference,
            // the actual objet must be remote to the server, where the proxy is created. To create this environment,
            // the lookup call is sent to Voyager on port 10000: "rmi://:10000/". This is an RMI call, since there
            // is rmi: prefix, thus the returned object will be a proxy to an RMI object. The actual object is going
            // to be lookup by the server on port 10000, and it will use the remaining portion of the
            // name: "cos://localhost:8000/CorbaDirectory/TravelEngine" which routes to the Voyager server on
            // port 8000 and it will return a proxy to a CORBA object.
            // AS A RESULT THE FOLLOWING lookup() CALL WILL RETURN AN RMI PROXY TO A CORBA
            OBJECT
            ITravelEngine travelEngine = ( ITravelEngine )Naming.lookup(
            "rmi://:10000/cos://localhost:8000/CorbaDirectory/TravelEngine" );

            // Get the flight availability for Dallas - London
```

```

System.out.println( "Getting flight availability for Dallas - London" );
IFlight[] flights = travelEngine.getFlightAvailability( "DFW", "LON" );

int bestFare = 0;
int availabilityOption = 0;

System.out.println( "Got flights back:" );

for( int i =0 ; i < flights.length; i++ )
{
    System.out.println( flights[i].getCityFrom() + " " +
        flights[i].getCityTo() + " " +
        flights[i].getDepartureDate() + " " +
        flights[i].getDepartureTime() + " " +
        flights[i].getDepartureDate() + " " +
        flights[i].getDepartureTime());

    int fare = travelEngine.getPrice( flights[i] );

    if( bestFare == 0 )
    {
        bestFare = fare;
        availabilityOption = i;
    }
    else
    {
        if( bestFare > fare )
        {
            bestFare = fare;
            availabilityOption = i;
        }
    }

    System.out.println( "Fare quote - $" + fare );
}

System.out.println( "\n\nBest fare - $" + bestFare );
System.out.println( "\nConfirming reservation..." );

IPassengerInfo passenger = travelEngine.createPassenger( "Joe", "Traveler", "100 Main St, Dallas, TX
75001", 456789012, 23232 );

```

```

        System.out.println( "Reservation number - " + new String( travelEngine.bookFlight( flights[
availabilityOption ], passenger ) ) );
    }
    catch( Exception e )
    {
        e.printStackTrace();
    }
}
}

```

Program examples\universalgateway\CorbaServer.java

```

import java.io.*;
import com.objectspace.voyager.*;
import com.objectspace.voyager.corba.*;
import com.objectspace.voyager.directory.*;

public class CorbaServer
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup( "8000" );

            ClassManager.enableResourceServer();

            ITravelEngine travelEngine = new TravelEngine();

            String ior = Corba.asIOR( travelEngine );

            Object corbaobject = Namespace.lookup( ior );

            Namespace.bind( "CorbaDirectory/TravelEngine", corbaobject );

            System.out.println( "Server is ready" );
        }
        catch( Exception e )
        {

```

```

        e.printStackTrace();
    }
}
}

```

Program examples\universalgateway\Flight.java

```

public class Flight implements IFlight
{
    String flightNumber, cityFrom, cityTo, depDate, depTime, arrDate, arrTime;

    public Flight( )
    {
    }

    public Flight( String flightNumber,
        String cityFrom,
        String cityTo,
        String depDate,
        String depTime,
        String arrDate,
        String arrTime )
    {
        this.flightNumber = flightNumber;
        this.cityFrom = cityFrom;
        this.cityTo = cityTo;
        this.depDate = depDate;
        this.depTime = depTime;
        this.arrDate = arrDate;
        this.arrTime = arrTime;
    }

    protected void finalize()
    {
        System.out.println( "Object is being finalized" );
    }

    public java.lang.String getFlightNumber()
    {
        return flightNumber;
    }
}

```

```

public java.lang.String getCityFrom()
{
    return cityFrom;
}

public java.lang.String getCityTo()
{
    return cityTo;
}

public java.lang.String getDepartureDate()
{
    return depDate;
}

public java.lang.String getDepartureTime()
{
    return depTime;
}

public java.lang.String getArrivalDate()
{
    return arrDate;
}

public java.lang.String getArrivalTime()
{
    return arrTime;
}
}

```

Program examples\universalgateway\IFlight.java

```

/**
 * IFlight.java
 * <p>
 * @version 1.0
 * @author generated by cgen 3.0b2 at Wed Mar 24 19:20:41 CST 1999
 */

public interface IFlight extends com.objectspace.voyager.IRemote
{

```



```

public java.lang.String getFlightNumber();
public java.lang.String getCityFrom();
public java.lang.String getCityTo();
public java.lang.String getDepartureDate();
public java.lang.String getDepartureTime();
public java.lang.String getArrivalDate();
public java.lang.String getArrivalTime();
public static com.objectspace.voyager.corba.ObjectCode _TYPECODE = new
com.objectspace.voyager.corba.ObjectCode( "IFlight", "IDL:IFlight:1.0", "IFlight" ); // added by cgen
}

```

Program examples\universalgateway\PassengerInfo.java

```

public class PassengerInfo implements IPassengerInfo, java.io.Serializable
{
    String firstName;
    String lastName;
    String address;
    long creditCardNumber;
    int expDate;

    public PassengerInfo( String firstName,
                          String lastName,
                          String address,
                          long creditCardNumber,
                          int expDate )
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.address = address;
        this.creditCardNumber = creditCardNumber;
        this.expDate = expDate;
    }

    public java.lang.String getFirstName()
    {
        return firstName;
    }

    public java.lang.String getLastName()
    {
        return lastName;
    }
}

```

```

    }

    public java.lang.String getAddress()
    {
        return address;
    }

    public long getCreditCardNumber()
    {
        return creditCardNumber;
    }

    public int getExpirationDate()
    {
        return expDate;
    }
}

```

Program examples\universalgateway\IPassengerInfo.java

```

/**
 * IPassengerInfo.java
 * <p>
 * @version 1.0
 * @author generated by cgen 3.0b2 at Wed Mar 24 19:20:41 CST 1999
 */

public interface IPassengerInfo extends com.objectspace.voyager.IRemote
{
    public java.lang.String getFirstName();
    public java.lang.String getLastName();
    public java.lang.String getAddress();
    public long getCreditCardNumber();
    public int getExpirationDate();
    public static com.objectspace.voyager.corba.ObjectCode _TYPECODE = new
com.objectspace.voyager.corba.ObjectCode( "IPassengerInfo", "IDL:IPassengerInfo:1.0", "IPassengerInfo"
); // added by cgen
}

```

Program examples\universalgateway\TravelEngine.java

```
import java.util.*;

public class TravelEngine implements ITravelEngine
{
    static Random random = new Random( Calendar.getInstance().getTime().getTime() );

    static String[] months =
    {
        "",
        "JAN",
        "FEB",
        "MAR",
        "APR",
        "MAY",
        "JUN",
        "JUL",
        "AUG",
        "SEP",
        "OCT",
        "NOV",
        "DEC"
    };

    public IFlight[] getFlightAvailability( java.lang.String cityFrom, java.lang.String cityTo )
    {
        IFlight[] flights = new IFlight[5];

        Calendar cal = Calendar.getInstance();

        for( int i=0; i < flights.length; i++ )
        {
            flights[i] = new Flight( "00" + i,
                                     cityFrom,
                                     cityTo,
                                     "" + cal.get( Calendar.DATE ) + months[ cal.get( Calendar.MONTH ) ],
                                     ( "" + cal.get( Calendar.HOUR_OF_DAY ) ) + ( ":" + cal.get( Calendar.MINUTE ) ),
                                     "" + cal.get( Calendar.DATE ) + months[ cal.get( Calendar.MONTH ) ],
                                     ( "" + cal.get( Calendar.HOUR_OF_DAY ) ) + ( ":" + cal.get( Calendar.MINUTE ) ) );
        }
    }
}
```

```

        return flights;
    }

    public int getPrice( IFlight flight )
    {
        return Math.abs( random.nextInt() ) / 1000;
    }

    public char[] bookFlight( IFlight flight, IPassengerInfo passenger )
    {
        System.out.println( "Performing booking for:\n" + passenger.getFirstName() + " " +
            passenger.getLastName() );
        System.out.println( "Flight details:\n\tFROM: " + flight.getCityFrom() +
            "\n\tTO: " + flight.getCityTo() +
            "\n\tDEPARTURE DATE: " + flight.getDepartureDate() +
            "\n\tDEPARTURE TIME: " + flight.getDepartureTime() +
            "\n\tARRIVAL DATE: " + flight.getDepartureDate() +
            "\n\tARRIVAL TIME: " + flight.getDepartureTime());

        char[] resNumber = new char[6];

        for( int i=0 ; i < resNumber.length; i++ )
            resNumber[i] = ( char )random.nextInt();

        return resNumber;
    }

    public IPassengerInfo createPassenger( String firstname, String lastname, String address, long
        creditcard, int expDate )
    {
        return new PassengerInfo( firstname, lastname, address, creditcard, expDate ) ;
    }
}

```

Program examples\universalgateway\ITravelEngine.java

```

/**
 * ITravelEngine.java
 * <p>

```

```

* @version 1.0
* @author generated by cgen 3.0b2 at Wed Mar 24 19:20:41 CST 1999
*/

public interface ITravelEngine extends com.objectspace.voyager.IRemote
{
    public IFlight[] getFlightAvailability( java.lang.String cityFrom, java.lang.String cityTo );
    public int getPrice( IFlight flight );
    public IPassengerInfo createPassenger( java.lang.String firstname, java.lang.String lastname,
java.lang.String address, long creditcard, int expDate );
    public char[] bookFlight( IFlight flight, IPassengerInfo passenger );
    public static com.objectspace.voyager.corba.ObjectCode _TYPECODE = new
com.objectspace.voyager.corba.ObjectCode( "ITravelEngine", "IDL:ITravelEngine:1.0", "ITravelEngine" ); //
added by cgen
}

```

Program examples\universalgateway\ITravelEngine.idl

```

interface IFlight
{
    string getFlightNumber();

    string getCityFrom();
    string getCityTo();

    string getDepartureDate();
    string getDepartureTime();

    string getArrivalDate();
    string getArrivalTime();
};

interface IPassengerInfo
{
    string getFirstName();
    string getLastName();

    string getAddress();

    long long getCreditCardNumber();
}

```

```

    long getExpirationDate();
};

interface ITravelEngine
{
    typedef sequence <IFlight> FlightList;
    typedef sequence <char,6> ConfirmationNumber;

    FlightList getFlightAvailability( in string cityFrom, in string cityTo );

    long getPrice( in IFlight flight );

    IPassengerInfo createPassenger( in string firstname, in string lastname, in string address, in long long
    creditcard, in long expDate);

    ConfirmationNumber bookFlight( in IFlight flight, in IPassengerInfo passenger );

};

```

Dynamic XML

The examples in this section demonstrate Dynamic XML usage. The first example shows how to access the content of an XML document based on a very simple DTD. The second example uses more complex structures to show how to handle compound elements, arrays, and API. The third example illustrates the same processes as the first two examples but uses more complex structures.

Example 1 - File System1

Files for this example are located in `examples\xml`

This example shows you how to:

- ◆ write a sample DTD file
- ◆ generate interfaces for the DTD using XGEN

- ◆ write a sample XML file based on the created DTD
- ◆ write a Java client accessing and displaying the content of the XML file

The DTD file describes a simple directory structure containing files. For simplicity, assume that a directory cannot contain other directories.

Create a file `filesystem1.dtd` containing:

```
<!ELEMENT Dir (Name,File*)>
<!ELEMENT File (Name, Date, Size)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT Size (#PCDATA)>
```

You can now generate the interfaces for this DTD using XGEN. Before you run the utility, make sure the directory where XGEN will store the output is accessible in your CLASSPATH. For example, if the directory from which you start XGEN is `\project\files`, then this directory must be included into the CLASSPATH environment variable. A universal alternative is to include `."`(dot).

Run XGEN on `filesystem1.dtd`:

```
xgen filesystem1.dtd
```

The interfaces are generated in a subdirectory named after the name of the DTD file (`filesystem1` in this case). The directory contains not only the source files of the interfaces but the compiled code as well.

The next step is to create an XML document based on the `filesystem1.dtd` file.

Create a file `filesystem1.xml` containing:

```
<?xml version="1.0"?>
<!DOCTYPE Dir SYSTEM "filesystem1.dtd">

<Dir>
  <Name>c:\</Name>
  <File>
    <Name>Autoexec.bat</Name>
    <Date>Friday, October 02, 1998 7:16:48 PM</Date>
    <Size>1024</Size>
  </File>
  <File>
```

```
<Name>Config.sys</Name>
<Date>Friday, October 02, 1998 7:07:29 PM</Date>
<Size>108,421,120</Size>
</File>
</Dir>
```

This file describes a directory called c:\ containing 2 files: Autoexec.bat and Config.sys.

The ultimate purpose of this example is to write a program that will access the XML file and display its content.

From examples\xml, compile the examples programs.

```
javac examples\xml\Filesystem1.java
```

Before running the client, check that you can see these packages from CLASSPATH:

- ◆ examples.xml: - the client
- ◆ filesystem1: - generated interfaces

Run the client:

```
>java examples.xml.Filesystem1
c:\
Autoexec.bat 1024 Friday, October 02, 1998 7:16:48 PM
Config.sys 108,421,120 Friday, October 02, 1998 7:07:29 PM
```

The source for Filesystem1.java is listed below.

```
Class examples\xml\Filesystem1.java
// (c) Copyright 1997-1999 ObjectSpace Inc.
```

```
package examples.xml;
```

```
import java.util.*;
import java.io.*;
import com.objectspace.xml.*;
import com.objectspace.voyager.*;
import filesystem1.*;
```

```
public class Filesystem1
{
    public static void main( String[] args )
    {
```



```

try
{
    Voyager.startup( "8000" );
    XmlDocument xmlDocument = Xml.openDocument( new File( "filesystem1.xml" ) );
    IDir root = (IDir) xmlDocument.getRoot();
    System.out.println( root.getName() );

    for( Enumeration e = root.getFileElements(); e.hasMoreElements(); )
    {
        IFile file = (IFile) e.nextElement();
        System.out.println( " " + file.getName() + " " + file.getSize() + " " + file.getDate());
    }

    Voyager.shutdown();
}
catch( Exception e )
{
    e.printStackTrace();
}

System.exit( 0 );
}

```

Example 2 - File system2

This example uses again the file system's directory and files to show how nested elements can be replaced by attributes.

The directory declares a collection of files and a name attribute that is required. This attribute is declared as ID to be able to reference it from a file.

A file is now an empty element, declaring a set of attributes:

- ◆ name: required ID
- ◆ date and size: required strings
- ◆ access is one of: read and read/write
- ◆ compressed: implied string
- ◆ os: string defaulted to "NT"
- ◆ parent: required IDREF

In addition both directory and file have a fixed attribute type whose value is respectively dir and file.

The purpose of this example is to show how to access and modify the attributes of two different files.

First create a DTD called filesystem2.dtd:

```
<!ELEMENT Dir (File*)>
<!ATTLIST Dir
  Name ID #REQUIRED
  Type CDATA #FIXED "Dir">

<!ELEMENT File EMPTY>
<!ATTLIST File
  Name ID #REQUIRED
  Type CDATA #FIXED "File"
  Date CDATA #REQUIRED
  Size CDATA #REQUIRED
  Access (r|rw) "rw"
  Compressed CDATA #IMPLIED
  OS CDATA "NT"
  Parent IDREF #REQUIRED>
```

Compile this DTD using xgen:

```
>xgen filesystem2.dtd
xgen 3.1, copyright objectspace 1997-1999

processing filesystem2.dtd...
Generating filesystem2\Dir.java ...
Generating filesystem2\File.java ...
Generating filesystem2\Entities.java ...
Generating filesystem2\DocumentTypeInfo.java ...
```

Make sure that the generated interfaces are accessible from your CLASSPATH, and compile Filesystem2.java:

```
>javac Filesystem2.java
```

Create a new file, filesystem2.xml:

```
<?xml version="1.0"?>
```

```
<!DOCTYPE Dir SYSTEM "filesystem2.dtd">
```

```
<Dir Name="c:">  
  <File  
    Name="autoexec.bat"  
    Date="4/4/99"  
    Size="154"  
    Compressed="winzip"  
    Parent="c:"/>  
  <File  
    Name="config.sys"  
    Date="4/4/99"  
    Size="230"  
    Access="r"  
    OS="Unix"  
    Parent="c:"/>  
</Dir>
```

Run Filesystem2:

```
>java examples.xml.Filesystem2  
type=dir  
name=c:
```

```
type=file  
os=NT  
access=rw  
date=4/4/99  
name=autoexec.bat  
compressed=winzip  
size=154  
parent=c:  
com.objectspace.voyager.xml.FixedAttributeException: Unable to set fixed attribute: type  
java.util.NoSuchElementException: No such attribute: undeclared  
com.objectspace.voyager.xml.RequiredAttributeException: Unable to remove required attribute: name  
os removed: true  
os removed: false  
compressed removed: true  
java.util.NoSuchElementException: No such attribute: undeclared  
  
type=file  
os=Unix  
access=r
```

```
date=4/4/99
name=config.sys
size=230
parent=c:
type removed: true
type removed: false
writing filesystem2.xml.out...
done
```

The file modified in the example is saved as filesystem2.xml.out:

```
<?xml version="1.0"?>
<!DOCTYPE Dir SYSTEM "filesystem2.dtd" >
<Dir Name="c:" Type="Dir">
  <File Name="autoexec.001" Date="4/4/99" Size="154" Parent="c:" Access="rw" Type="File"/>
  <File Name="config.sys" Date="4/4/99" Size="230" Access="r" OS="Unix" Parent="c:"
Compressed="winzip"/>
</Dir>
```

Class examples\xml\FileSystem2.java

// (c) Copyright 1997-1999 ObjectSpace Inc.

```
package examples.xml;

import java.util.*;
import java.io.*;
import com.objectspace.xml.*;
import com.objectspace.voyager.*;
import filesystem2.*;

public class FileSystem2
{
    static int indent = 0;

    public static void main( String[] args )
    {
        try
        {
            Voyager.startup( "8000" );
            IXml xmlDocument = Xml.openDocument( new File( "filesystem2.xml" ) );
            IDir dir = (IDir) xmlDocument.getRoot();
```

```

printAttributes( dir.getAttributes() );
IFile[] files = dir.GetFiles();
indent++;

for( int i = 0; i < files.length; i++ )
{
    indentPrint( "" );
    IFile file = files[i];
    Hashtable attributes = file.getAttributes();
    printAttributes( attributes );
    String name = (String) attributes.get( "Name" );

    if( "autoexec.bat".equals( name ) )
        changeAttributesForAutoexec( file );
    else if( "config.sys".equals( name ) )
        changeAttributesForConfig( file );
}

indent--;
indentPrint( "writing filesystem2.xml.out..." );
xmlDocument.saveDocument( new File( "filesystem2.xml.out" ) );
indentPrint( "done" );
Voyager.shutdown();
}
catch( Exception e )
{
    e.printStackTrace();
}

System.exit( 0 );
}

static void changeAttributesForAutoexec( IFile file )
{
    // set required
    file.setAttribute( "Name", "autoexec.001" );

    // set fixed with default value
    file.setAttribute( "Type", "File" );

    // set fixed with invalid value
    try

```

```

    {
        file.setAttribute( "Type", "invalid" );
    }
catch( FixedAttributeException e )
    {
        indentPrint( e.toString() );
    }

// set attribute not implied, fixed or required
file.setAttribute( "OS", "OS/2" );

// set undeclared attribute
try
    {
        file.setAttribute( "undeclared", "invalid" );
    }
catch( NoSuchElementException e )
    {
        indentPrint( e.toString() );
    }

// attempt to remove a required attribute
try
    {
        file.removeAttribute( "Name" );
    }
catch( RequiredAttributeException e )
    {
        indentPrint( e.toString() );
    }

// remove implied attribute
indentPrint( "os removed: " + file.removeAttribute( "OS" ) );

// remove implied attribute
indentPrint( "os removed: " + file.removeAttribute( "OS" ) );

// remove attribute not declared as one of: implied, required, fixed
indentPrint( "compressed removed: " + file.removeAttribute( "Compressed" ) );

// attempt to remove an undeclared attribute
try

```

```

    {
        file.removeAttribute( "undeclared" );
    }
    catch( NoSuchElementException e )
    {
        indentPrint( e.toString() );
    }
}

static void changeAttributesForConfig( IFile file )
{
    // remove fixed attribute
    indentPrint( "type removed: " + file.removeAttribute( "Type" ) );

    // remove fixed attribute
    indentPrint( "type removed: " + file.removeAttribute( "Type" ) );

    // set implied attribute
    file.setAttribute( "Compressed", "winzip" );
}

static void indentPrint( String output )
{
    {
        for( int i = 0; i < indent; i++ )
            System.out.print( " " );

        System.out.println( output );
    }
}

static void printAttributes( Hashtable attributes )
{
    {
        for( Enumeration e = attributes.keys(); e.hasMoreElements(); )
        {
            Object key = e.nextElement();
            Object value = attributes.get( key );
            indentPrint( key+"="+value );
        }
    }
}

```

Example 3 - Person

Files for this example are located in examples\xml

This example will go through the same process used in the first two examples, but this example will use more complex structures. The DTD document for this example describes a person that has a name and a list of e-mail addresses. An e-mail address is a string, whereas the name is a structure containing multiple instances of these elements:

- ◆ a nickname and a real name
- ◆ a family name
- ◆ a given name

We assume that a person can have multiple family names, given names, or nicknames and real names.

The purpose of this example is to:

- ◆ Create a DTD file describing a person and generate the interfaces for the DTD.
- ◆ Create an XML file describing a specific person.
- ◆ Create a client program that will access the XML file, modify its content, and save as another file for comparison with the original file.

First, create a DTD file using the following source:

```
<!ELEMENT Person (Name, Email*) >
<!ELEMENT Family (#PCDATA) >
<!ELEMENT Nick (#PCDATA) >
<!ELEMENT Real (#PCDATA) >
<!ELEMENT Given (#PCDATA) >
<!ELEMENT Name ( (Nick,Real) | Family | Given )+ >
<!ELEMENT Email (#PCDATA) >
```

Save the file as person.dtd and generate the interfaces by running xgen on person.dtd:

Note: XGEN represents each substructure as a different interface.

(Nick,Real) is represented by INickAndReal

((Nick,Real),Family,Given) is represented by IFamilyOrNickAndRealOrGiven

The subdirectory person should now contain the sources for the generated interfaces and the compiled files.

Write an XML file using this code and save as person.xml:

```
<?xml version="1.0"?>
<!DOCTYPE Person SYSTEM "person.dtd">

<Person>
  <Name>
    <Nick>Al</Nick>
    <Real>Albert</Real>
    <Family>Brown</Family>
  </Name>
  <Email>abrown@hotmail.com</Email>
  <Email>alber_brown@aardvark.com</Email>
</Person>
```

The XML file describes a single person whose nickname is Al and real name is Albert. This person has a family name Brown but no given name, and 2 e-mail addresses.

The client program we are going to write will perform the following actions:

- ◆ remove the nickname and the real name
- ◆ create a new nickname and real name, and insert it as the first element of the name structure
- ◆ iterate over the collection of elements making a name to find a family name Brown
- ◆ set this element to be a given name whose value is Black
- ◆ create a new collection of e-mails containing a single address and replace the existing collection
- ◆ save the modified document in a different file called person.xml.out

The goal is to obtain a person.xml.out file with the following content:

```
<Person>
  <Name>
    <Nick>jeff</Nick>
    <Real>Jeffrey</Real>
    <Given>Black</Given>
```

```
</Name>
  <Email>jblack@stratfort.com</Email>
</Person>
```

Write the client program following the next several steps:

Get the proxy pointing to the name element in the XML document:

```
IXml xmlDocument = Xml.openDocument( new File( "person.xml" ) );
IPerson root = (IPerson) xmlDocument.getRoot();
IName name = root.getName();
```

Remove the first element from the name (Al/Albert):

```
name.removeFamilyOrNickAndRealOrGivenAt( 0 );
```

Create a compound name and set the nickname and the real name to be: jeff/Jeffrey:

```
// create an empty compound name
IFamilyOrNickAndRealOrGiven compoundName = name.newFamilyOrNickAndRealOrGiven();
```

```
// create an empty nickname and real name
INickAndReal nickReal = compoundName.newNickAndReal();
```

```
// set the nickname and the real name
nickReal.setNick( "jeff" );
nickReal.setReal( "Jeffrey" );
```

```
// associate the nickname and the real name with the compound name
compoundName.setNickAndReal( nickReal );
```

```
// associate the compound name with the main name
name.insertFamilyOrNickAndRealOrGivenAt( compoundName, 0 );
```

Look for a compound name whose family name is Brown:

Note: If you knew that we were looking for the second element of name, we could simply use `name.getFamilyOrNickAndRealOrGivenAt(1)`. Assume we do not know where the family name is.

```
IFamilyOrNickAndRealOrGiven brown = null;
for( Enumeration e = name.getFamilyOrNickAndRealOrGivenElements(); e.hasMoreElements(); )
{
    brown = (IFamilyOrNickAndRealOrGiven) e.nextElement();
    String familyName = brown.getFamily();
}
```

```
if( familyName != null && familyName.equals( "Brown" ) )  
break;  
}
```

Note: Using `getFamilyOrNickAndRealOrGivens()` would have returned an array of `IFamilyOrNickAndRealOrGiven`. Doing the search on the array would have made the code simpler by removing the need for a cast.

Now that we have the compound element containing family name Brown, we can simply make it a given name Black:

```
brown.setGiven( "Black" );
```

Note: Because in the DTD “|” is an exclusive OR (XOR), setting the given name automatically removes the family name, thus ensuring the consistency of your XML document.

The second step of our program is to build an array of e-mails containing a single address,:

```
String[] emails = new String[] { "jblack@stratfort.com" };
```

and set it as the list of addresses of our person now called Jeffrey:

```
root.setEmails( emails );
```

The last step of our program is to save the modified document under `person.xml.out`. To save the document, we need to use an instance of `IXml` returned by `Xml.openDocument()`:

```
xmlDocument.saveDocument( new File( filename+".out" ) );
```

The complete source of the program is provided at the end of the section.

Finally execute this class and check the content of `person.xml.out`:

```
Class \examples\xml\Person.java  
// (c) Copyright 1997-1999 ObjectSpace Inc.
```

```
package examples.xml;
```

```
import java.io.File;  
import java.util.Enumeration;
```

```

import com.objectspace.xml.*;
import com.objectspace.voyager.*;
import person.*;

public class Person
{
    public static void main( String[] args )
    {
        try
        {
            Voyager.startup( "8000" );
            System.out.println( "Reading XML document - person.xml..." );

            IXml xmlDocument = Xml.openDocument( new File( "person.xml" ) );
            IPerson root = (IPerson) xmlDocument.getRoot();
            IName name = root.getName();

            System.out.println( "Performing document modifications..." );

            /*****
             * modification of the name
             *****/
            //remove nick/real name: al/albert
            name.removeFamilyOrNickAndRealOrGivenAt( 0 );

            // create a new nick/real name: jeff/jeffrey
            IFamilyOrNickAndRealOrGiven compoundName = name.newFamilyOrNickAndRealOrGiven();
            INickAndReal nickReal = compoundName.newNickAndReal();
            nickReal.setNick( "jeff" );
            nickReal.setReal( "Jeffrey" );
            compoundName.setNickAndReal( nickReal );
            name.insertFamilyOrNickAndRealOrGivenAt( compoundName, 0 );

            // look for family name "Brown"
            IFamilyOrNickAndRealOrGiven brown = null;

            for( Enumeration e = name.getFamilyOrNickAndRealOrGivenElements(); e.hasMoreElements(); )
            {
                brown = (IFamilyOrNickAndRealOrGiven) e.nextElement();
                String familyName = brown.getFamily();

                if( familyName != null && familyName.equals( "Brown" ) )

```

```

        break;
    }

    // replace family name "Brown" by given name "Black"
    // note: setting the given name, automatically nils-out the family name (exclusive OR)
    if( brown != null )
        brown.setGiven( "Black" );

    /*****
    * modification of emails
    *****/
    // create an array of emails
    String[] emails = new String[] { "jblack@stratfort.com" };

    // set the new array of emails
    root.setEmails( emails );

    // save the document
    System.out.println( "Saving XML document - person.xml.out" );
    xmlDocument.saveDocument( new File( "person.xml.out" ) );
    Voyager.shutdown();
    }
    catch( Exception e )
    {
        e.printStackTrace();
    }
    }
}

```

