OBJECTSPACE
VOYAGER

ORB 3.0
Developer Guide

OBJECTSPACE

Software Version   3.0

First 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.

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)
```

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

Overview ⓵

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, and, shortly, DCOM. Its innovative dynamic proxy generation removes the need for stub generators. Voyager ORB includes a universal naming service, 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.

1

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.

Voyager Universal Architecture

NameSpace				
CORBA	VDIR	RMI	JNDI	archive directory

Proxy	
dynamic	static (pgen)

Messaging				
sync	oneway	future	publish/ subscribe	distributed event

Object Model			
CORBA	RMI	VNM	DCOM

Transport	
TCP	UDP

Universal Naming Service

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

Voyager Universal Naming Service
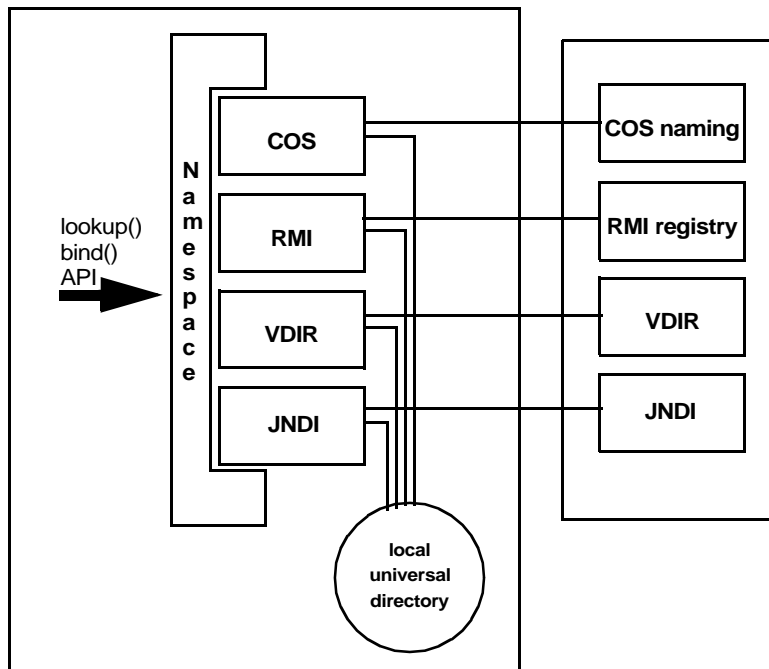
Universal Communications Infrastructure

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

Voyager Universal Communications Infrastructure
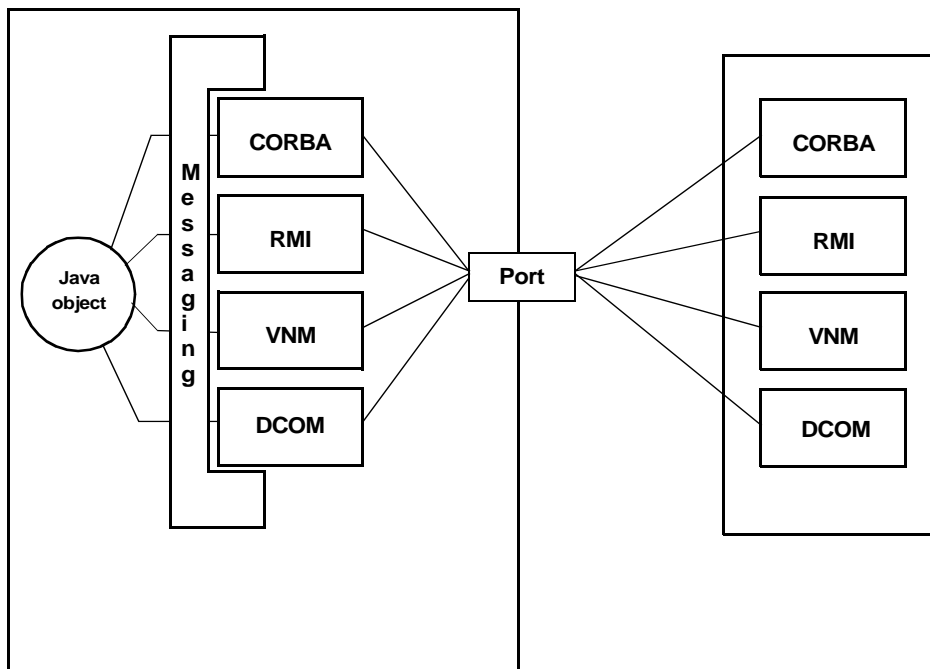
Voyager ORB Features

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.

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.

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.

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 3.0, copyright 1997-1999 objectspace
```

The voyager utility has several options. For more information, refer to

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:
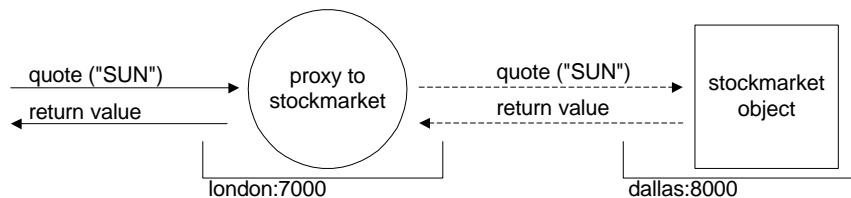
If a remote method throws an exception, it is caught and re-thrown in the local program. If a Voyager-related exception, such as a network error, argument serialization error, etc., occurs and the interface method explicitly throws java.rmi.RemoteException[†], the exception is thrown wrapped in a RemoteException, 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 example on page 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 pinging" 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.

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 example on page 163 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 236 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 activation framework example on page 226 illustrates Snapshot.

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

27

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.

```
aggregate
                        ┌──────────┐
                        │ primary  │
                        │ object   │
          ┌───────┐     └──────────┘     ┌───────┐
          │ facet │                      │ facet │
          └───────┘                      └───────┘
                        ┌───────┐
                        │ facet │
                        └───────┘
```
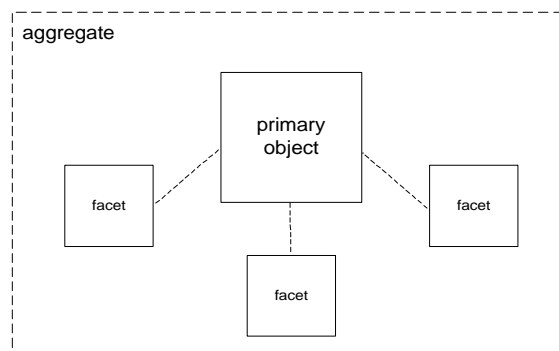
There are several rules associated with facets:

◆ 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 example on page 166 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 example on page 170 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

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 example on page 178 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 36 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 example on page 181 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", newObject[] { 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 example on page 183 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", newObject[] { "SUN" } );
// perform other operations here
int price = result.readInt(); // block for price, if necessary
```

Example   The example on page 185 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 example on page 187 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 example on page 190 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 example on page 192 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 example on page 193 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.
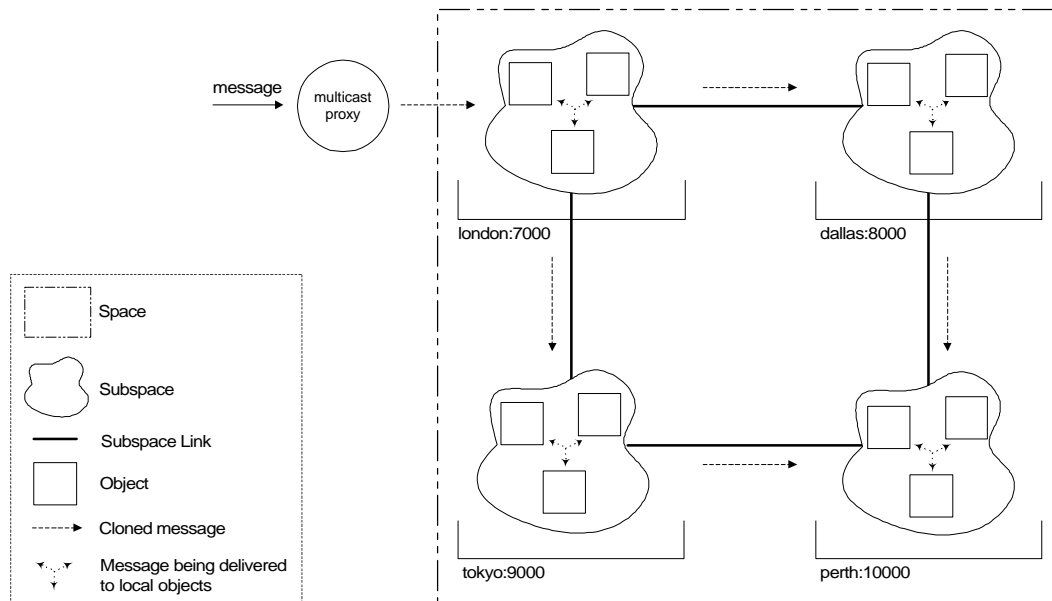
Creating and Populating a Space

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 example on page 196 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 example on page 199 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 to 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 example on page 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 example on page 208 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 example on page 210 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 49 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 that 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().

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 example on page 218 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
Namespace.lookup("Fred")	<Directory @ local program>.get("Fred")
Namespace.lookup("8000/Fred")	<Directory @ //localhost:8000>.get("Fred")
Namespace.lookup("//dallas:8000/Fred")	<Directory @ //dallas:8000>.get("Fred")
Namespace.lookup("//dallas:8000/Fred/Bloggs")	<Directory @ //dallas:8000>.get("Fred/Bloggs")

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

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

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 example on page 220 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 example on page 222 illustrates the JNDI naming service.

Using PersistentDirectory

Pro Only

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.
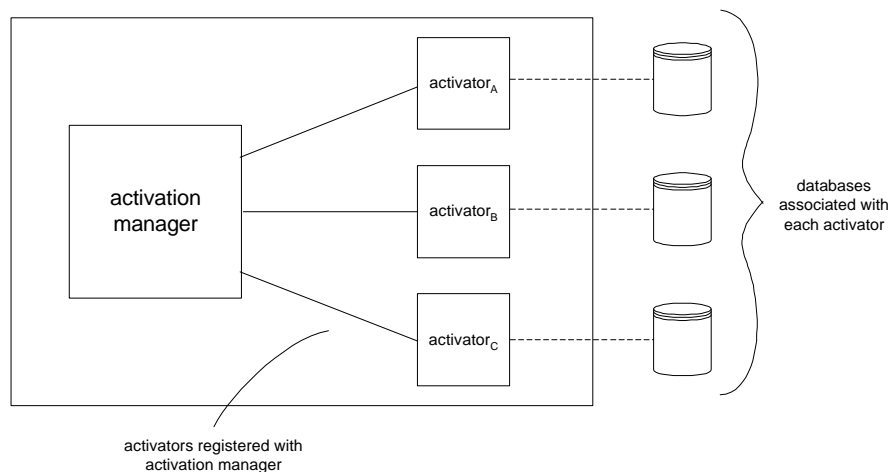
An activator must implement IActivator, defined as follows:

◆ 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.
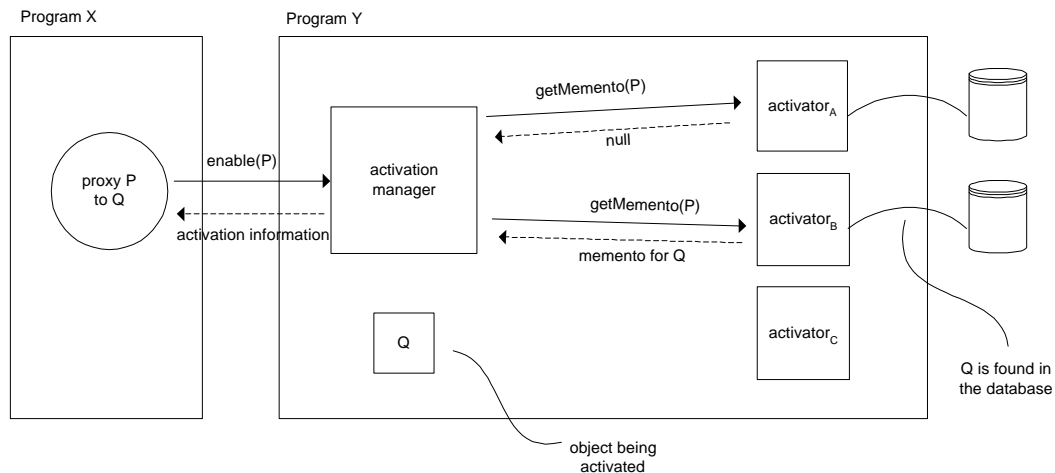
Activating an Object

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.
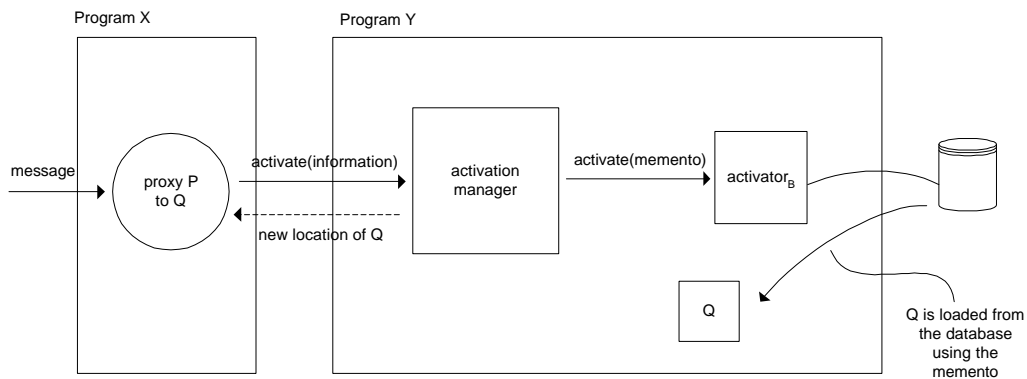
Writing an Activator

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 example on page 226 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 example on page 236 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	shutdown() addSystemListener() removeSystemListener
ClassManager	setParentClassLoader() enableResourceServer() resetClassLoader()
VoyagerClassLoader	addResourceLoader() removeResourceLoader() setResourceLoadingEnabled()

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

Pro Only

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

Pro Only
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

Pro Only
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 example on page 245 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 to 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 examples on page 241 demonstrates two Voyager-enabled applets.

Chapter 9 • Ultra-Light Client, Applets and Servlets

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

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
- case
- context
- enum
- fixed
- inout
- module
- oneway
- readonly
- string
- TRUE
- union
- wstring

- attribute
- char
- default
- exception
- float
- interface
- Object
- out
- sequence
- struct
- typedef
- void

- boolean
- const
- double
- FALSE
- in
- long
- octet
- raises
- short
- switch
- unsigned
- wchar

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 on page 92.

- 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

```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

```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 on page 123.

◆ 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

```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:000000000000000e4944...0000
CORBA server is ready
open account


% java Client
bank IOR = IOR:000000000000000e4944...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

```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 example on page 251 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 on page 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 on page 98. 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 Istring; // placeholder for international string type

  struct NameComponent
   {
   Istring id;
   Istring 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

```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 on page 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 );
 publicContainers.IQueuePackage.Element remove();
 }
```

Containers\IQueuePackage\Element.java

```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

```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

```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 on page 251 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

```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 on page 252 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 IConstants.

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 on page 253 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 on page 254 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
IOException	IDL:omg.org/CORBA/MARSHAL:1.0
ClassNotFoundException	IDL:omg.org/CORBA/MARSHAL:1.0
MethodNotFoundException	IDL:omg.org/CORBA/NO_IMPLEMENT:1.0
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

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

```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

```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 on page 257 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 on page 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 on page 256 illustrates the use of dynamic method invocation.

Naming Service

Pro Only

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 on page 259 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.

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 example on page 261 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 example on page 263 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 example on page shows a SunSoft RMI client looking up an object placed into a Voyager universal directory.

Timers 12

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 example on page 274 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 example on page 276 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 example on page 277 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 example on page 279 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.

Configuration and Management $\boxed{13}$

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 IActivator 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 example on page 301 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 example on page 304 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 example on page 307 demonstrates specifying multiple Voyager properties.

Using the Voyager Management Framework

Pro Only

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 141.

Example

The example on page 282 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 example on page 282 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 IEventListener 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 example on page 282 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 example on page 282 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 on page 282 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

Pro Only

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.

Appendices

Utilities $\left(\begin{array}{c}\textbf{A}\end{array}\right)$

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

   Pro Only

   Use -d *url* to configure this Voyager server from the profile at the given directory URL.

◆ -f (file)

   Pro Only

   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\IVector 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 "MyModule\MyInterface.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.

Pro Only

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 3.0, 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
```

```java
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 )
```

```java
    {
    System.out.println( "news: " + announcement ); // display news
    }
  }
```

Program examples\basics\Basics1.java

```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

```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

```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

```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

```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

```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;

```java
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;

```java
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

```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

```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

```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 3.0, 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\IAlarm.java

```java
// copyright 1997-1999 objectspace

package examples.message;

public interface IAlarm
  {
  int delay( int milliseconds );
  }
```

Class examples\message\Alarm.java

```java
// copyright 1997-1999 objectspace

package examples.message;

public class Alarm implements IAlarm
  {
  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

```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 3.0, 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 alarn
     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 3.0, 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

```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() );
```

```
    // 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 3.0, 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;

```java
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 3.0, 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

```java
// copyright 1997-1999 objectspace

package examples.message;

import com.objectspace.voyager.*;
import com.objectspace.voyager.message.*;
```

```java
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 3.0, 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

```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 3.0, copyright objectspace 1997-1999
delay( 5000 )
done!
```

Window2

```
>java examples.message.Message7

>
```

The source code for Message7.java follows:

Program examples\message\Message7.java

```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 3.0, copyright objectspace 1997-1999
created Consumer( jack )
created Consumer( sasha )
```

Window2

```
 >voyager 9000
voyager 3.0, 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

```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

```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 3.0, 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 3.0, 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

```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

```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 3.0, 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 3.0, 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

```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

```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 3.0, 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 3.0, 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.*;
```

```java
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

```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 3.0, 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
 {
```

```java
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 3.0, copyright objectspace 1997-1999
```

Window2

```
>voyager 9000
voyager 3.0, 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

```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 3.0, copyright objectspace 1997-1999
construct stockmarket
```

```
>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

```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) 3.0, 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

```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 );
```

```java
// 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

```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.
 */
```

```java
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.*;
```

```java
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

```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.

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

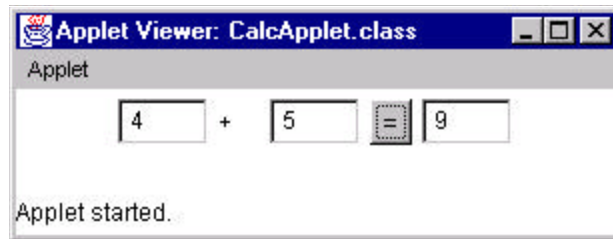From examples\applets, compile the example program:

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 examples.calculator.CalcServer
4 + 5 = 9
```

Window2

```
>appletviewer 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;
```

```java
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

```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 webservers.

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:000000000000001449444c3a49506f73744f66666963653a312e3000
Address of Person( Graham, 36 ) is Address( 123 arkwright road, TX, 75248 )
```

Window2

```
% java Server
postoffice IOR = IOR:000000000000001449444c3a49506f73744f666
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:00000000000000001149444c3a495072696e7465723a312
```

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:0000000000000001149444c3a494c6962726172
CORBA server is ready
printX()
printY()
```

Window2

```
% java Client
library IOR = IOR:0000000000000001149444c3a494c6962726172793a3
```

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:000000000000000f49444c3a4954696d65723a3
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:000000000000000e49444c3a4942616e6b3a31
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:000000000000000e49444c3a4942616e6b3a312e3000000000000001
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 3.0b1, 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 CSCO
```

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, RmiCient2 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 CSCO
```

Window 2

```
> java examples.rmi.RmiClient2
BINDING...
BOUND!
```

The source code for VoyagerServer2.java and RmiClient2.java follows.

Interface examples\rmi\VoyagerServer2.java

```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();
   }
 }
}
```

Ultra-Light Client

The examples in this section demonstrate how Voyager works with ultra-light clients.

Pro Only

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 examples.calculator.CalcServer
4 + 5 = 9
```

Window2

>appletviewer Calculator.html

The source code for ICalculator.java, Calculator.java, CalcServer.java, CalcApplet.java and Calculator.html follows:

Applet examples\applets\LightCalcApplet.java

```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="CalcApplet" 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

```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.

```
 >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

```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

```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" );
```

```
  }
}
```

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) 3.0, 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.
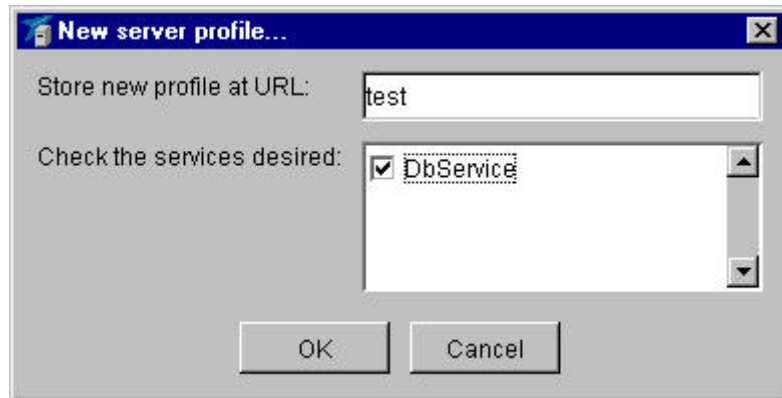
A node called test appears in the *tree view* on the left. Highlight the test node. In the *Startup URL* field, replace 8000 with 9000, and press the *Accept* button.
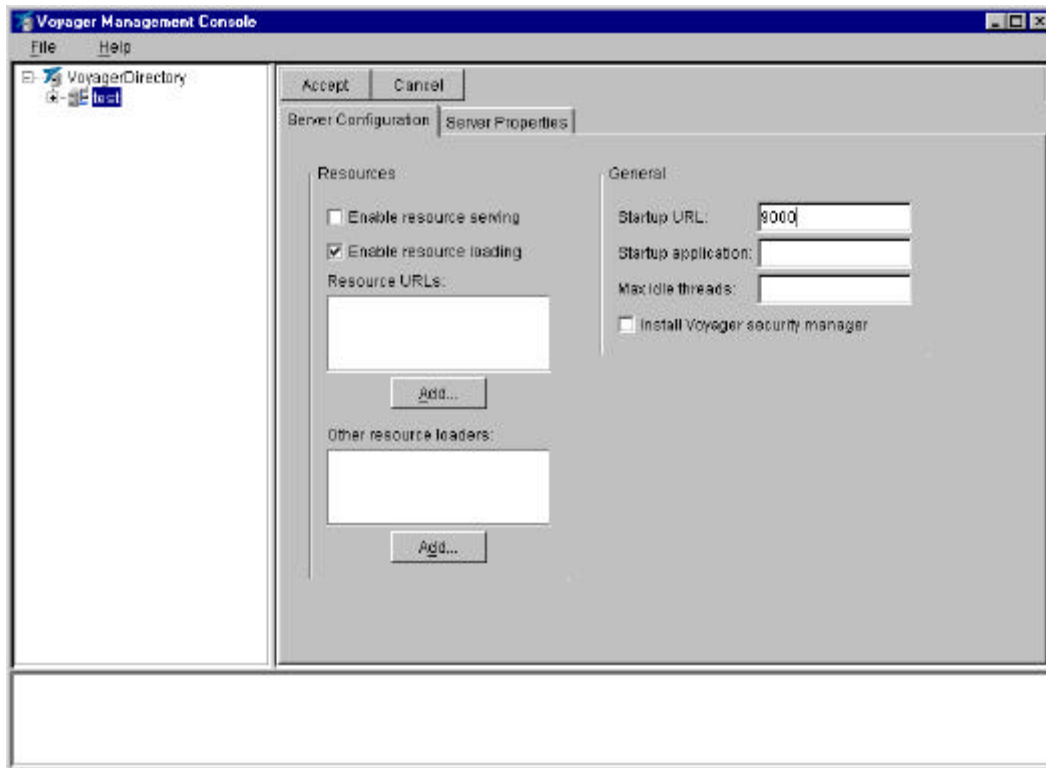
Expand the test node, and highlight the DbService child node. Enter test.db in the
Database name field.
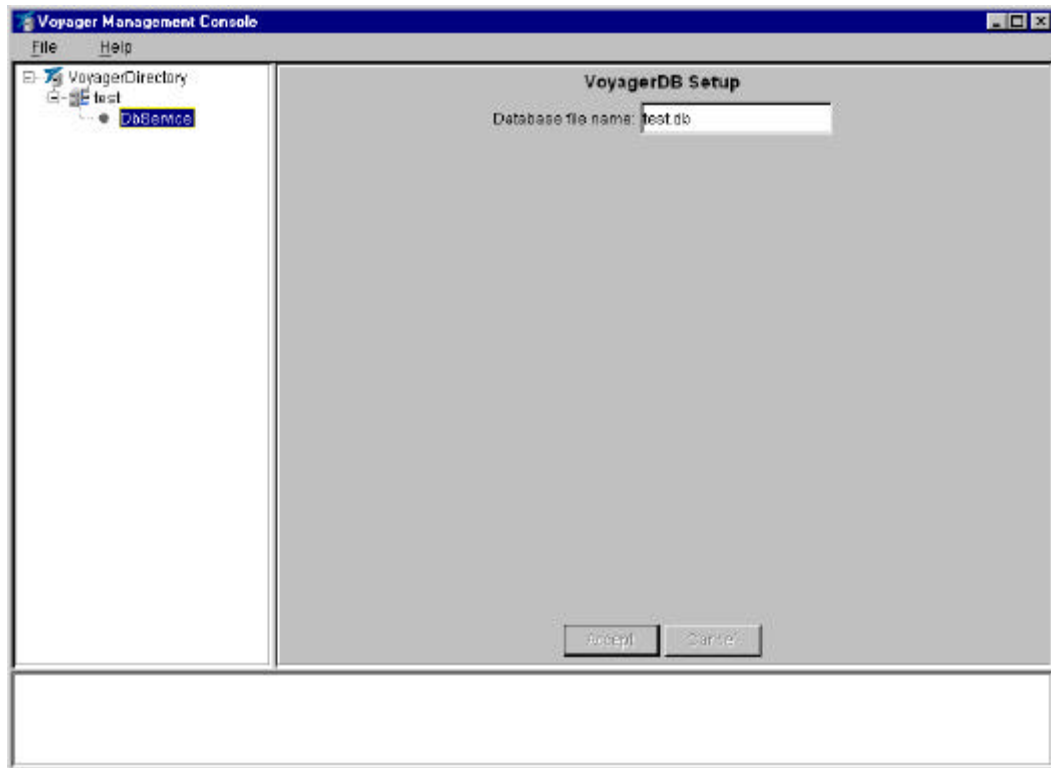
Shut down the console. Start a Voyager server, and associate it with the test entry in the
directory, using the -d option.

Window 3

```
> voyager -d 8000/dir/test
voyager orb professional 3.0, 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.
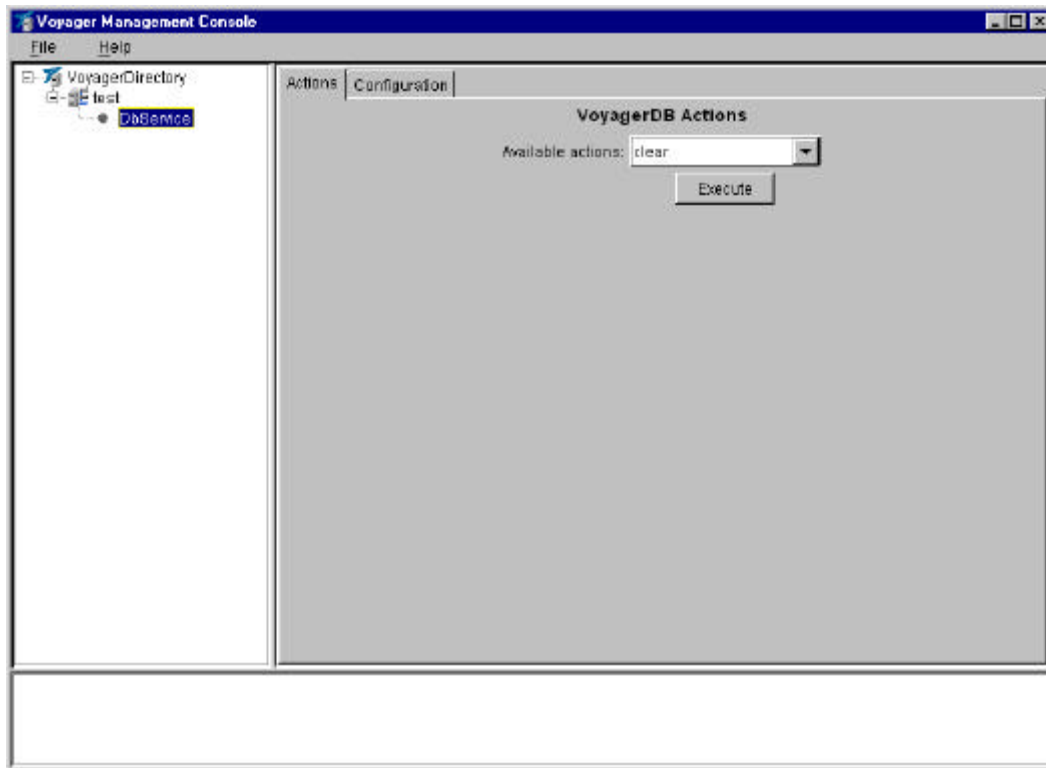
The source code for DbConfig.java, DbMonitor.java, DbService.java, and MonitorTool.java follows. DbConfig is an implementation of IConfiguration, DbMonitor is an implementation of IManagementAgent, DbService is the class being configured and managed, and MonitorTool is an implementation of ITool.

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.*;
```

```java
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()
  {
```

```java
    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 );
   }
  }
```

```java
  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

```java
// copyright 1999 objectspace

package examples.management;
```

```java
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 3.0, 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 3.0, 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 3.0, 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

```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

```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

Pro Only

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
```

>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
```

```java
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

```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;
   }
```

```java
  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

```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
 {
```

```java
  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

```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

```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 );

  };
```