



ZeroC, Inc.

Ice Programming with Java

Student Workbook

A photograph of several clear, rectangular ice cubes melting on a light blue surface. The background is a soft, out-of-focus blue gradient.

Version 2.0.0
November 2010
© ZeroC, Inc.

Disclaimer

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book and ZeroC was aware of the trademark claim, the designations have been printed in initial caps or all caps.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Contents

1	INTRODUCTION TO ICE	1.2
1-1	Lesson Overview.....	1.2
1-2	What is Ice?.....	1.3
1-3	Clients and Servers.....	1.4
1-4	Ice Objects.....	1.5
1-5	Proxies.....	1.7
1-6	Stringified Proxies.....	1.8
1-7	Servants.....	1.10
1-8	At-Most-Once Semantics.....	1.12
1-9	Method Invocation and Dispatch.....	1.14
1-10	Client and Server Structure.....	1.16
1-11	Ice Services.....	1.18
2	THE SLICE INTERFACE DEFINITION LANGUAGE	2.2
2-1	Lesson Overview.....	2.2
2-2	What is Slice?.....	2.3
2-3	Single-Language Development.....	2.4
2-4	Cross-Language Development.....	2.5
2-5	Slice Source Files.....	2.6
2-6	Comments and Keywords.....	2.7
2-7	Identifiers.....	2.8
2-8	Modules.....	2.10
2-9	The Ice Modules.....	2.11
2-10	Basic Slice Types.....	2.12
2-11	Enumerations.....	2.14
2-12	Structures.....	2.15
2-13	Sequences.....	2.17
2-14	Dictionaries.....	2.18
2-15	Constants and Literals.....	2.19
2-16	Interfaces.....	2.21
2-17	Operations and Parameters.....	2.23
2-18	idempotent Operations.....	2.24
2-19	User Exceptions.....	2.26
2-20	Exception Inheritance.....	2.27
2-21	Ice Run-Time Exceptions.....	2.29
2-22	Run-Time Exceptions Raised by the Server.....	2.32
2-23	Proxies.....	2.34
2-24	Interface Inheritance.....	2.36
2-25	Interface Inheritance Limitations.....	2.38
2-26	Implicit Inheritance from Object.....	2.39
2-27	Self-Referential Interfaces & Forward Declarations.....	2.40
2-28	Classes.....	2.41
2-29	Passing Classes as Parameters and Slicing.....	2.42
2-30	Classes as Unions.....	2.44
2-31	Self-Referential Classes.....	2.45
2-32	Classes with Operations.....	2.47
2-33	Classes Implementing Interfaces.....	2.48
2-34	Class Inheritance Limitations.....	2.49
2-35	Pass-by-Value Versus Pass-by-Reference.....	2.50
2-36	Architectural Implications of Classes.....	2.51
2-37	Classes Versus Structures.....	2.52
2-38	The :: Scope Qualification Operator.....	2.53
2-39	Type Identifiers.....	2.55

2-40	Operations on Object	2.56
2-41	Local Types	2.58
2-42	Metadata	2.59
2-43	The <code>slice2java</code> Compiler	2.60
3	ASSIGNMENT 1: CREATING SLICE DEFINITIONS.....	3.2
3-1	Exercise Overview.....	3.2
3-2	A Simple Remote File System	3.3
3-3	What You Need to Do	3.4
3-4	Slice Definitions for a Simple Remote File System.....	3.5
4	CLIENT-SIDE SLICE-TO-JAVA MAPPING	4.2
4-1	Lesson Overview.....	4.2
4-2	Client-Side Java Mapping	4.3
4-3	Initializing the Ice Run Time	4.4
4-4	Mapping for Identifiers.....	4.5
4-5	Mapping for Modules.....	4.6
4-6	Mapping for Built-In Types	4.8
4-7	Mapping for Enumerations	4.9
4-8	Mapping for Structures.....	4.10
4-9	Mapping for Sequences	4.11
4-10	Custom Mapping for Sequences.....	4.12
4-11	Mapping for Dictionaries	4.13
4-12	Custom Mapping for Dictionaries.....	4.14
4-13	Mapping for Constants	4.15
4-14	Mapping for User Exceptions	4.16
4-15	Mapping for Run-Time Exceptions.....	4.18
4-16	Mapping for Interfaces	4.19
4-17	The Proxy Interface	4.21
4-18	Methods on <code>Ice.ObjectPrx</code>	4.23
4-19	Proxy Helpers.....	4.26
4-20	Mapping for Operations.....	4.28
4-21	Mapping for Return Values and In-Parameters	4.29
4-22	Mapping for Out-Parameters.....	4.30
4-23	Exception Handling	4.32
4-24	Mapping for Classes	4.34
4-25	Inheritance from <code>Ice.Object</code>	4.37
4-26	Abstract Classes	4.38
4-27	Class Factories.....	4.39
4-28	Factory Registration	4.41
4-29	Default Factory	4.42
4-30	Stringified Proxies	4.43
4-31	Compiling and Running a Client.....	4.45
5	ASSIGNMENT 2: CREATING AN ICE CLIENT	5.2
5-1	Exercise Overview.....	5.2
5-2	Creating a Client for the Remote Filesystem	5.3
5-3	What You Need to Do	5.4
5-4	The <code>main</code> Method	5.6
5-5	The <code>listRecursive</code> Method.....	5.8
6	SERVER-SIDE SLICE-TO-JAVA MAPPING	6.2
6-1	Lesson Overview.....	6.2
6-2	Server-Side Java Mapping.....	6.3
6-3	Initializing the Ice Run Time	6.4

6-4	Server-Side Initialization	6.5
6-5	Mapping for Interfaces	6.6
6-6	Mapping for Interfaces (cont. 1)	6.8
6-7	Mapping for Parameters.....	6.9
6-8	Throwing Exceptions	6.10
6-9	Tie Classes.....	6.11
6-10	Creating an Object Adapter.....	6.13
6-11	Object Adapter States	6.14
6-12	Controlling Adapter State	6.15
6-13	Object Identity	6.16
6-14	Stringified Object Identity	6.17
6-15	The Active Servant Map (ASM).....	6.18
6-16	Activating Servants	6.19
6-17	Creating Proxies.....	6.20
6-18	The <code>Ice::Application</code> Class	6.21
6-19	Shutdown Hook.....	6.23
6-20	Compiling and Running a Server	6.25
7	ASSIGNMENT 3: CREATING AN ICE SERVER	7.2
7-1	Exercise Overview.....	7.2
7-2	Creating a Server for the Remote Filesystem.....	7.3
7-3	What You Need to Do	7.4
7-4	The <code>server</code> Class	7.6
7-5	The <code>Directory</code> Class.....	7.8
7-6	The <code>FileI</code> Class	7.10
8	PROPERTIES AND CONFIGURATION.....	8.2
8-1	Lesson Overview.....	8.2
8-2	Ice Properties	8.3
8-3	Configuration Files	8.4
8-4	Setting Ice Properties on the Command Line	8.5
8-5	Ice Initialization.....	8.6
8-6	Reading Properties Programmatically.....	8.7
8-7	Using <code>InitializationData</code>	8.9
8-8	Command-Line Application Properties	8.10
8-9	Commonly-Used Ice Properties	8.12
8-10	Converting Properties to Proxies	8.13
8-11	Object Adapter Properties.....	8.14
9	ASSIGNMENT 4: USING PROPERTIES	9.2
9-1	Exercise Overview.....	9.2
9-2	Adding an Application-Specific Property	9.3
9-3	What You Need to Do	9.4
9-4	Using Properties.....	9.6
10	MULTI-THREADED ICE	10.2
10-1	Lesson Overview.....	10.2
10-2	Ice Threading Model	10.3
10-3	Thread Pool Configuration	10.4
10-4	Thread Pool Configuration (cont. 1).....	10.5
10-5	Thread Pool Configuration (cont. 2).....	10.6
10-6	Thread Safety.....	10.7
11	ASSIGNMENT 5: THREAD SAFETY.....	11.2

11-1	Exercise Overview.....	11.2
11-2	Thread Safety.....	11.3
11-3	What You Need to Do	11.4
11-4	Server Modifications.....	11.6
11-5	Client Modifications	11.10
12	OBJECT LIFE CYCLE.....	12.2
12-1	Lesson Overview.....	12.2
12-2	Object Creation	12.3
12-3	Object Creation and Thread Safety.....	12.5
12-4	The Current Object.....	12.6
12-5	Object Destruction.....	12.8
12-6	Implementing Object Destruction	12.9
12-7	Object Destruction and Thread Safety.....	12.11
12-8	When to Remove Servant State?.....	12.13
12-9	Object Identity and Uniqueness	12.15
12-10	Object Identity and Uniqueness (cont. 1).....	12.16
12-11	Object Identity and Uniqueness (cont. 2).....	12.17
12-12	Uniqueness Recommendations	12.19
12-13	Dealing with Stale Objects	12.20
12-14	Dealing with Stale Objects (cont. 1).....	12.21
12-15	Dealing with Stale Objects (cont. 2).....	12.23
12-16	Dealing with Stale Objects (cont. 3).....	12.24
12-17	Dealing with Stale Objects (cont. 4).....	12.28
12-18	Dealing with Stale Objects (cont. 5).....	12.29
12-19	Dealing with Stale Objects (cont. 6).....	12.31
12-20	Dealing with Stale Objects (cont. 7).....	12.33
13	ASSIGNMENT 6: OBJECT LIFE CYCLE	13.2
13-1	Exercise Overview.....	13.2
13-2	Life Cycle.....	13.3
13-3	What You Need to Do	13.4
13-4	Thread Safety Modifications.....	13.6
13-5	<code>createDir</code> Implementation	13.8
13-6	<code>createFile</code> Implementation	13.10
13-7	<code>DirectoryI.destroy</code> Implementation.....	13.12
13-8	<code>FileI.destroy</code> Implementation	13.14
14	GLACIER2	14.2
14-1	Lesson Overview.....	14.2
14-2	Running an Ice Server Behind a Firewall.....	14.3
14-3	Glacier2	14.5
14-4	Glacier2 as a Firewall.....	14.6
14-5	Glacier2 Behind a Firewall	14.7
14-6	Running Glacier2.....	14.8
14-7	Glacier2 Configuration	14.9
14-8	Glacier2 Sessions	14.10
14-9	Client Configuration.....	14.12
14-10	Creating a Password File	14.14
14-11	Custom Authentication	14.15
14-12	The Admin Interface	14.16
14-13	Custom Object Identities	14.17
14-14	Session Timeouts.....	14.18
14-15	Explicit Session Management.....	14.19
14-16	Supporting Callbacks	14.21

14-17	Supporting Callbacks (cont. 1).....	14.22
14-18	Supporting Callbacks (cont. 2).....	14.24
14-19	Helper Classes.....	14.25
14-20	The Glacier2.Application Class.....	14.26
14-21	The Glacier2.Application Class (cont. 1)	14.28
15	ASSIGNMENT 7: USING GLACIER2	15.2
15-1	Exercise Overview.....	15.2
15-2	Using Glacier2.....	15.3
15-3	What You Need to Do	15.4
15-4	Client Modifications	15.5
15-5	Client Configuration.....	15.7
15-6	Glacier2 Configuration	15.8
15-7	Glacier2 Password File	15.9
16	ICEGRID	16.2
16-1	Lesson Overview.....	16.2
16-2	IceGrid.....	16.3
16-3	IceGrid Components	16.5
16-4	IceGrid Architecture.....	16.6
16-5	Indirect Proxies.....	16.7
16-6	Client Configuration.....	16.8
16-7	Registry Endpoints	16.9
16-8	Registry Configuration.....	16.10
16-9	Node Configuration	16.11
16-10	Starting an IceGrid Node.....	16.13
16-11	Starting the Registry.....	16.14
16-12	Server Configuration	16.15
16-13	Server Configuration (cont. 1).....	16.17
16-14	Server Configuration (cont. 2).....	16.18
16-15	Server Configuration (cont. 3).....	16.19
16-16	Server Configuration (cont. 4).....	16.20
16-17	Server Configuration (cont. 5).....	16.21
16-18	Command-Line Administration.....	16.22
16-19	icegridadmin Application Commands	16.23
16-20	icegridadmin Node Commands	16.24
16-21	icegridadmin Server Commands	16.25
16-22	icegridadmin Server Commands (cont. 1).....	16.26
16-23	icegridadmin Server Commands (cont. 2).....	16.27
16-24	Server Activation and Deactivation	16.28
16-25	The Process Object.....	16.29
16-26	Server Environment	16.30
16-27	Server Code Changes.....	16.31
16-28	The Graphical Admin Tool	16.32
16-29	Well-Known Proxies	16.34
16-30	Well-Known Proxies (cont. 1).....	16.36
16-31	Security Considerations	16.37
16-32	Troubleshooting.....	16.38
16-33	Other Features	16.39
17	ASSIGNMENT 8: USING ICEGRID.....	17.2
17-1	Exercise Overview.....	17.2
17-2	Using IceGrid.....	17.3
17-3	What You Need to Do	17.4
17-4	Registry Configuration.....	17.6

17-5	Node Configuration	17.7
17-6	Deployment Descriptor.....	17.8
17-7	Admin Configuration.....	17.9
17-8	Server Source Modification	17.10
17-9	Client Modification.....	17.12
18	THE ICE RUN TIME IN DETAIL	18.2
18-1	Lesson Overview.....	18.2
18-2	The <code>Ice::Communicator</code> Interface	18.3
18-3	The <code>Ice::Communicator</code> Interface (cont. 1)	18.5
18-4	Creating a Communicator	18.7
18-5	Object Adapters.....	18.8
18-6	Servant Locators	18.9
18-7	Threading Guarantees for Servant Locators.....	18.11
18-8	Servant Locator Registration.....	18.12
18-9	Call Dispatch Rules	18.13
18-10	Implementing Servant Locators	18.14
18-11	Implementing <code>locate</code>	18.15
18-12	Information Provided to <code>locate</code>	18.17
18-13	Lazy Initialization	18.19
18-14	Creating Proxies.....	18.21
18-15	Default Servants.....	18.22
18-16	Default Servants (cont. 1)	18.24
18-17	Default Servants (cont. 2)	18.26
18-18	Hybrid Approaches and Caching	18.27
18-19	Evictors.....	18.28
18-20	Evictors (cont. 1)	18.29
18-21	Evictors (cont. 2)	18.30
18-22	Evictors (cont. 3)	18.31
18-23	Evictor Implementation.....	18.32
18-24	Evictor Implementation (cont. 1)	18.34
18-25	Using <code>EvictorBase</code>	18.39
19	LICENSE.....	19.1

1 Introduction to Ice

1 Introduction to Ice

1-1 Lesson Overview

Lesson Overview

- This lesson covers:
 - the motivation for using Ice
 - the fundamentals of the Ice architecture
 - the Ice object model
- This lesson also provides an overview of the major Ice components (including some components not covered in this course).
- By the end of this lesson, you will have a basic understanding of the Ice architecture and how Ice helps you to develop distributed applications.

Introduction to Ice
Copyright © 2005-2010 ZeroC, Inc.1-1

Notes:

This lesson covers the motivation for using Ice, the fundamentals of the Ice architecture, and the Ice object model. The lesson also provides an overview of the major Ice components (including components not covered in this course).

1-1-1 Lesson Objectives

By the completion of this lesson, you will have a basic understanding of the Ice architecture and how Ice helps you to develop distributed applications.

1-2 What is Ice?

What is Ice?

- An object-oriented distributed middleware platform.
- Ice includes:
 - object-oriented RPC mechanism
 - language-neutral specification language (Slice)
 - language mappings for various languages: C++, Java, C#, Python, Objective-C, Ruby and PHP (Ruby and PHP for the client-side only)
 - support for different transports (TCP, SSL, UDP) with highly-efficient protocol
 - external services (server activation, firewall traversal, etc.)
 - integrated persistence (Freeze)
 - threading support



Introduction to Ice
Copyright © 2005-2010 ZeroC, Inc.

1-2

Notes:

Ice (*Internet Communications Engine*) is an object-oriented middleware platform. It supports a variety of operating systems, compilers, and programming languages and allows you to create heterogeneous distributed applications. So Ice allows you to easily create distributed applications that run on different operating systems and are written in different languages. Ice allows these applications to seamlessly interoperate.

A specification language (Slice) allows you to define types and interfaces used by your applications, regardless of what languages you use to implement applications. A compiler then translates such language-independent specifications into a language-specific API.

The generated API takes care of many of the communications chores that you would otherwise have to implement yourself. In addition, Ice supports not only TCP, but also SSL and UDP transports. A plug-in interface allows you to add support for other transports without having to modify the Ice source code.

Ice provides a number of services that implement commonly-required functionality, such as server-activation on demand, firewall traversal, and persistence.

The Ice run time is fully threaded and (for C++) provides a thread-abstraction library that helps you write threaded code that is source-code portable for both Windows and UNIX environments.

1-3 Clients and Servers

Clients and Servers

A client–server system is any software system in which different parts of the system cooperate on an overall task.

- A server is an entity that, on request, provides a service (such as a computation) to clients. Servers are passive.
- A client is an entity that requests services from servers. Clients are active.
- Client and server often run on separate machines, but might also run on the same machine or be linked into a single process.

Frequently, clients and servers are not “pure” clients and servers.

- A server might act as a client, and a client might act a server.
- Client and server are therefore roles that have a well-defined meaning only for the duration of a single request. The initiating side is, by definition, the client; the responding side is, by definition, the server.



Notes:

The terms client and server are not firm designations for particular parts of an application; rather, they denote roles that are taken by parts of an application for the duration of a request:

- Clients are active entities. They issue requests for service to servers.
- Servers are passive entities. They provide services in response to client requests.

Frequently, servers are not “pure” servers, in the sense that they never issue requests and only respond to requests. Instead, servers often act as a server on behalf of some client but, in turn, act as a client to another server in order to satisfy their client’s request.

Similarly, clients often are not “pure” clients, in the sense that they only request service from an object. Instead, clients are frequently client–server hybrids.

Example: A client might start a long-running operation on a server; as part of starting the operation, the client can provide a callback object to the server that is used by the server to notify the client when the operation is complete. In that case, the client acts as a client when it starts the operation, and as a server when it is notified that the operation is complete. Such role reversal is common in many systems, so, frequently, client–server systems could be more accurately described as peer-to-peer systems.

1-4 Ice Objects

Ice Objects

- An Ice object is a conceptual entity, that is, an abstraction.
- An Ice object:
 - can exist in the local or a remote address space
 - responds to operation invocations
 - can have multiple redundant instantiations
 - has one or more interfaces (facets), and has a single most-derived default interface (the default facet)
 - provides operations that can accept in-parameters, and can return out-parameters and/or a return value
 - has a unique object identity

Introduction to Ice
Copyright © 2005-2010 ZeroC, Inc.1-4

Notes:

An Ice object is a conceptual entity, or abstraction. It can be characterized by the following points:

- An Ice object is an entity in the local or a remote address space that can respond to client requests.
- A single Ice object can be instantiated in a single server or, redundantly, in multiple servers. If an object has multiple simultaneous instantiations, it is still a single Ice object.
- Each Ice object has one or more interfaces. An interface is a collection of named operations that are supported by an object. Clients issue requests by invoking operations.

An operation has zero or more parameters as well as a return value. Parameters and return values have a specific type. Parameters are named and have a direction: in-parameters are initialized by the client and passed to the server; out-parameters are initialized by the server and passed to the client. (The return value is simply a special out-parameter.)

- An Ice object has a distinguished interface, known as its main interface. In addition, an Ice object can provide zero or more alternate interfaces, known as facets. Clients can select among the facets of an object to choose the interface they want to work with.
- Each Ice object has a unique object identity. An object's identity is an identifying value that distinguishes the object from all other objects. The Ice object model assumes that object identities are globally unique, that is, no two objects within an Ice communication domain can have the same object identity.

In practice, you need not use object identities that are globally unique, such as UUIDs, only identities that do not clash with any other identity within your domain of interest.

1-5 Proxies

Proxies

- Clients contact Ice objects via proxies.
- A proxy is a handle that uniquely denotes an Ice object.
- A proxy is the local ambassador for a (possibly remote) Ice object.
- When a client invokes an operation on a proxy, the Ice run time:
 1. Locates the Ice object
 2. Activates the object's implementation within the server
 3. Transmits in-parameters to the object
 4. Waits for the operation to complete
 5. Returns any out-parameters and the return value to the client (or an exception in case of an error)

Introduction to Ice
Copyright © 2005-2010 ZeroC, Inc.1-5

Notes:

Proxies are handles that clients use to access Ice objects. You can think of a proxy as akin to a Java reference, except that a proxy can denote an object in a remote address space. When a client invokes an operation on a proxy, the Ice run time takes care of locating the object's server¹, instantiating the object if necessary, and transmitting parameters between client and server.

A proxy encapsulates all the necessary information for this sequence of steps to take place. In particular, a proxy contains:

- addressing information that allows the client-side run time to contact the correct server,
- an object identity that identifies which particular object in the server is the target of a request,
- an optional facet identifier that determines which particular facet of an object the proxy refers to.

¹ Using the external IceGrid service, a proxy invocation can also cause the Ice object's server to be started automatically. See Chapter 16 for more information on IceGrid.

1-6 Stringified Proxies

Stringified Proxies

- Proxies can be converted to and from strings.
`SimplePrinter: default -h host.xyz.com -p 10000`
- This is a proxy for an object with identity `SimplePrinter`.
- The object's server runs on `host.xyz.com` and listens on port 10000 for incoming requests.
- The server can be contacted using the configured default protocol. (If no default protocol is configured, the protocol defaults to TCP).
- Because such a proxy directly contains the endpoint at which the server can be found, it is known as a *direct* proxy. The general form of stringified direct proxies is:
`<identity>: <endpoint>[: <endpoint>. . .]`
- Endpoints have the general form:
`<protocol> [-h <host>] [-p <port>] [-t timeout] [-z]`



Introduction to Ice
 Copyright © 2005-2010 ZeroC, Inc.

1-6

Notes:

Proxies can be converted to and from strings. No part of a proxy is opaque; if a client knows the identity of an Ice object, and knows on which host and port the object's server listens, the client can create a proxy to that Ice object at any time.

Direct proxies contain the identity of the Ice object, as well as the endpoint(s) at which the server listens. If a proxy contains multiple endpoints, the Ice run time will use one of these endpoints to contact the server. This provides redundancy: invocations on the object fail only if none of the object's endpoints are functional.

The protocol identifier of an endpoint can be `default`, in which case the configured default protocol will be used (TCP if no default protocol is configured). The protocol can also be specified explicitly as `tcp`, `ssl`, or `udp`.

The `-h` option specifies the host on which the server runs (and defaults to the local host if omitted).

The `-p` option specifies the port on which the server listens.

The `-t` option can be used to set a timeout that causes operation invocations to return with an exception if the operation does not complete within the specified number of seconds.

The `-z` option specifies that protocol compression is to be used for invocations via this proxy.

Note: UDP endpoints support a few additional options—consult the Ice manual for details.

1-7 Servants

Servants

A servant is a server-side programming-language artifact that provides the concrete representation of an abstract Ice object.

Servants are said to *incarnate* Ice objects.

- Typically, servants are object instances with methods that correspond to the operations supported by an Ice object.
- Servants are written by you, the developer.
- When a client invokes an operation, the Ice run time takes care of invoking the corresponding method on the servant.
- The method bodies on a servant provide the behavior of the corresponding Ice object.
- A single servant can incarnate a single Ice object, or simultaneously incarnate several Ice objects.
- A single Ice object can have multiple servants (typically in different servers, for redundancy).



Notes:

An Ice object is a conceptual entity that has a type, identity, and addressing information. However, client requests ultimately must end up with a concrete server-side processing entity that can provide the behavior for an operation invocation. To put this differently, a client request must ultimately end up executing code inside the server.

The server-side artifact that provides behavior for operation invocations is known as a servant. A servant provides substance for (or incarnates) one or more Ice objects. In practice, a servant is simply an instance of a class that is written by the server developer and that is registered with the server-side run time as the servant for one or more Ice objects. Methods on the class correspond to the operations on the Ice object's interface and provide the behavior for the operations.

A single servant can incarnate a single Ice object at a time or several Ice objects simultaneously. If the former, the identity of the Ice object incarnated by the servant is implicit in the servant. If the latter, the servant is provided the identity of the Ice object with each request, so it can decide which object to incarnate for the duration of the request.

Conversely, a single Ice object can have multiple servants. For example, we might choose to create a proxy for an Ice object with two different addresses for different machines. In that case, we will have two servers, with each server containing a servant for the same Ice object.

When a client invokes an operation on such an Ice object, the client-side run time sends the request to exactly one server. In other words, multiple servants for a single Ice object allow you to build redundant systems: the client-side run time attempts to send the request to one server and, if that attempt fails, sends the request to the second server. Only if the second attempt fails is an error reported back to the client-side application code.

1-8 At-Most-Once Semantics

At-Most-Once Semantics

The Ice run time guarantees at-most-once semantics.

A single operation invocation by a client is guaranteed to:

- either invoke the operation exactly once
- or invoke the operation not at all

It is impossible for a single invocation of a client to result in the operation being invoked more than once.

At-most-once semantics are important if an operation is not idempotent.

You can mark individual operations as idempotent to relax the strict at-most-once semantics.



Notes:

Ice requests have at-most-once semantics by default: the Ice run time does its best to deliver a request to the correct destination and, depending on the exact circumstances, may retry a failed request. Ice guarantees that it will either deliver the request, or, if it cannot deliver the request, inform the client with an appropriate exception; under no circumstances is a request delivered twice, that is, retries are attempted only if it is known that a previous attempt definitely failed.²

At-most-once semantics are important because they guarantee that operations that are not idempotent can be used safely. An idempotent operation is an operation that, if executed twice, has the same effect as if executed once.

Example: `x = 1` is an idempotent operation. If we execute the operation twice, the end result is the same as if we had executed it once. On the other hand, `x++`; is not idempotent. If we execute the operation twice, the end result is not the same as if we had executed it once.

² This guarantee does not hold for UDP invocations, due to the nature of UDP. (Duplicated UDP packets can lead to violation of at-most-once semantics.)

Ice permits you to mark operations as idempotent, which indicates to the Ice run time that it is safe to violate at-most-once semantics. For such operations, the Ice run time uses a more aggressive error recovery strategy that can result in an operation being executed more than once.

This ability to relax at-most-once semantics allows us to build distributed systems that are more robust in the presence of network failures. However, realistic systems also require non-idempotent operations, so at-most-once semantics are a necessity even though they make the system less robust in the presence of network failures.

1-9 Method Invocation and Dispatch

Method Invocation and Dispatch

Ice supports:

- Oneway and twoway synchronous method invocation
- Oneway and twoway asynchronous method invocation (AMI)
- Batched oneway invocation
- Datagram invocation
- Batched datagram invocation
- Synchronous method dispatch
- Asynchronous method dispatch (AMD)



Introduction to Ice
Copyright © 2005-2010 ZeroC, Inc.

1-9

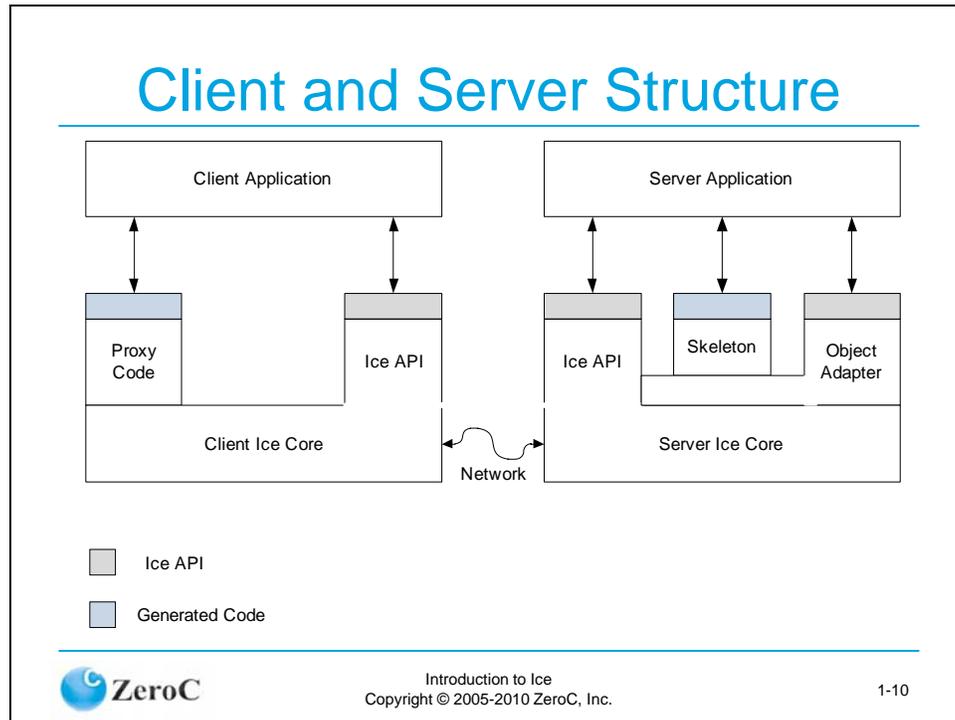
Notes:

Ice provides a rich set of invocation modes.

- Synchronous twoway invocation is the most common way to invoke a remote operation: the client invokes the operation and blocks until the operation completes.
- Asynchronous method invocation (AMI) allows the client to start a twoway invocation, but the thread of control is returned to the client without waiting for the operation to complete. In fact, Ice guarantees that such invocations *never* block the calling thread. When the operation is complete, the client typically receives the results via a callback.
- Oneway invocations apply only to operations that do not return anything (have `void` return type, no out-parameters, and do not raise user exceptions). A synchronous oneway invocation returns the thread of control as soon as the invocation has been written to the client's local transport, whereas an asynchronous oneway invocation always returns the thread of control immediately. (If the invocation fails in the server for some reason, the client is not notified.)
- It is possible to group a number of oneway invocations into a batch and send them all at once, instead of one after another. This reduces network overhead.

- Datagram invocations can be used for servers that provide UDP endpoints. Like oneway invocations, datagram invocations can be made only on operations that do not return anything. And, due to the nature of UDP, they are unreliable.
 - As for oneway invocations, datagram invocations can be batched for efficiency.
- On the server side, Ice supports:
- Synchronous method dispatch. Each client invocation ties up a server thread for the duration of the operation. The operation completes when the corresponding method in the server returns.
 - Asynchronous method dispatch (AMD). The server can use fewer threads than there are concurrent invocations to service these invocations.

1-10 Client and Server Structure



Notes:

Both client and server consist of a mixture of application code, library code, and code generated from Slice definitions:

- The Ice core contains the client- and server-side run-time support for remote communication. Much of this code is concerned with the details of networking, threading, byte ordering, and many other networking-related issues that we want to keep away from application code. The Ice core is provided as a number of libraries that client and server link with.
- The generic part of the Ice core (that is, the part that is independent of the specific types you have defined in Slice) is accessed through the Ice API. You use the Ice API to take care of administrative chores, such as initializing and finalizing the Ice run time. The Ice API is identical for clients and servers (although servers use a larger part of the API than clients).
- The proxy code is generated from your Slice definitions and, therefore, specific to the types of objects and data you have defined in Slice. The proxy code has two major functions:
 - It provides a down-call interface for the client. Calling a function in the generated proxy API ultimately ends up sending an RPC message to the server that invokes a corresponding function on the target object.

- It provides *marshaling* and *unmarshaling* code.
Marshaling is the process of serializing a complex data structure, such as a sequence or a dictionary, for transmission on the wire. The marshaling code converts data into a form that is standardized for transmission and independent of the byte order and padding rules of the local machine.
Unmarshaling is the reverse of marshaling, that is, deserializing data that arrives over the network and reconstructing a local representation of the data in types that are appropriate for the programming language in use.
- The skeleton code is also generated from your Slice definitions and, therefore, specific to the types of objects and data you have defined in Slice. The skeleton code is the server-side equivalent of the client-side proxy code: it provides an up-call interface that permits the Ice run time to transfer the thread of control to the application code you write. The skeleton also contains marshaling and unmarshaling code, so the server can receive parameters sent by the client, and return parameters and exceptions to the client.
- The object adapter is a part of the Ice API that is specific to the server side: only servers use object adapters. An object adapter has several functions:
 - The object adapter maps incoming requests from clients to specific methods on programming-language objects. In other words, the object adapter tracks which servants with what object identity are in memory.
 - The object adapter is associated with one or more transport endpoints. If more than one transport endpoint is associated with an adapter, the servants incarnating objects within the adapter can be reached via multiple transports. For example, you can associate both a TCP/IP and a UDP endpoint with an adapter, to provide alternate quality-of-service and performance characteristics.
 - The object adapter is responsible for the creation of proxies that can be passed to clients. The object adapter knows about the type, identity, and transport details of each of its objects and embeds the correct details when the server-side application code requests the creation of a proxy.

Note: As far as the process view is concerned, there are only two processes involved: the client and the server. All the run time support for distributed communication is provided by the Ice libraries and the code that is generated from Slice definitions. (A third process, IceGrid, provides features such as object location and on-demand server activation.)

1-11 Ice Services

Ice Services

Ice provides a number of services:

- Persistence service (Freeze)
- Replication, load balancing, server activation service (IceGrid)
- Application server (IceBox)
- Publish–subscribe service (IceStorm)
- Software distribution and patching service (IcePatch2)
- Firewall traversal and session management (Glacier2)

Freeze is a library; the other services are implemented as stand-alone processes.



Introduction to Ice
Copyright © 2005-2010 ZeroC, Inc.

1-11

Notes:

The Ice core provides a sophisticated client–server platform for distributed application development. However, realistic applications usually require more than just a remoting capability: typically, you also need a way to start servers on demand, distribute proxies to clients, distribute asynchronous events, configure your application, distribute patches for an application, and so on.

Ice ships with a number of services that provide these and other features. The services are implemented as Ice servers to which your application acts as a client. None of the services use Ice-internal features that are hidden from application developers so, in theory, you could develop equivalent services yourself. Having these services available, as part of the platform, allows you to focus on application development, instead of having to build a lot of infrastructure first. Moreover, building such services is not a trivial effort, so it pays to know what is available and use it instead of reinventing your own wheel.

1-11-1 Freeze

Ice has a built-in object persistence service, known as *Freeze*. Freeze makes it easy to store object state in a database: you define the state stored by your objects in Slice, and the Freeze compiler generates code that stores and retrieves object state to and from a database. Freeze uses Berkeley DB as its database.

1-11-2 IceGrid

IceGrid is an implementation of an Ice location service that resolves the symbolic information in an indirect proxy to a protocol–address pair for indirect binding. A location service is only the beginning of IceGrid’s capabilities.

In addition, IceGrid:

- Allows you to register servers for automatic start-up: instead of requiring a server to be running at the time a client issues a request, IceGrid can start servers on demand, when the first client request arrives.
- Provides tools that make it easy to configure complex applications containing several servers.
- Supports replication and load-balancing.
- Automates the distribution and patching of server executables and dependent files.
- Provides a simple query service that allows clients to obtain proxies for objects they are interested in.
- Allows you to allocate Grid resources for the exclusive use of authenticated clients.

1-11-3 IceBox

IceBox is a simple application server that can orchestrate the starting and stopping of a number of application components. Application components can be deployed as a dynamic library instead of as a process. This reduces overall system load, for example, by allowing you to run several application components in a single Java virtual machine instead of having multiple processes, each with its own virtual machine.

1-11-4 IceStorm

IceStorm is a publish–subscribe service that decouples clients and servers. Fundamentally, IceStorm acts as a distribution switch for events. Publishers send events to the service, which, in turn, passes the events to subscribers. In this way, a single event published by a publisher can be sent to multiple subscribers. Events are categorized by topic, and subscribers specify the topics they are interested in. Only events that match a subscriber’s topic are sent to that subscriber. The service permits selection of a number of quality-of-service criteria to allow applications to choose the appropriate trade-off between reliability and performance.

IceStorm is particularly useful if you have a need to distribute information to large numbers of application components. (A typical example is a stock ticker application with a large number of subscribers.) IceStorm decouples the publishers of information from subscribers and takes care of the redistribution of the published events. In addition, IceStorm can be run as a federated service, that is, multiple instances of the service can be run on different machines to spread the processing load over a number of CPUs.

1-11-5 IcePatch2

IcePatch2 is a software patching service. It allows you to easily distribute software updates to clients. Clients simply connect to the IcePatch2 server and request updates for a particular application. The service automatically checks the version of the client's software and downloads any updated application components in a compressed format to conserve bandwidth. Software patches can be secured using the Glacier2 service, so only authorized clients can download software updates.

1-11-6 Glacier2

Glacier2 is the Ice firewall service: it allows clients and servers to securely communicate through a firewall without compromising security. Client-server traffic is fully encrypted using public key certificates and is bidirectional. Glacier2 offers support for mutual authentication as well as secure session management.

2 The Slice Interface Definition Language

2 The Slice Interface Definition Language

2-1 Lesson Overview

Lesson Overview

- This lesson presents:
 - the syntax and semantics of the Slice interface definition language
- Slice is an acronym for Specification Language for Ice, but is pronounced as a single syllable, to rhyme with Ice.
- By the end of this lesson, you will be able to write interface definitions in Slice and to compile these definitions into Java stubs and skeletons.

Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.2-1

Notes:

This lesson presents the syntax and semantics of the Slice interface definition language. (Slice is an acronym for *Specification Language for Ice*, but is pronounced as a single syllable, to rhyme with *Ice*.)

2-1-1 Lesson Objectives

By the end of this lesson, you will be able to write interface definitions in Slice and to compile these definitions into Java stubs and skeletons.

2-2 What is Slice?

What is Slice?

- Slice separates language-independent types from language-specific implementation.
- A compiler creates language-specific source code from Slice definitions.
- Slice is a declarative language that defines types. You cannot write executable statements in Slice.
- Slice establishes the client-server contract: data can be exchanged only if it is defined in Slice, via operations that are defined in Slice.
- Slice definitions are analogous to C++ header files: they ensure that client and server agree about the interfaces and data types they use to exchange data.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-2

Notes:

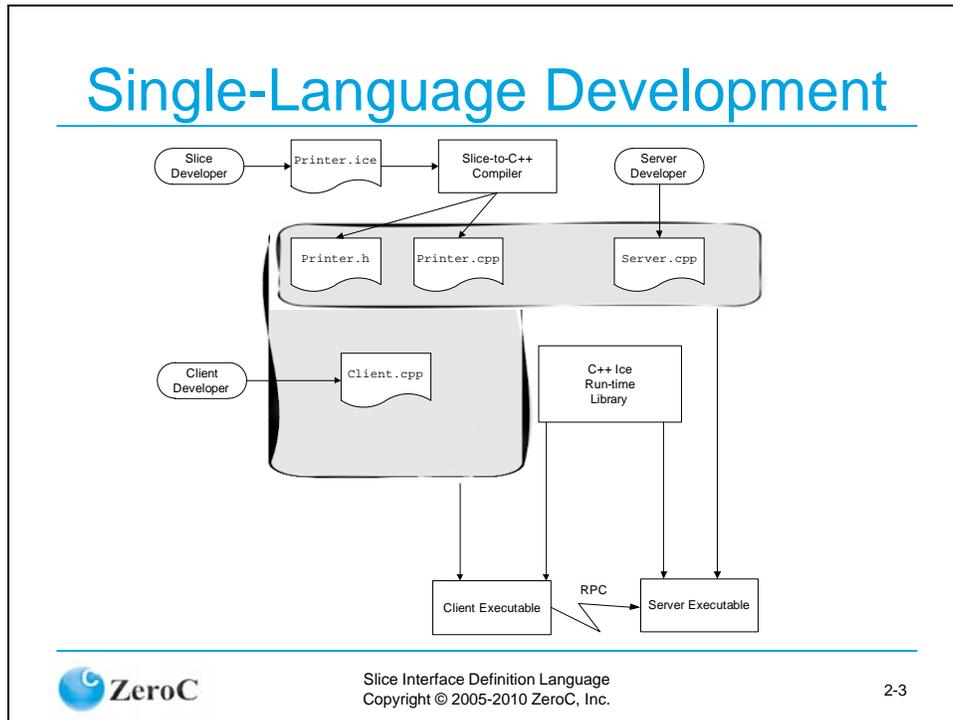
Slice is a type definition language that defines the types, interfaces, and operations that are used by clients and servers. Slice establishes the client–server contract: it is the distributed equivalent of C++ header files and serves the same purpose, namely, to ensure that client and server agree on the types they use. Slice provides the abstraction layer that separates language-independent interfaces from language-specific implementation.

A compiler compiles Slice definitions into source code for a particular implementation language, such as C++ or Java. The generated code provides the API that your application (client and server) use to interact.

If you want to exchange data between client and server, you must define corresponding types in Slice. You cannot exchange arbitrary language-native data between clients and servers because that would destroy the language independence of Ice.

Slice is a purely declarative language—there is no way to write executable statements in Slice (because Slice is an *interface* definition language, not an *implementation* definition language).

2-3 Single-Language Development

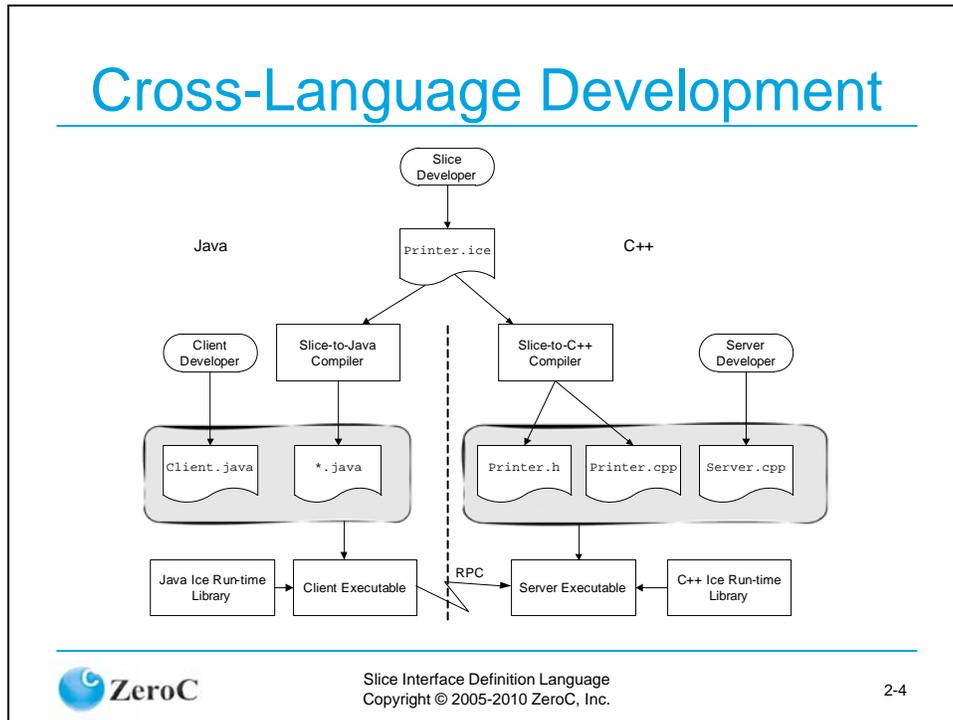


Notes:

The above diagram illustrates the development process if both client and server are written in C++.

- The compiler generates a header file that is included in the source code of both client and server. The header file defines an API for client- and server-side application code and ensures that client and server agree on the types that are defined in Slice.
- Both client and server are linked against the generated source file, which contains the code that implements the Slice-specific APIs. (The generated source also includes run-time support code, such as functions that are called by the Ice run time to marshal and unmarshal data.)
- Client and server each link against the Ice run-time library, which contains support code that is independent of the Slice types.

2-4 Cross-Language Development



Notes:

If client and server are written in different languages, the Slice definitions are compiled twice.

Example: We compile definitions once for the client to generate Java, and once for the server to generate C++. Client and server then link with their respective generated code and Ice run-time library.

It is important to note that client and server development can proceed independently. The only link between client and server is the common Slice definition.

2-5 Slice Source Files

Slice Source Files

- Slice source files must end in a `.ice` extension.
- Slice source files are preprocessed by the C++ preprocessor, so you can use `#include`, `#define`, etc.
- If you `#include` another file, the compiler parses everything, but generates code only for the including file—the included file must be compiled separately.
- Slice is a free-form language, so indentation and white space are not lexically significant (other than as token separators).
- Definitions can appear in any order, but things must be defined before they are used (or forward declared).



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-5

Notes:

Slice source files must end in a `.ice` extension. For example, `Clock.ice` is a valid file name. The compilers reject extensions other than `.ice`. For case-insensitive file systems, such as DOS, the file extension can be in uppercase or lowercase letters, so `Clock.ICE` is legal. For case-sensitive file systems, such as UNIX, `Clock.ICE` is illegal. (The extension must be in lowercase letters.)

Slice files are preprocessed in the same way as C++ source files. This means that you can use `#include` and other preprocessor features (such as macro definitions). The most common use of the preprocessor is to include another Slice definition, and to provide a double-include guard:

```
// File MyDefs.ice

#ifndef _MYDEFS_ICE
#define _MYDEFS_ICE

// Definitions here...

#endif
```

We strongly recommend that you routinely use such a guard for your definitions.

2-6 Comments and Keywords

Comments and Keywords

- Slice supports both C- and C++-style comments:


```
/*
 * This is a comment.
 */

// This comment extends to the end of this line.
```
- Slice keywords are written in lowercase (e.g. `class`), except for the keywords `Object` and `LocalObject`, which must be capitalized as shown.



Slice Interface Definition Language
 Copyright © 2005-2010 ZeroC, Inc.

2-6

Notes:

Slice allows you to use both C-style and C++-style comments.

Slice keywords must be spelled in lowercase, except for `Object` and `LocalObject`, which must be spelled as shown. The complete list of Slice keywords is:

<code>bool</code>	<code>enum</code>	<code>implements</code>	<code>module</code>	<code>struct</code>
<code>byte</code>	<code>exception</code>	<code>int</code>	<code>Object</code>	<code>throws</code>
<code>class</code>	<code>extends</code>	<code>interface</code>	<code>out</code>	<code>true</code>
<code>const</code>	<code>false</code>	<code>local</code>	<code>sequence</code>	<code>void</code>
<code>dictionary</code>	<code>float</code>	<code>LocalObject</code>	<code>short</code>	
<code>double</code>	<code>idempotent</code>	<code>long</code>	<code>string</code>	

2-7 Identifiers

Identifiers

- Identifiers consist of alphabetic characters, digits, and optionally underscores.
- Identifiers must start with an alphabetic character.
- Identifiers are case insensitive: **Foo** and **foo** cannot both be defined in the same naming scope.
- Identifiers must be capitalized consistently: once you have defined **Foo**, you must refer to it as **Foo** (not **foo** or **FOO**).
- Slice identifiers cannot begin with **I ce**.
- You *can* define identifiers that are the same as a keyword, by escaping them:

```
\d i c t i o n a r y // I d e n t i f i e r , n o t k e y w o r d
```

This mechanism exists as an escape hatch in case new keywords are added to the language over time.

- Avoid creating identifiers that are likely to be programming-language keywords, such as **functi on** or **new**.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-7

Notes:

Slice identifiers begin with an alphabetic character (A-Z and a-z), followed by any number of alphabetic characters and digits. You may also use underscores in your identifiers, but underscores may not appear consecutively, nor can they appear at the beginning or end of the identifier. A compiler option (**--underscore**) must be specified to enable the use of underscores.

To allow Slice to be mapped to languages that use case-sensitive identifiers (such as C++) as well as languages that use case-insensitive identifiers (such as Visual Basic), Slice identifiers are case-insensitive. This means that **Foo** and **foo** are considered the same identifier within a naming scope. Moreover, you must capitalize identifiers consistently throughout so, once you have defined **Foo**, you must continue to refer to it as **Foo** (not **foo** or **FOO**).

Slice reserves all identifiers beginning with **I ce** (in any capitalization). The reason for this is that language mappings require a namespace that is guaranteed to be separate from the namespace for Slice identifiers. That way, a language mapping can introduce additional identifiers into the generated code without fear of clashing with a user-defined Slice identifier.

Slice allows you create identifiers that are the same as a Slice keyword by escaping them: for example, `\dictionary` is a legal Slice identifier. However, this mechanism exists primarily to make it possible to add new keywords to Slice over time without hopelessly breaking existing applications—you should avoid using this mechanism otherwise.

If you create a Slice identifier that is a programming-language keyword, such as `new`, the language mapping will take care of mapping that identifier to something that is acceptable.

Example: Slice `new` becomes `_cpp_new` for C++, and `@new` for C#. However, escaped identifiers such as these make code harder to read, so you should avoid using Slice identifiers that are likely to be programming-language keywords. Identifiers such as `package`, `switch`, `union`, `try`, and `function` are best avoided.

2-8 Modules

Modules

A Slice file contains one or more module definitions.

```
module Example {  
    // Definitions here...  
};
```

The only definition that can appear at global scope (other than comments and preprocessor directives) is a module definition.

All other definitions must be nested inside modules.
Modules can be reopened and can be nested.

```
module Example {  
    // Some definitions here.  
};  
module Example {  
    // More definitions here...  
    module Nested { /* ... */;  
};
```

Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.2-8

Notes:

All Slice definitions (other than comments and preprocessor directives) must appear inside a module. Slice does not permit other definitions at global scope because they can be difficult to map to various languages that limit the kinds of things that can be defined at global scope (such as C# and Visual Basic).

Modules can be reopened. This is most useful if you have a large project: if you split your Slice definitions over a number of source files, all the definitions in these files can still be part of a single module. However, if you modify the definitions in one of the source files, only those parts of the application that actually include the corresponding header file need to be recompiled. In other words, reopened modules can reduce compilation time and make it easier for multiple developers to work on a single project.

Module definitions can be nested to any depth.

Modules map to the appropriate scoping construct in the various target languages. (For example, Slice modules map to Java packages and to C++ namespaces.) By using the appropriate C++ `using` or Java `import` declaration, you can avoid getting excessively long identifiers at the programming language level.

2-9 The Ice Modules

The Ice Modules

Ice uses a number of top-level modules: `Ice`, `Freeze`, `Glacier2`, `IceBox`, `IcePatch2`, `IceStorm`, and `IceGrid`.

- The `Ice` module contains definitions for basic run time features.
- The remaining modules contain definitions for specific services.

Almost all of the Ice run time APIs are defined in `Slice`. This automatically defines the API for all implementation languages.

Only a few key functions (the initialization for the run time) and a few language-specific helper functions are defined natively.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-9

Notes:

The interface to the Ice run time is defined in the `Ice` module. By defining run time APIs in `Slice`, a single definition suffices for all implementation languages. The mapping rules for each language then determine the exact shape of the API.

The various language mappings also define a small number of other APIs that are not defined in `Slice`. One such API exists to initialize the Ice run time in each language. Language mappings also may define helper functions in order to make the mapping easier to use. However, the bulk of the APIs are defined in `Slice`.

2-10 Basic Slice Types

Basic Slice Types

Slice provides a number of built-in basic types:

Type	Range of Mapped Type	Size of Mapped Type
bool	false or true	= 1 bits
byte	-128-127 or 0-255	= 8 bits
short	-2^{15} to $2^{15}-1$	= 16 bits
int	-2^{31} to $2^{31}-1$	= 32 bits
long	-2^{63} to $2^{63}-1$	= 64 bits
float	IEEE single-precision	= 32 bits
double	IEEE double-precision	= 64 bits
string	All Unicode glyphs, excluding the character with all bits zero.	Variable-length



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-10

Notes:

The built-in Slice types all are subject to change in representation as they are transmitted between clients and servers. For example, integers may undergo byte-swapping as they are exchanged between a little-endian and big-endian machine, and floating-point values may be converted to IEEE format for transmission if the native floating-point format is not IEEE. Similarly, strings may undergo representational changes. For example, they may change in size, depending on the native representation of Unicode characters.

The one type that is guaranteed not to undergo representational changes is **byte**: the bit pattern in a byte is preserved faithfully regardless of the platform. This makes **byte** suitable for transmission of binary data. (Other types such as **string** and **int** are subject to changes in representation, for example, due to changes in codeset or byte ordering.)

Slice provides the integer types **short**, **int**, and **long**, with 16-bit, 32-bit, and 64-bit ranges, respectively. Note that, on some architectures, any of these types may be mapped to a native type that is wider. Also note that no unsigned types are provided. (This choice was made because unsigned types are difficult to map into languages without native unsigned types, such as Java.)

Strings use the Unicode character set. The only character that cannot appear inside a string is the zero character. (This is a concession to standard library functions, such as `strcmp`, which become impossible to use if strings can contain embedded zero characters.) Slice does not have the concept of a null string, because this would be difficult to map to languages such as C++ (where strings are mapped to `std::string`). If you need to model an “optional” string, use the empty string to indicate the “not there” condition or, if that is impossible because the empty string is a valid string, refer to the technique shown in Section 2-13.

2-11 Enumerations

Enumerations

Enumerations are much like their Java counterpart:

```
enum Fruit { Apple, Pear, Orange };
```

You cannot specify the value of the enumerators:

```
enum Fruit { Apple=0, Pear=7, Orange=2 }; // Illegal!
```

As for C++ (and unlike Java), enumerators enter the namespace enclosing the enumeration:

```
enum Fruit { Apple, Pear, Orange };  
enum ComputerBrands { Apple, IBM, Sun, HP }; // Error!  
Empty enumerations are illegal.
```



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-11

Notes:

Enumerations work like their Java counterparts, but do not permit you to define the value of the enumerators, as a concession to languages without support for this feature.

2-12 Structures

Structures

Structures contain at least one member of arbitrary type:

```
struct TimeOfDay {
    short hour;           // 0-23
    short minute;        // 0-59
    short second;        // 0-59
};
```

The name of the structure, `TimeOfDay`, becomes a type name in its own right. (There are no typedefs in Slice.)

Structures form a namespace, the member names must be unique only within their enclosing structure.

Members may optionally declare a default value.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-12

Notes:

Structures must have at least one member of arbitrary type. Structures do not support inheritance, and have no reference concept—they are intended to be used as simple groups of fields. If you need inheritance or references, use classes. (Refer to Section 2-28.)

Structure definitions cannot be nested:

```
struct TwoPoints {
    struct Point { // Illegal!
        short x;
        short y;
    };

    Point coord1;
    Point coord2;
};
```

This is true for Slice in general: type definitions cannot be nested. You can achieve the equivalent effect (and more cleanly) by writing:

```
struct Point {
    short x;
    short y;
};

struct TwoPoints { // Legal (and cleaner!)
    Point coord1;
    Point coord2;
};
```

By default, the language-specific code generated for a Slice structure does not initialize the members, meaning they are initialized using whatever semantics the native language provides. In Java, all members are initialized to a zero or null value.

You can ensure that members are initialized to different default values by defining them in Slice:

```
struct Point {
    short x = 1;
    short y = 1;
};
```

The legal syntax for default values is the same as for constants. (Refer to Section 2-15.)

2-13 Sequences

Sequences

Sequences are (possibly empty) variable-length collections:

```
sequence<Fruit> FruitPlatter;
```

The element type can be anything, including another sequence type:

```
sequence<FruitPlatter> FruitBanquet;
```

The order of elements is never changed during transmission; sequences are ordered collections.

Use sequences to model collections, such as sets, lists, arrays, bags, queues, and trees.

Use sequences to model optional values:

```
sequence<string> InitialOpt;
```

```
struct Person {
    string      firstName;
    InitialOpt  initial;
    string      lastname;
};
```



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-13

Notes:

Sequences are variable-length collections of elements. It is valid for a sequence to be empty, that is, to contain no elements. The element type can be a built-in or user-defined type, including another sequence. Sequences can be used to model sets, lists, arrays, bags, stacks, queues, trees. (Trees are often modeled with nested sequences.)

Sequences are the preferred way to model optional values, and their use for this purpose is idiomatic.

You can use classes (refer to Section 2-28) to model optional values, but sequences are more efficient to marshal and therefore the preferred option.

2-14 Dictionaries

Dictionaries

A dictionary is a map of key-value pairs:

```
struct Employee {
    long number;
    string firstName;
    string lastName;
};
dictionary<long, Employee> EmployeeMap;
```

Use dictionaries to model maps and sparse arrays.

Dictionaries map to efficient lookup data structures, such as STL maps or hash tables.

The key type of a dictionary must be one of:

- An integral type (**bool**, **byte**, **short**, **int**, **long**, **enum**) or **string**
- A structure containing only members of integral type or type **string**



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-14

Notes:

Dictionaries are maps of name–value pairs. They map to efficient data structures in the respective target languages, such as STL maps or hash tables.

The key type of dictionaries is restricted to avoid complexities in language mappings, and to avoid the vagaries of floating-point representation.

2-15 Constants and Literals

Constants and Literals

Slice permits constants of type:

- `bool`, `byte`, `short`, `int`, `long`
- enumerated type
- `float` and `double`
- `string`

Examples:

```
const bool      AppendByDefault = true;
const byte     LowerNibble = 0x0f;
const string   Advice = "Don't Panic!";
const short    TheAnswer = 42;
const double   PI = 3.1416;
```

```
enum Fruit { Apple, Pear, Orange };
const Fruit FavoriteFruit = Pear;
```

Slice does not support constant expressions.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-15

Notes:

Constants are initialized by literals. The syntax for literals is similar to C++, with some minor restrictions:

1. Numeric and enumerated literals must have a value within the range of the constant type. There is no type casting of any kind, and the compiler will emit a diagnostic if a value is out of range.
2. `bool` constants must be initialized with `true` or `false`. (You cannot use `1` and `0`.)
3. You can initialize integer constants using decimal, octal, or hexadecimal notation, so `42`, `052`, `0X2A`, and `0x2a` all mean the same value: 42 decimal.

For `short`, `int`, and `long`, the value is always interpreted as a positive number. For example, to get the value `-32768` into a short, you must use a unary minus: `-0x8000`, not `0xffff`.

For byte values, the value is always interpreted as a bit pattern, and the value must be in the range `0-255` (but can be specified as an octal or hexadecimal literal).

The suffixes `u`, `U`, `l`, and `L` (used by C++ to indicate unsigned and long values) are illegal.

4. Floating-point constants use the same syntax as C++, except that the **l** and **L** suffixes to indicate an extended floating-point value are illegal. The **f** and **F** suffixes for single-precision floating-point values are legal, but are ignored. (The type of a constant is determined by its formal declared type, not by the type of the literal.)

Here are a few examples:

```
const float P1 = -3.14f;    // Integer & fraction, with suffix
const float P2 = +3.1e-3;  // Integer, fraction, and exponent
const float P3 = .1;       // Fraction part only
const float P4 = 1.;       // Integer part only
const float P5 = .9E5;     // Fraction part and exponent
const float P6 = 5e2;      // Integer part and exponent
```

5. Strings support the same escape sequences as C++. Here are a few examples:

```
const string AnOrdinaryString = "Hello World!";

const string DoubleQuote =    "\"";
const string TwoSingleQuotes = "'\''";    // ' and \' are OK
const string Newline =       "\n";
const string CarriageReturn = "\r";
const string HorizontalTab = "\t";
const string VerticalTab =   "\v";
const string FormFeed =     "\f";
const string Alert =        "\a";
const string Backspace =    "\b";
const string QuestionMark = "\?";
const string Backslash =    "\\";

const string OctalEscape =   "\007";    // Same as \a
const string HexEscape =    "\x07";    // Ditto
```

```
const string UniversalCharName = "\u03A9"; // Greek Omega
```

Adjacent string literals are concatenated, as for C++:

```
const string MSG1 = "Hello World!";
const string MSG2 = "Hello" " " "World!";    // Same message

/*
 * Escape sequences are processed before concatenation,
 * so the string below contains two characters,
 * '\xa' and 'c'.
 */
const string S = "\xa" "c";
```

As referenced in Section 2-10, Slice has no concept of a null string, so the following is illegal:

```
const string nullString = 0;    // Illegal!
```

2-16 Interfaces

Interfaces

Interfaces define object types:

```
struct TimeOfDay { /* ... */ };

interface Clock {
    TimeOfDay getTime();
    void setTime(TimeOfDay time);
};
```

- Interfaces define the public interface of an object. There is no notion of a private part of an object in Slice.
- Interfaces only have operations, not data members. (Data members are implementation state, not interface.)
- Invoking an operation on an interface sends a (possibly remote) invocation (RPC) to the target object.
- Interfaces define the smallest and only granularity of distribution: if something does not have an interface (or Slice class, which is also an interface), it cannot be invoked remotely.



Slice Interface Definition Language
 Copyright © 2005-2010 ZeroC, Inc.

2-16

Notes:

The central focus in Slice is on defining interfaces. The above interface defines two operations. Clients invoke these operations via a proxy to an object of type **Clock**. The proxy encapsulates the identity of the target object (that is, denotes a specific instance of **Clock**). Invoking an operation on the instance sends an RPC from the client to the instance. The target object can be on a different machine than the client, in a process on the same machine as the client, or even inside the client's own address space—the Ice run time ensures that such differences are hidden.

Interfaces only contain operations, but not data members. The only thing that is invocable remotely is an operation, so it does not make sense to have data members in interfaces, because data members are about implementation, not interface. For the same reason, everything in an interface is public. You keep things private by simply not mentioning them.

A Slice interface defines the smallest grain of distribution in Ice: each Ice object has a unique identity (encapsulated in its proxy) that distinguishes it from all other Ice objects; for communication to take place, you must invoke operations on an object's proxy. There is no other notion of an addressable entity in Ice.

The partition of an application into interfaces therefore has profound influence on the overall architecture. Distribution boundaries must follow interface (or class) boundaries; you can spread the implementation of interfaces over multiple address spaces (and you can implement multiple interfaces in the same address space), but you cannot implement parts of interfaces in different address spaces.

2-17 Operations and Parameters

Operations and Parameters

An interface contains zero or more operation definitions.

Each operation definition has:

- an operation name
- a return type (or **void** if none)
- zero or more parameters
- an optional **idempotent** modifier
- an optional exception specification

If an operation has **out**-parameters, they must follow in-parameters.

Operations cannot be overloaded.

```
interface Example {
    void op();
    int otherOp(string p1, out string p2);
};
```



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-17

Notes:

An interface contains zero or more operation definitions. (Interfaces can be empty.) The operations of an interface must have different names, that is, operation overloading is not supported (due to the difficulty of mapping overloading into some languages).

An operation can return a result of any type; if no result is returned by an operation, it must have **void** return type.

An operation can have zero or more parameters. Parameters come in two flavors:

- **in**-parameters
In-parameters are values that travel from client to server, that is, the client initializes them, and the server receives them.
- **out**-parameters
out-parameters are values that travel from server to client, that is, the server initializes them and the client receives them.

An in-parameter cannot follow an **out**-parameter in an operation definition:

```
interface BadExample {
    int otherOp(out string p1, string p2); // Error!
};
```

2-18 idempotent Operations

i dempote nt Operations

An **i dempote nt** operation is an operation that, if invoked twice, has the same effect as if it is invoked once:

```
i dempote nt void setName(string name);  
i dempote nt string getName();
```

The **i dempote nt** keyword affects the error-recovery behavior of the Ice run time: for normal operations, the run time has to be more conservative to preserve at-most-once semantics.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-18

Notes:

You can use an **i dempote nt** modifier on operation definitions. An operation is idempotent if it meets one of the following criteria:

- it is (conceptually) a read-only operation, that is, it does not visibly modify state in the server, or
- if two successive invocations of the operation (with the same parameters) have the same effect as a single invocation.

In other words, idempotent operations have read-only or assignment semantics. For example, **x = 1;** is an idempotent statement, whereas **x++;** is not.

The Ice run time uses more aggressive error recovery for operations that are idempotent because, for such operations, there is no need to maintain at-most-once semantics: at-most-once semantics guarantee that an operation is either invoked or not invoked; in no case can a single invocation in a client result in more than one invocation in the server.

For normal operations (which potentially make state changes in the server), at-most-once semantics are preserved, which requires the Ice run time to be conservative under certain error conditions: the run time cannot retry a failed request if there is a possibility the request was received by the server and only the reply to the request was lost. On the other hand, for idempotent operations, the Ice run time can retry a request even if it is not known whether the previous attempt was received by the server because retrying such operations is harmless.

2-19 User Exceptions

User Exceptions

Operations can throw exceptions:

```
exception Error {}; // Empty exceptions are legal
exception RangeError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};
interface Clock {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time)
        throws RangeError, Error;
};
```

Operations *must* declare the exceptions they can throw in the exception specification.

Exceptions are *not* data types: they cannot be used as data members or parameters.



Slice Interface Definition Language
 Copyright © 2005-2010 ZeroC, Inc.

2-19

Notes:

Operations can throw user-defined exceptions. If an operation throws user exceptions, it *must* declare those exceptions in its exception specification. Exceptions can contain any user-defined type, and can be empty.

Exceptions themselves are *not* types: you cannot pass them as parameters, and you cannot use them as the member of a structure or class, as a sequence element, or the key or value type of a dictionary.

You cannot throw built-in types (such as `int` or `string`) as exceptions.¹

Exception members can define default values, as with structures (Refer to Section 2-12).

¹ This is a concession to language mappings; C++ is unusual in that it allows any type to be thrown as an exception—other languages, such as Java and C#, are much more restrictive.

2-20 Exception Inheritance

Exception Inheritance

Exceptions can form single-inheritance hierarchies:

```
exception ErrorBase {
    string reason;
};
enum RError {
    DivideByZero, NegativeRoot, IllegalNull /* ... */
};
exception RuntimeError extends ErrorBase {
    RError err;
};
```

An operation that specifies a base exception in its exception specification can throw the base exception and any exceptions derived from the base:

```
void op() throws ErrorBase; // Can throw RuntimeError
```

Derived exceptions cannot redefine data members defined in a base.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-20

Notes:

You can set up single-inheritance hierarchies for exceptions. This allows you to classify exceptions and handle them at different levels of abstraction in the code. (The standard libraries of many programming languages use this technique.)

If an operation's exception specification mentions a base exception, this implies that, at run time, the operation can throw the base exception and any exceptions derived from the base exception (in keeping with the *is-a* meaning of inheritance).

A derived exception cannot have a data member with the same name as a data member in one of its base exceptions:

```
exception Base {
    int i;
};

exception Intermediate extends Base {
    int j;
};

exception Derived extends Intermediate {
    int i; // Error!
};
```

If an operation throws a derived exception, but the client only has knowledge of the base type (for example, because the derived exception was added after the client was deployed), the client receives the base part of the exception, that is, the Ice run time slices the exception to the most-derived part that is understood by the receiver. (This is analogous to catching C++ exceptions by value: the exception is sliced to the formal type used in the `catch` clause.)

2-21 Ice Run-Time Exceptions

Ice Run-Time Exceptions

Any operation (whether it has an exception specification or not) can always throw Ice run-time exceptions.

Run-time exceptions capture common error conditions, such as out of memory, connect timeout, etc.

Three exceptions have special meaning:

- **UnknownException**
The operation in the server has thrown a non-Ice exception (such as `java.lang.ClassCastException`).
- **UnknownUserException**
The operation has thrown an exception that is not in its exception specification.
- **UnknownLocalException**
The operation on the server-side has thrown a run-time exception that is not marshaled back to the client. (See next slide.)



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-21

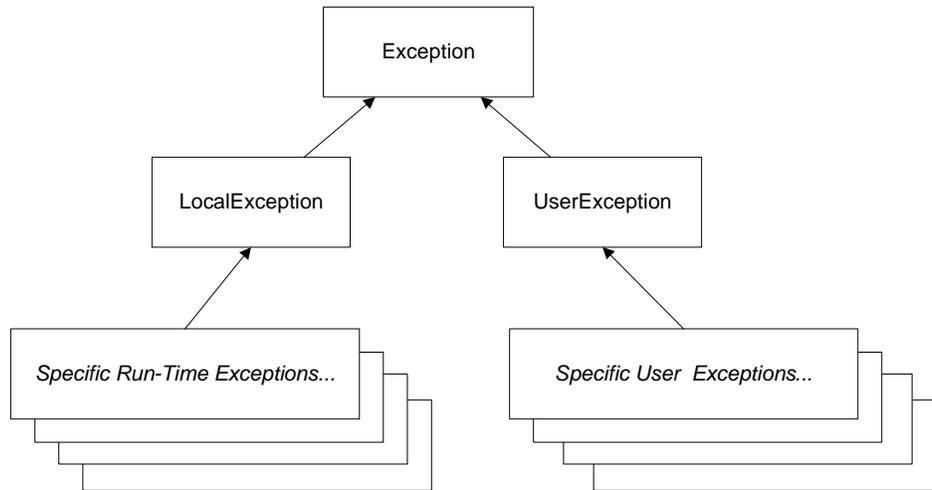
Notes:

An operation can throw an Ice run-time exception at any time. Ice run-time exceptions capture common error conditions, such as out of memory, connection loss, timeouts, and so on.

You cannot specify an Ice run-time exception in an operation's exception specification. (In effect, every operation has an implicit exception specification for all Ice run-time exceptions.)

If an operation implementation throws a native exception (such as `ClassCastException`), or throws an Ice run-time exception that is not known to the client, the client-side Ice run time delivers `UnknownException` to the client. (A client can receive an unknown Ice run-time exception from a server if the server uses a later version of the Ice run time than the client.)

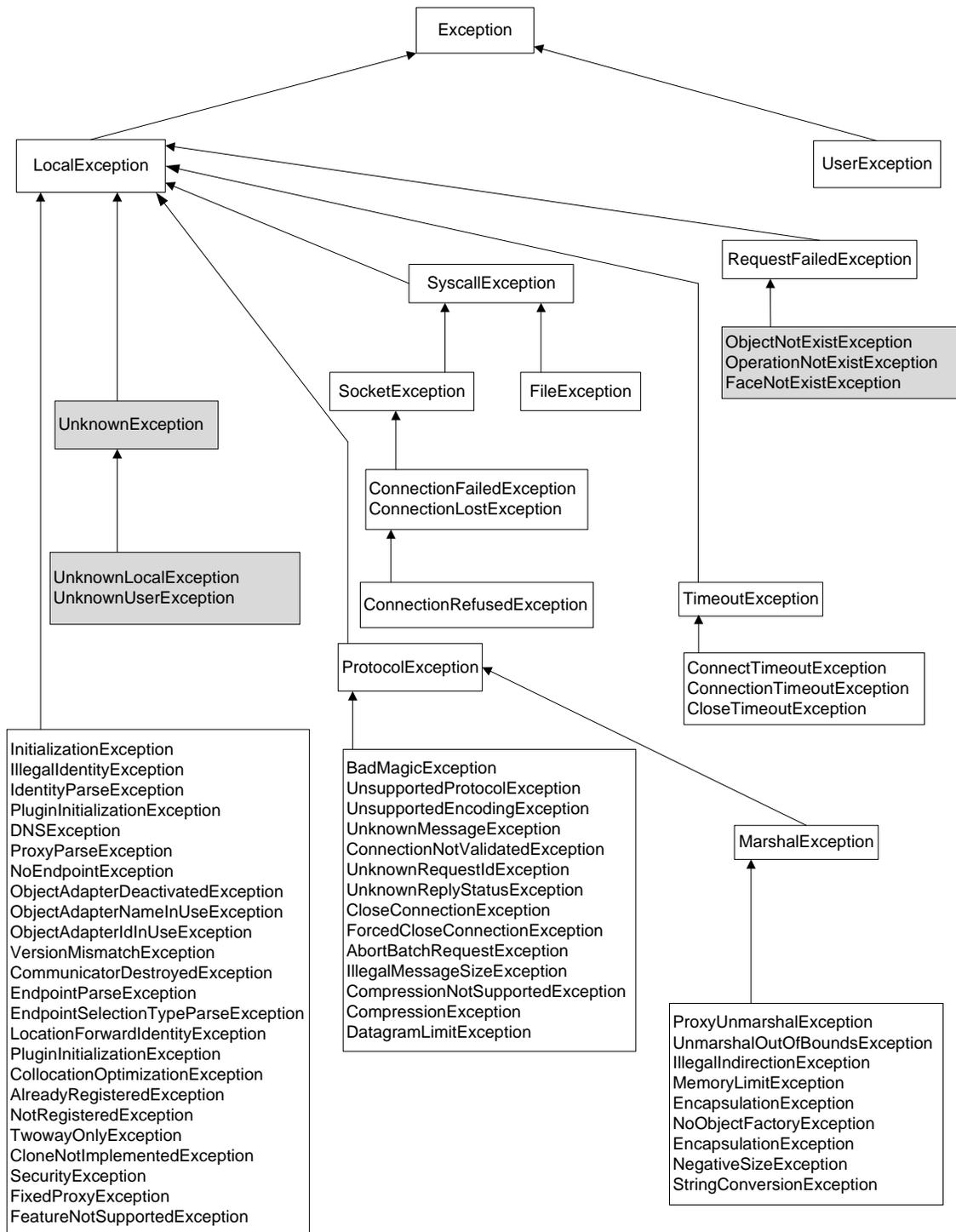
The Ice run-time exceptions are arranged into a hierarchy as follows:



All user exceptions implicitly inherit from `Ice::UserException`, and all Ice run-time exceptions implicitly inherit from `Ice::LocalException`.

You can use this at the language-mapping level to catch all user exception, all Ice run-time exceptions, or all Ice exceptions with a single exception handler.

The complete hierarchy of Ice run-time exceptions is as follows:



The shaded exceptions are the *only* exceptions that are ever received remotely. If an operation raises any of the unshaded exceptions, you know that the exception was raised by the local run time.

2-22 Run-Time Exceptions Raised by the Server

Run-Time Exceptions Raised by the Server

There are three exceptions that can be received from the remote end:

- **ObjectNotExistException**

The client has called an operation via a proxy that denotes a servant that does not exist. Most likely cause: the object existed in the past but has since been destroyed.

- **OperationNotSupportedException**

The client has invoked an operation that the target object does not support. Most likely cause: client and server were compiled with mismatched Slice definitions.

- **FacetNotExistException**

The client has called an operation via a proxy that denotes an existing object, but the specified facet does not exist. Most likely cause: the client specified an incorrect facet name, or the facet existed in the past but has since been destroyed.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-22

Notes:

Apart from **UnknownException**, **UnknownLocalException**, and **UnknownUserException**, there are only three other exceptions that can be received from the remote end:

- **ObjectNotExistException**

This exception is raised if a client invokes an operation via a proxy that denotes a servant that does not exist: the operation was successfully sent to the server, and the server-side run time consulted with the server-side application code; it is the application code that ultimately decides whether an object exists or not. If you receive an **ObjectNotExistException**, it usually means that the Ice object does not exist. Depending on how you have written the server, it may also mean that the object may never exist in the future; however, the server can create semantics such that the object can come into existence later. Whether this can happen depends on whether the server allows “resurrecting” an object, that is, whether it re-uses the object ID of a previous object for a new one. (Refer to Section 12-9.)

- **OperationNotExistException**

This exception is raised if the client invokes a non-existent operation on an object. This can happen only if client and server have been compiled with mismatched Slice definitions (which should never happen), or if the client uses dynamic invocation. (See the Ice manual for more details on dynamic invocation.)

- **FacetNotExistException**

This exception is raised if the client invokes an operation on an existing object, but on a facet of that object that does not exist. (See the Ice manual for more details on facets.)

2-23 Proxies

Proxies

Proxies are the distributed equivalent of class pointers or references:

```
Interface Clock { /* ... */ };
dictionary<string, Clock*> TimeMap; // Time zones
exception BadZoneName { /* ... */ };
Interface WorldTime {
  idempotent Clock* findZone(string zoneName) throws BadZoneName;
  idempotent TimeMap listZones();
};
```

The * operator is known as the *proxy operator*.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-23

Notes:

Proxies are the distributed equivalent of pointers: proxies denote objects that can be in a remote address space or in the local address space. Proxies act as the local ambassador for a remote object: invocations on the proxy are sent to the remote object. Passing a proxy as a parameter passes the proxy, not the object!

Note: You must use the * proxy operator to define a proxy type. (Leaving the * operator out has a different meaning, which we examine in Section 2-29.)

Proxies, apart from their ability to point to a remote address space, have the same semantics as C++ pointers:

- Proxies can dangle, that is, point at a non-existent object. Contrary to C++ dangling pointers, invoking via a dangling proxy is safe: instead of crashing, you get an exception. (Typically, `ObjectNotExistException`, `ConnectFailedException`, or `ConnectionRefusedException`, depending on whether the server is running (but the object does not exist), the machine on which the server is supposed to be is down or unreachable, or the machine on which the server is running is reachable, but the server is down, respectively.)

- Proxies can point nowhere. A dedicated null value is provided by all language mappings to indicate that a proxy points at no object. (This value can be marshaled just like any other proxy.)
- Calling an operation on a proxy always uses late binding: if the proxy's actual object is more derived than the proxy's formal type, the implementation in the derived object is invoked, not in the base object.

2-24 Interface Inheritance

Interface Inheritance

Interfaces support inheritance:

```
interface AlarmClock extends Clock {
    TimeOfDay getAlarmTime();
    void setAlarmTime(TimeOfDay alarmTime)
        throws BadTimeVal;
};
```

Multiple inheritance is legal as well:

```
interface Radio {
    void setFrequency(long hertz) throws GenericError;
    void setVolume(long dB) throws GenericError;
};

enum AlarmMode { RadioAlarm, BeepAlarm };

interface RadioClock extends Radio, AlarmClock {
    void setMode(AlarmMode mode);
    AlarmMode getMode();
};
```



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

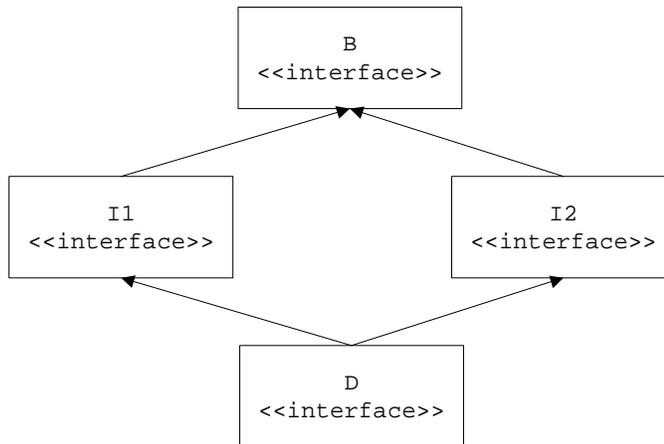
2-24

Notes:

Interfaces support single and multiple inheritance. The semantics are as for C++ and Java: you can pass a proxy to a derived object where a proxy to a base object is expected. With multiple inheritance, a derived interface can inherit the same base interface via more than one path:

```
interface B { /* ... */ };
interface I1 extends B { /* ... */ };
interface I2 extends B { /* ... */ };
interface D extends I1, I2 { /* ... */ };
```

This results in the familiar diamond shape:



2-25 Interface Inheritance Limitations

Interface Inheritance Limitations

An interface cannot inherit an operation with the same name from more than one base interface:

```
interface Clock {
    void set(TimeOfDay time); // set time
};

interface Radio {
    void set(long hertz); // set frequency
};

interface RadioClock extends Radio, Clock { // Illegal!
    // ...
};
```

There is no concept of overriding or overloading.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-25

Notes:

An operation name cannot be inherited from more than one base interface. (Allowing this would complicate some language mappings considerably.)

Once an interface has inherited an operation, you must not re-define the operation in a derived interface, even if the signature is identical:

```
interface Base {
    void op();
};

interface Derived extends Base {
    void op(); // Illegal!
};
```

It is understood that the derived interface has an operation `op`; you are not allowed to restate that.

2-26 Implicit Inheritance from Object

Implicit Inheritance from Object

All interfaces implicitly inherit from `Object`, which is the root of the inheritance hierarchy.

```
interface ProxyStore {
    void putProxy(string name, Object* o);
    Object* getProxy(string name);
};
```

Because any proxy is assignment compatible with `Object`, `ProxyStore` can store and return proxies for any interface type.

Explicit inheritance from `Object` is illegal:

```
interface Wrong extends Object { // Error!
    // ...
};
```

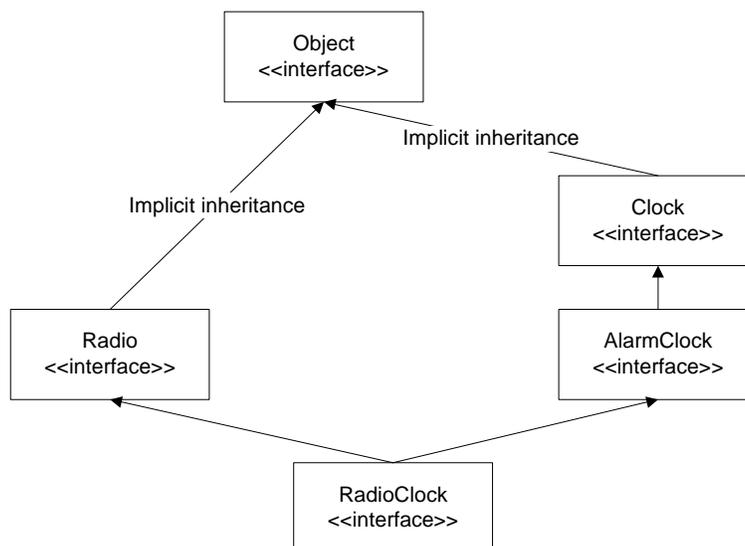


Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-26

Notes:

All interfaces implicitly inherit from `Object`, which is the root of the inheritance tree. For example, the inheritance for the `RadioClock` we defined in Section 2-24 really looks as follows:



Note: You cannot explicitly inherit from `Object`.

2-27 Self-Referential Interfaces & Forward Declarations

Self-Referential Interfaces & Forward Declarations

Interfaces can be self-referential:

```
interface Node {
    int val();
    Node* next();
};
```

You can forward-declare an interface to create interfaces that mutually refer to each other:

```
interface Wife; // Forward declaration

interface Husband {
    Wife* getWife();
};

interface Wife {
    Husband* getHusband();
};
```



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-27

Notes:

The name of an interface is defined once the opening curly brace is parsed, so an interface can refer to itself, as in the case of **Node** above.

You can use a forward declaration to create interfaces that mutually refer to each other.

You cannot derive from an interface until after it has been defined:

```
interface Base; // Forward declaration
```

```
interface Derived1 extends Base {}; // Error!
```

```
interface Base {}; // Definition
```

```
interface Derived2 extends Base {}; // OK, definition was seen
```

2-28 Classes

Classes

Classes can contain data members as well as operations.

Classes support single implementation and multiple interface inheritance. They implicitly derive from **Object** (just like interfaces).

One way to use classes is as structures that are extensible by inheritance:

```
class TimeOfDay {
    short hour;          // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

class DateTime extends TimeOfDay {
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
};
```

Empty classes are legal.

Data members may define default values, as with structures.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-28

Notes:

Classes, at their most basic, are structures that support single implementation inheritance. Multiple implementation inheritance is illegal:

```
class TimeOfDay {
    short hour;          // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

class Date {
    short day;
    short month;
    short year;
};

class DateTime extends TimeOfDay, Date { // Error!
    // ...
};
```

Class members can define default values, as with structures. (Refer to Section 2-12.)

2-29 Passing Classes as Parameters and Slicing

Classes as Parameters and Slicing

Classes are passed *by value*, just like structures.

You can pass a derived class where a base is expected:

```
interface Clock {
    void setTime(TimeOfDay t);
};
```

You can pass a `TimeOfDay` instance or a `DateTime` instance to `setTime`.

The receiver gets the most-derived type that it has static type knowledge of:

- If the server was linked with the stubs for both `TimeOfDay` and `DateTime`, the server receives a `DateTime` instance (as the static type `TimeOfDay`).
- If the server was linked with the stubs for only `TimeOfDay`, the `DateTime` object is sliced to `TimeOfDay` in the server.

Use classes if you need polymorphic *values* (instead of *interfaces*).



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-29

Notes:

Classes are passed by value, just like structures. Classes are also polymorphic: you can pass a derived instance where a base instance is expected. Provided that the receiver of a derived instance that is passed as a base knows the derived type, it receives the derived instance; otherwise, the derived instance is sliced to the most-derived type that is known to the receiver.

The semantics of class slicing are similar to passing a C++ class by value. However, in C++, passing a C++ class by value *always* results in the instance being sliced to the formal parameter type. On the other hand, in Ice, slicing occurs only if the receiver does not have static type knowledge of the actual run-time type of an instance. In other words, Ice preserves the run-time type of a class whenever possible and applies slicing only if it has no other choice.

Classes are useful if you need polymorphic values instead of interfaces:

```
class Shape { /* ... */ };

class Circle extends Shape { /* ... */ };

class Rectangle extends Shape { /* ... */ };

sequence<Shape> ShapeSeq;

interface ShapeProcessor {
    void processShapes(ShapeSeq ss);
};
```

Note: `processShapes` can accept any of the types derived from `Shape`. Language mappings provide a dynamic cast that allows the receiver to safely down-cast a `Shape` into its actual derived type.

2-30 Classes as Unions

Classes as Unions

You can use derivation from a common base class to model unions. It is often useful to include a discriminator in the base class, so the receiver can use a `switch` statement to find which member is active (instead of an `if-then-else` chain of dynamic casts).

```
class UnionDiscriminator {
    int d;
};
class Member1 extends UnionDiscriminator {
    // d == 1
    string s;
};
class Member2 extends UnionDiscriminator {
    // d == 2
    double d;
};
```



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-30

Notes:

You can use classes to model unions. As a rule, you should include the union discriminator in the base class. This not only makes it more explicit what is going on, but also allows the receiver of a union to more efficiently determine the active member because it can use a `switch` statement on the union discriminator instead of having to use an `if-then-else` chain of dynamic casts.

Note: You also can use classes to model optional parameters, for example:

```
class Empty {};
```

```
class StringOpt extends Empty {
    string s;
};
```

However, we recommend that you use the approach shown in Section 2-13 instead—it is both idiomatic and more efficient at run time.

2-31 Self-Referential Classes

Self-Referential Classes

Like interfaces, classes can be self-referential:

```
class Link {
    SomeType value;
    Link next; // Note: NOT Link* !
};
```

This looks like `Link` includes itself but really means that `next` contains a pointer to another `Link` instance that is in the same address space.

Passing an instance of `Link` as a parameter passes *the entire chain* of instances to the receiver.

You can use self-referential classes to model arbitrary graphs.

Passing a node of the graph as a parameter marshals the entire graph that is reachable using that node as a starting point.

Cyclic graphs are permitted, as are graphs with nodes of in-degree > 1.

Forward declarations are legal (with the same syntax as for interfaces).



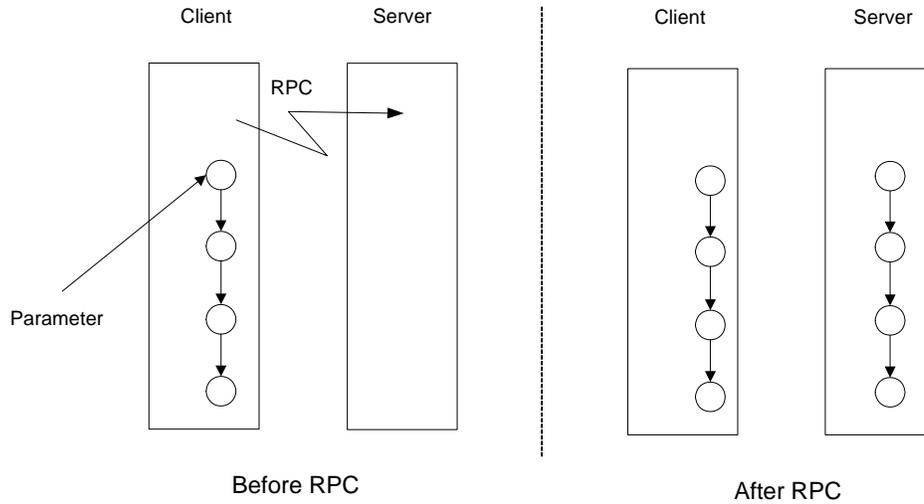
Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-31

Notes:

Classes can be self-referential. A class that “includes itself” really is a class that contains a pointer (or reference) to another instance of the same type. The pointer *always* points at another instance in the *same* address space. (Language mappings provide a null value, so you can make a pointer point at no instance.)

If you pass an instance of a class as a parameter, *all* instances that are reachable via pointers in the passed instance are marshaled. (The run time ensures that cyclic graphs do not cause problems.) For example, passing a four-element chain of `Link` instances marshals the entire chain:



Identity relationships are correctly preserved across RPC calls. For example, if nodes have an in-degree greater than one, the graph is re-created on the receiving side exactly as it was sent; this holds true even if different in-parameters point at the same node or different nodes in a graph.

You can forward-declare a class using the same syntax as to forward-declare an interface.

2-32 Classes with Operations

Classes with Operations

```
class TimeOfDay {
    short hour;        // 0 - 23
    short minute;     // 0 - 59
    short second;     // 0 - 59

    string format();
};
```

Classes with operations are mapped to abstract base classes with abstract methods.

The application provides the implementation for the operations.

Invoking an operation on a class invokes the operation in the local address space of the class.

It follows that, if a class with operations is sent as a parameter, the code for the operation must exist at the receiving end. The Ice run time only marshals the data, not the code.

Classes with operations allow you to implement client-side processing.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-32

Notes:

Classes can have operations. The syntax is the same as for operations on interfaces.

At the language-mapping level, classes with operations become abstract base classes, and the application code defines a derived class that contains implementations for the operations.

Invoking an operation on a class runs the operation on the local class instance, that is, invocations on classes are always local.

Classes are useful to implement client-side processing. If a client invokes on a proxy, the invocation goes remote, back to the server in which the object is implemented, and so incurs RPC overhead. In contrast, if a client invokes on a class by value (instead of by proxy) the invocation stays local and executes inside the client's local copy of the class, so there is no overhead. However, if a class has operations, both the sender and the receiver must have an implementation of the class' operations—the Ice run time does not send code from sender to receiver, only the data members.

2-33 Classes Implementing Interfaces

Classes Implementing Interfaces

```

Interface Time {
    Idempotent TimeOfDay getTime();
    Idempotent void setTime(TimeOfDay time);
};

Interface Radio {
    Idempotent void setFrequency(Long hertz);
    Idempotent void setVolume(Long dB);
};

class RadioClock implements Time, Radio {
    TimeOfDay time;
    Long hertz;
};

```

Classes can implement one or more interfaces (in addition to extending a single other class).

The derived class inherits all of the operations of its base interface(s).



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-33

Notes:

A class can implement one or more interfaces. (The class can add data members and operations of its own.)

A class that implements an interface can act as the servant of an interface (that is, the class can provide the behavior in a server of remotely callable operations on an interface).

If a class uses both implementation and interface inheritance, the implementation inheritance must be listed first:

```

interface Time { /* ... */ };

class Clock implements Time { /* ... */ };

interface AlarmClock extends Time { /* ... */ };

interface Radio { /* ... */ };

class RadioAlarmClock
    extends Clock
    implements AlarmClock, Radio {
    // ...
};

```

2-34 Class Inheritance Limitations

Class Inheritance Limitations

Operations and data members must be unique within a hierarchy:

```
interface BaseInterface {
    void op();
};

class BaseClass {
    int member;
};

class DerivedClass
    extends BaseClass
    implements BaseInterface {
    void someOperation(); // OK
    int op(); // Error!
    int someMember; // OK
    long member; // Error!
};
```

As for interfaces, you cannot inherit the same operation from different base interfaces.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-34

Notes:

Classes cannot inherit the same operation name from more than one base class or interface. In addition, classes cannot redefine a data member inherited from a base class. (Both restrictions exist to simplify language mappings.)

2-35 Pass-by-Value Versus Pass-by-Reference

Pass-by-Value Versus Pass-by-Reference

You can create proxies to classes:

```
class TimeOfDay { /* ... */ };

interface Clock {
    TimeOfDay getTime(); // Returns class
    TimeOfDay* getTimeProxy(); // Returns proxy
};
```

Invoking an operation on a *class* invokes the operation locally.

Invoking an operation on a *proxy* invokes the operation remotely.

Only operations (but not data members) of a class are accessible via its proxy.

You can also pass an interface by value:

```
interface Time { /* ... */ };

interface Clock {
    void set(Time t); // Note: NOT Time* !
};
```



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-35

Notes:

Classes have an interface. For a class that does not derive from anything, the interface of the class is its operations. For a derived class, its interface is all the operations of the class and its base class and/or base interfaces.

The proxy for a class is like the proxy for an interface: invocations on the proxy are sent to the class via an RPC, that is, invocations via proxy are non-local, whereas invocations on the class itself are local.

You can pass an interface by value as well as by proxy. If you do this, you are effectively stating that any class derived from that interface can be passed.

2-36 Architectural Implications of Classes

Architectural Implications of Classes

- Classes enable client-side processing and avoid RPC overhead.
- The price is that the behavior of (that is, the code for) class operations must be available wherever the class is used.
- If you have a C++ class with operations, and want to use it from a Java client, you must re-implement the operations of the class in Java, with identical semantics.
- Classes with operations destroy language- and OS-transparency (if they are passed by value).
- Use classes with operations only if you can control the deployment environment for the entire application!



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-36

Notes:

The operations of a class effectively use client-side native code: the behavior of a class must be implemented wherever the class is used. If clients are written in different languages and run on different operating systems, you must provide *semantically identical* implementations of the operations for all combinations of language, compiler, and OS that are used by clients.

Doing this obviously can be quite a lot of work and, because clients need to run native code, classes with operations effectively destroy the language- and operating system transparency of an application. This does not mean that classes with operations are automatically bad. For example, if you have operations on classes, but do not pass them by value (only using them to implement servants), there are no problems. But do exercise caution if you *do* pass classes with operations by value: unless you can control the client-side deployment for the application, you may be better off forgoing operations on classes.

Note: Classes without operations do not suffer any of these problems: because no client-side code is required, no language- or operating system transparency is lost; classes without operations are equivalent to structures that support inheritance and pointer semantics.

2-37 Classes Versus Structures

Classes Versus Structures

Classes can model structures, so why have structures?

Structures are more efficient because they can be stack-allocated whereas classes are always heap-allocated.

Classes are slower to marshal than structures, and consume more bandwidth on the wire.

Use classes if you need one or more features not provided by structures:

- inheritance
- pointer semantics
- client-side local operations
- choice of local versus remote invocation



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-37

Notes:

If you are doubtful about whether to use a class or a structure in a particular situation, use the simple rule of thumb that a structure is preferable. Use a class only when you cannot model what you need with a structure.

2-38 The :: Scope Qualification Operator

The :: Scope Qualification Operator

The :: scope resolution operator allows you to refer to types that are not in the current scope or immediately enclosing scope:

```
module Types {
    sequence<long> LongSeq;
};

module MyApp {
    sequence<Types::LongSeq> NumberTree;
};
```

You can anchor a lookup explicitly at the global scope with a leading :: operator: ::Types::LongSeq



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-38

Notes:

The :: operator allows you to refer to types that are defined in a different module, that is, are not in the enclosing scope. The operator also makes it possible to have mutually-dependent interfaces in different modules.

```
module Children {
    interface Child; // Forward declaration
};

module Parents {
    interface Mother {
        Children::Child* getChild(); // OK
    };
    interface Father {
        Children::Child* getChild(); // OK
    };
};

module Children { // Reopen module
```

```
interface Child { // Define Child
  Parents::Mother* getMother();
  Parents::Father* getFather();
};
};
```

Re-opening the **Children** module is necessary because, syntactically, you cannot forward-declare an interface that is in a different module (a forward declaration requires a simple identifier, not a qualified name). The example is somewhat contrived though, because mutually-dependent interfaces are closely related and, therefore, should probably not be in different modules.

2-39 Type Identifiers

Type Identifiers

Each Slice type has a unique internal identifier, call the type ID:

- For built-in types, the type ID is the name of the type, e.g. `int` or `string`.
- For user-defined types, the type name is the fully-scoped name:


```
module Times {
    struct Time { /* ... */ };
    interface Clock { /* ... */ };
};
```

The type IDs for this definition are `::Times`, `::Times::Time`, and `::Times::Clock`.
- For proxies, the type ID has a trailing `*`, so the type ID of the proxy for the `Clock` interface is `::Times::Clock*`.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-39

Notes:

Internally, Ice identifies each Slice type by its type ID. Type IDs for built-in types are the name of the type itself. Type IDs for user-defined types are the fully-qualified scoped name of the type, with a trailing `*` if the type is a proxy.

2-40 Operations on Object

Operations on Object

All interfaces and classes implicitly inherit from `Object`:

```
sequence<string> StringSeq;
interface Object { // "Pseudo" Slice!
    void ice_ping();
    bool ice_isA(string typeId);
    string ice_id();
    StringSeq ice_ids();
    // ...
};
```

- `ice_ping` provides a basic reachability test.
- `ice_isA` tests whether an interface is compatible with the supplied type.
- `ice_id` returns the type ID of the interface.
- `ice_ids` returns all types IDs of the interface (the type ID of the interface itself, plus the type IDs of all base interfaces).



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-40

Notes:

All interfaces implicitly inherit from `Object`, which provides a number of built-in operations. (Note that you cannot define `Object` in Slice because `Object` is a keyword. But, if you could define it, part of its definition would look as shown above.)

Note that all operations of `Object` contain an `ice_` prefix. This makes name clashes with user-defined operations impossible (because user-defined operations cannot contain this prefix).

- `ice_ping` provides a basic reachability test. If it completes successfully, the server hosting the target object is reachable, and the target object was found in the server, that is, the proxy denotes a valid object at the time of the call.

An `ObjectNotExistException` indicates that the server is reachable, but that the proxy is dangling.

If the server is down, `ice_ping` raises `ConnectionRefusedException`; other reachability problems are reported with an appropriate exception, such as `ConnectTimeoutException`.

- `ice_isA` tests whether a proxy is compatible with the supplied type ID. It is the distributed equivalent of a Java `instanceof`.

- `ice_id` returns the type ID of an interface. Note that the returned ID denotes the most-derived type of the object, not the static interface type of the proxy.
- `ice_ids` returns a sequence of type IDs of an interface, that is, the actual most-derived type of the object, plus all its ancestor type IDs.

2-41 Local Types

Local Types

The APIs for the Ice run time are (almost) completely defined in Slice. Most of the Slice definitions use the **local** keyword, for example:

```
module Ice {
    local interface Communicator { /* ... */ };
};
```

local types cannot be accessed remotely; they define library objects.

local types do *not* inherit from **Object**. Instead, they derive from a common base **LocalObject**.

Therefore, you cannot pass a local object where a non-local object is expected and vice-versa.

You can define your own **local** interfaces, but there will rarely be a need to do so.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-41

Notes:

Most of the APIs for the Ice run time (apart from a few language-specific helper functions) are defined in Slice. This obviates the need to define a separate language-specific API for each language mapping. Most of the interfaces to the run time are specified as **local** interfaces. **local** interfaces are implemented in libraries, that is, they are simply objects that live in the application's local address space and that the application links against.

local interfaces have a separate inheritance hierarchy with **LocalObject** as the root, so you cannot accidentally pass a local object where a non-local object is expected and vice-versa. Moreover, the Slice compilers do not generate marshaling code for local types, so it is physically impossible to pass them to another address space.

You can define and implement your own local interfaces, but there will rarely be a need to do so. (One exception to this rule is servant locators, which must be implemented as local objects. Refer to Section 18-6.)

2-42 Metadata

Metadata

Any Slice construct can be preceded by a metadata directive, for example:

```
["java: type: java.util.LinkedList<Integer>"] sequence<int> IntSeq;
```

Metadata directives can also appear at global scope:

```
[["java: package: com.acme"]]
```

Global metadata directives must precede any Slice definitions in a source file.

Metadata directives affect the code generator only.

Metadata directives *never* affect the client–server contract: no matter how you add, remove, or change metadata directives, the information that is exchanged on the wire is always the same.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-42

Notes:

A local metadata directive (delimited by single square brackets) can appear as a prefix to any Slice definition. A global metadata directive (delimited by double square brackets) must appear at global scope and precede any other Slice definitions in a source file.

You can have several metadata directives in a single pair of brackets, for example:

```
["amd", "java:getset"] interface Foo { /* ... */};
```

Metadata directives provide supplementary information to the code generator, for example, to change the default language mapping of a type, or to cause the compiler to emit asynchronous dispatch APIs.

Metadata directives *never* change the information that is passed across the wire, that is, changing a metadata directive cannot invalidate the client–server contract.

We will examine the various metadata directives in the relevant chapters to which they apply.

2-43 The slice2java Compiler

The slice2java Compiler

The `slice2java` command compiles one or more Slice definition files.
`slice2java [options] file...`

For example:

```
slice2java MyDefs.ice
```

This generates a number of source files, one for each class, using the usual directory hierarchy for modules (which map to Java packages).

Commonly used options:

- `-DNAME`, `-DNAME=DEF`, `-UNAME`
Define or undefine preprocessor symbol *NAME*.
- `-I DIR`
Add *DIR* to the search path for `#include` directives.
- `--impl`
Create sample implementation files.



Slice Interface Definition Language
Copyright © 2005-2010 ZeroC, Inc.

2-43

Notes:

You use the `slice2java` command to compile Slice definitions. You can compile multiple Slice files in a single invocation of the compiler. For each input file, the compiler creates several output files, one for each generated Java class.

Note: `slice2java` has many more options than are shown here. Consult the Ice manual for details.

One option you may want to explore is `--impl`: it creates a `classnameI.java` file for each class that you need to implement. These files contain an outline of the code that you need to write to implement the Slice interfaces you have compiled. Using this option can save you a lot of tedious typing.

3 Assignment 1

Creating Slice

Definitions

3 Assignment 1: Creating Slice Definitions

3-1 Exercise Overview

Exercise Overview

In this exercise, you will:

- gain hands-on experience of how to create Slice definitions by designing interfaces for a simple application.

By the completion of this exercise, you will have gained experience in creating Slice definitions, the syntax and semantics of the language, and how to use the `slice2java` compiler.

Assignment 1 Creating Slice Definitions
Copyright © 2005-2010 ZeroC, Inc.3-1

Notes:

In this exercise, you will gain hands-on experience of how to create Slice definitions by designing interfaces for a simple application.

3-1-1 Exercise Objectives

By the completion of this exercise, you will have gained experience in creating Slice definitions, the syntax and semantics of the language, and how to use the `slice2java` compiler.

3-2 A Simple Remote File System

Simple Remote File System

Functionality

- The file system consists of directories and files. The usual hierarchical structure applies, so the file system has a single root directory that, recursively, can contain other directories and files.
- Each directory and file has a name; names within the same parent directory must be unique, as for a Windows or UNIX file system.
- Directories provide a way to list their contents.
- The content of files can be read and written. (Only text files are supported, not binary files.)
- For the time being, the file system does not permit life cycle operations, that is, clients can read and write the contents of files and list the contents of directories, but cannot create or delete files or directories.



Assignment 1 Creating Slice Definitions
Copyright © 2005-2010 ZeroC, Inc.

3-2

Notes:

3-2-1 Functionality

The file system consists of directories and files. The usual hierarchical structure applies, so the file system has a single root directory that, recursively, can contain other directories and files.

Each directory and file has a name; names within the same parent directory must be unique, as for a Windows or UNIX file system.

Directories provide a way to list their contents.

The content of files can be read and written. (Only text files are supported, not binary files.)

For the time being, the file system does not permit life cycle operations, that is, clients can read and write the contents of files and list the contents of directories, but cannot create or delete files or directories.

3-3 What You Need to Do

What You Need to Do

Create Slice definitions for this application.

- In your `lab1` directory, locate the file named `Filesystem.ice`.
- Place your definitions into this file. The directory also contains a project file `build.xml` that you can use to compile your definitions.

Consider the following:

- What interfaces need to be present in your definitions, and how they should relate to each other.
- What error conditions can arise and how to best inform clients of any errors.
- What interaction patterns are clients likely to exhibit. Would it be advisable to modify your definitions to accommodate such patterns and, if so, why?

Once you have compiled your definitions, have a look at the generated code. What parts of your specification do you recognize in the generated code?



Assignment 1 Creating Slice Definitions
Copyright © 2005-2010 ZeroC, Inc.

3-3

Notes:

3-3-1 Create Slice definitions for this application.

In your `lab1` directory, you will find a file named `Filesystem.ice`.

Place your definitions into this file. The directory also contains a project file `build.xml` that you can use to compile your definitions. (The project file uses a custom build step that invokes the `slice2java` command.)

Consider the following:

- What interfaces need to be present in your definitions, and how they should relate to each other.
- What error conditions can arise and how to best inform clients of any errors.
- What interaction patterns are clients likely to exhibit? Would it be advisable to modify your definitions to accommodate such patterns and, if so, why?

When you have compiled your definitions:

- Have a look at the generated code.
- What parts of your specification do you recognize in the generated code?

3-4 Slice Definitions for a Simple Remote File System

One Possible Solution

```

module Filesystem {
  exception IOError {
    string reason;
  };
  interface Node {
    idempotent string name();
  };
  sequence<string> Lines;
  interface File extends Node {
    idempotent Lines read() throws IOError;
    idempotent void write(Lines text) throws IOError;
  };
  sequence<Node*> NodeSeq;
  interface Directory extends Node {
    idempotent NodeSeq list();
  };
};

```



Assignment 1 Creating Slice Definitions
Copyright © 2005-2010 ZeroC, Inc.

3-4

Notes:

3-4-1 One Possible Solution

The example below shows one possible solution to this exercise. Keep in mind that this is not the only possible or correct solution—it merely provides one particular way to solve the problem.

```

module Filesystem {
  exception IOError {
    string reason;
  };

  interface Node {
    idempotent string name();
  };

  sequence<string> Lines;

```

```
interface File extends Node {
    idempotent Lines read() throws IOError;
    idempotent void write(Lines text) throws IOError;
};

sequence<Node*> NodeSeq;

interface Directory extends Node {
    idempotent NodeSeq list();
};
};
```

3-4-2 Another Way to Structure Slice Definitions

Note that clients are likely to call the `list` operation on a directory for display purposes. This means that, if a directory contains 20 files, it is likely that the client will immediately follow up by calling the `name` operation on each file so it can display its name. A more efficient way to structure the Slice definitions is therefore as follows:

```
module Filesystem {
    exception IOError {
        string reason;
    };

    interface Node {
        idempotent string name();
    };

    sequence<string> Lines;

    interface File extends Node {
        idempotent Lines read() throws IOError;
        idempotent void write(Lines text) throws IOError;
    };

    enum NodeType { FileT, DirT };

    struct NodeDetails {
        string name;
        NodeType type;
        Node* proxy;
    };

    sequence<NodeDetails> NodeDetailsSeq;

    interface Directory extends Node {
        idempotent NodeDetailsSeq list();
    };
};
```

4 Client-Side Slice-to-Java Mapping

4 Client-Side Slice-to-Java Mapping

4-1 Lesson Overview

Lesson Overview

- This lesson presents:
 - the mapping from Slice to Java for the client side.
 - the relevant APIs that are necessary to initialize and finalize the Ice run time
 - instructions for compiling a Java Ice client.
- By the end of this lesson, you will know how each Slice type maps to Java and be able to write a working Ice client.

Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.4-1

Notes:

This lesson presents the mapping from Slice to Java for the client side. It includes the relevant APIs that are necessary to initialize and finalize the Ice run time, and explains how to compile a Java Ice client.

We present the server-side mapping in Chapter 6.

4-1-1 Lesson Objectives

By the end of this lesson, you will know how each Slice type maps to Java and be able to write a working Ice client.

4-2 Client-Side Java Mapping

Client-Side Java Mapping

The client-side Java mapping defines rules for:

- initializing and finalizing the Ice run time
- mapping each Slice type into Java
- invoking operations and passing parameters
- handling exceptions

The mapping is fully thread-safe: you need not protect any Ice-internal data structures against concurrent access.

The mapping rules are simple and regular: know them! The generated files are no fun to read at all!

`slice2java`-generated code is platform independent.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-2

Notes:

The bulk of the Ice Java mapping defines rules for how each Slice construct is represented in Java. It also defines rules for invoking operations, passing parameters, and handling exceptions. In addition, the Java mapping provides a small number of helper functions that you need to initialize and finalize the Ice run time.

The mapping is fully thread-safe: for example, concurrent access by different threads to a proxy (that is, having two threads invoke on the same proxy concurrently) is safe. The only critical regions you need to worry about are those around your own data.

The mapping rules are simple and regular, and you should know them: it is far easier to look at the Slice definitions to deduce what the corresponding Java API looks like, than it is to read the generated source files. The source files are full of sometimes cryptic code and are cluttered with mapping-internal definitions that are not part of the public API. This is not to say that you cannot use the generated source files to confirm some detail, but they are definitely unsuitable to get an idea of what the generated API looks like.

Note that the code that is generated by `slice2java` is platform independent: for example, you can compile Slice definitions under Windows and then compile the generated code under Linux without problems.

4-3 Initializing the Ice Run Time

Initializing the Ice Run Time

```

public static void main(String[] args)
{
    int status = 1;
    Ice.Communicator ic = null;
    try {
        ic = Ice.Util.Initialize(args);
        // client code here...
        status = 0;
    } catch (Exception e) {
    }
    finally {
        if (ic != null) {
            try {
                ic.destroy();
            } catch (Exception e) {
            }
        }
    }
    System.exit(status);
}

```



Client-Side Slice-to-Java Mapping
 Copyright © 2005-2010 ZeroC, Inc.

4-3

Notes:

Before you can do anything with the Ice run time, you must initialize it. The above code shows the general pattern for how to do this. (There is quite a bit unexplained here. For now, please accept the above code at face value. We explore the initialization and purpose of the Ice communicator in more detail in Chapter 18.)

The main handle to the Ice run time is of type **Ice.Communicator**. You must first obtain an instance of this type before you can do anything else. You obtain the instance by calling **Ice.Util.initialize** (a static function) as shown above.

If the initialization fails, **Ice.Util.initialize** throws an exception and, as a result, the value of **ic** is unchanged. (**ic** still has the initial null value.)

If initialization succeeds, **ic** has a non-null value.

If the code has successfully initialized the Ice run time, it *must* destroy the communicator before returning from **main**. Failure to do so causes undefined behavior. Destroying the communicator reclaims resources used by the Ice run time. In particular, **destroy** terminates and joins with all threads used by the run time.

All Ice programs follow this basic pattern:

1. Initialize the communicator.
2. If initialization succeeded, destroy the communicator before leaving **main**.

4-4 Mapping for Identifiers

Mapping for Identifiers

Slice identifiers map to corresponding Java identifiers:

```
struct Employee {
    int number;
    string name;
};
```

The generated Java contains:

```
public class Employee
    implements java.lang.Cloneable, java.lang.Serializable
{
    public int number;
    public String name;
    // ...
}
```

Slice identifiers that clash with Java keywords are escaped with a `_` prefix, so Slice `while` maps to Java `_while`.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-4

Notes:

Slice identifiers map to the same Java identifiers, except when a Slice identifier happens to be a Java keyword. To avoid the clash, `slice2java` prefixes such identifiers with `_`. Some Slice types create more than one Java type. For example, a Slice proxy `Foo` creates Java `Foo` and `FooPrx` types (plus some others). In such cases, if the Slice identifier is a Java keyword, the `_` prefix is applied only where necessary. For example, the proxy name `while` creates the Java symbols `_while` and `whilePrx` (not `_whilePrx`).

4-5 Mapping for Modules

Mapping for Modules

Slice modules map to Java packages. The nesting of definitions is preserved:

```

module M1 {
  module M2 {
    // ...
  };
  // ...
};

```

This maps to Java as:

```

package M1;
// Definitions for M1 here...

package M1.M2;
// Definitions for M1.M2 here...

```



Client-Side Slice-to-Java Mapping
 Copyright © 2005-2010 ZeroC, Inc.

4-5

Notes:

Slice modules map to Java packages, using the same structure of nesting as the Slice definitions.

If a Slice module is re-opened, the corresponding Java package is re-opened as well:

```

module M1 {
  // ...
};
module M3 {
  // ...
};
module M1 { // Re-open M1
  // ...
};

```

This maps to Java as:

```

package M1;
// Definitions for M1 here...

```

```
package M1.M2;  
// Definitions for M1.M2 here...  
  
package M1;  
// More definitions for M1 here...
```

4-6 Mapping for Built-In Types

Mapping for Built-In Types

The built-in Slice types map to Java types as follows:

Slice Type	Java Type
<code>bool</code>	<code>boolean</code>
<code>byte</code>	<code>byte</code>
<code>short</code>	<code>short</code>
<code>int</code>	<code>int</code>
<code>long</code>	<code>long</code>
<code>float</code>	<code>float</code>
<code>double</code>	<code>double</code>
<code>string</code>	<code>String</code>



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-6

Notes:

Not surprisingly, the Slice built-in types map to their Java counterparts. Slice `string` maps to `java.lang.String`.

4-7 Mapping for Enumerations

Mapping for Enumerations

Slice enumerations map unchanged to the corresponding Java enumeration.

```
enum Fruit { Apple, Pear, Orange };
```

This maps to the Java definition:

```
public enum Fruit implements java.io.Serializable {  
    Apple,  
    Pear,  
    Orange;  
    // ...  
}
```



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-7

Notes:

Not surprisingly, Slice enumerations map to the corresponding Java enumeration without change.

4-8 Mapping for Structures

Mapping for Structures

Slice structures map to Java classes with all data members public:

```
struct Employee {  
    string lastName;  
    string firstName;  
};
```

This maps to:

```
public class Employee  
    implements java.lang.Cloneable, java.io.Serializable {  
    public String lastName;  
    public String firstName;  
  
    public Employee();  
    public Employee(String lastName, String firstName);  
    public boolean equals(java.lang.Object rhs);  
    public int hashCode();  
    public java.lang.Object clone();  
};
```



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-8

Notes:

Structures map to classes with a public member for each Slice member, in the same order. The generated class provides a default constructor that explicitly initializes those data members that declare default values, as well as a “one-shot” constructor that takes an initial value for each data member. The class also defines **equals**, **hashCode**, and **clone** member functions, which have the usual behavior.

4-9 Mapping for Sequences

Mapping for Sequences

By default, Slice sequences map to Java arrays.

```
sequence<Fruit> platter;
```

No code is generated for the sequence. Use it as you would any other array, for example:

```
Fruit[] platter = { Fruit.Apple, Fruit.Pear };
```

```
assert(platter.length == 2);
```



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-9

Notes:

By default, Slice sequences map to Java arrays, so all the operations applicable to Java arrays apply to Slice sequences.

4-10 Custom Mapping for Sequences

Custom Mapping for Sequences

You can change the mapping for a sequence to a custom type:

```
["java: type: java.util.LinkedList<Fruit>"]
sequence<Fruit> FruitPlatter;
```

The nominated type must implement the `java.util.List<T>` interface. You can override members, parameter, or return values, for example:

```
sequence<Fruit> Breakfast;
["java: type: java.util.LinkedList<Fruit>"]
sequence<Fruit> Dessert;
```

```
struct Meal1 {
    Breakfast b;
    Dessert d;
};
```

```
struct Meal2 {
    ["java: type: java.util.LinkedList<Fruit>"] Breakfast b;
    ["java: type: java.util.Vector<Fruit>"] Dessert d;
};
```



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-10

Notes:

The `java:type` metadata directive overrides the default sequence mapping to use a custom type instead. You can specify any type that implements the `java.util.List<T>` interface, including types you have created yourself.

The mapping is overridable for individual members, parameters, and return values. For example, with the above definitions, `Meal1.b` is an array, whereas `Meal2.b` is a linked list.

4-11 Mapping for Dictionaries

Mapping for Dictionaries

Slice dictionaries map to Java maps:

```
dictionary<long, Employee> EmployeeMap;
```

No code is generated for this dictionary. Rather, `slice2java` substitutes `java.util.Map<Long, Employee>` for `EmployeeMap`.

It follows that you can use the dictionary like any other Java map, for example:

```
java.util.Map<Long, Employee> em =
    new java.util.HashMap<Long, Employee>();

Employee e = new Employee();

e.number = 31;
e.firstName = "James";
e.lastName = "Gosling";

em.put(e.number, e);
```



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-11

Notes:

Slice dictionaries map to types that support the `java.util.Map<K,V>` interface (`java.util.HashMap<K,V>`, by default), so there are no surprises here—all the usual operations for Java maps apply.

4-12 Custom Mapping for Dictionaries

Custom Mapping for Dictionaries

You can change the default mapping for dictionaries via metadata:

```
["J ava: type: J ava. uti l . L i n k e d H a s h M a p" ]  
d i c t i o n a r y < s t r i n g , s t r i n g > S t r i n g T a b l e ;
```

The type specified for the dictionary must support the `java.util.Map` interface.

As for sequences, you can override the type for individual members, parameters, and return values.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-12

Notes:

As for sequences, you can use the `java:type` metadata directive to override the default mapping. The nominated type must implement the `java.util.Map<K,V>` interface.

4-13 Mapping for Constants

Mapping for Constants

Slice constants map to a Java interface with a `value` member that stores the value.

```
const string Advice = "Don't Panic!";
```

```
enum Fruit { Apple, Pear, Orange };  
const Fruit FavoriteFruit = Pear;
```

This maps to:

```
public interface Advice {  
    String value = "Don't Panic!";  
}
```

```
public interface FavoriteFruit {  
    Fruit value = Fruit.Pear;  
}
```



Notes:

Slice constants map to Java interfaces with a `value` member that stores the value.

4-14 Mapping for User Exceptions

Mapping for User Exceptions

User exceptions map to Java classes derived from `UserException`.

```
exception GenericError {
    string reason;
};
```

This maps to:

```
public class GenericError extends Ice.UserException {
    public String reason;
    public GenericError();
    public GenericError(String reason);
    public String ice_name() {
        return "GenericError";
    }
}
```

Slice exception inheritance is preserved in Java, so if Slice exceptions are derived from `GenericError`, the corresponding Java exceptions are derived from `GenericError`.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-14

Notes:

Slice exceptions map to Java classes with a public data member for each exception member.

The generated class provides a default constructor that explicitly initializes those data members that declare default values, as well as a “one-shot” constructor that takes an initial value for each data member. The class also contains an `ice_name` method that returns the name of the exception.

Note that all user exceptions derive from `Ice.UserException`, so you can install a catch handler that catches all user exceptions. In turn, `Ice.UserException` derives from `java.lang.Exception`.

```
package Ice;
```

```
public abstract class UserException extends Exception {
    public java.lang.Object clone();
    public abstract String ice_name();
    public String toString();
}
```

The exception inheritance of Slice is preserved for the corresponding Java exceptions, so if you define derived exceptions, the generated Java code uses the same derivation as the Slice definition.

4-15 Mapping for Run-Time Exceptions

Mapping for Run-Time Exceptions

Ice run-time exceptions are derived from `Ice.LocalException`. In turn, `Ice.LocalException` derives from `java.lang.RuntimeException`. As for user exceptions, `Ice.LocalException` provides an `ice_name` method that returns the name of the exception.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-15

Notes:

Run-time exceptions are derived from `Ice.LocalException`, which in turn derives from `java.lang.RuntimeException`.

4-16 Mapping for Interfaces

Mapping for Interfaces

A Slice interface maps to a number of classes.

```
interface Simple {
    void op();
};
```

This generates the following interfaces and classes:

```
interface Simple
final class SimpleHolder

interface SimplePrx
final class SimplePrxHolder
final class SimplePrxHelper

interface _SimpleOperations
interface _SimpleOperationsNC
```



Client-Side Slice-to-Java Mapping
 Copyright © 2005-2010 ZeroC, Inc.

4-16

Notes:

For the above **Simple** interface, the compiler generates a number of interfaces and classes for use by the client:

- **Simple**
For each Slice interface, the compiler generates a corresponding Java interface with the same name. All interfaces ultimately derive from the **Ice.Object** interface (and therefore provide the methods defined for **Ice.Object**).
- **SimpleHolder**
This class defines a holder type for the interface, which is used to implement out-parameters.
- **SimplePrx**
The proxy interface provides access to the operations of a remote object.
- **SimplePrxHolder**
This class defines a holder type for the proxy, which is used to implement out-parameters.

- **`SimplePrxHelper`**
The proxy helper class provides methods that allow you to safely down-cast a proxy to a more derived type.
- **`_SimpleOperations`**
`_SimpleOperationsNC`
These interfaces contain definitions of the Slice operations.

4-17 The Proxy Interface

The Proxy Interface

An instance of a proxy interface acts as the local ambassador for a remote object. Invoking a method on the proxy results in an RPC call to the corresponding object in the server.

```
interface Simple {
    void op();
};
```

This generates:

```
public interface SimplePrx extends Ice.ObjectPrx {
    public void op();
    public void op(java.util.Map<String, String> __ctx);
}
```

The version without the `__ctx` parameter simply calls the version with the `__ctx` parameter, supplying a default context.

`SimplePrx` derives from `Ice.ObjectPrx`, so all proxies support the operations on `Ice.Object`.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-17

Notes:

The compiler generates a proxy interface for each Slice interface. In general, the name of the proxy interface is `<interface-name>Prx`. If an interface is nested in a module `M`, the generated class is part of package `M`, so the fully-qualified name is `M.<interface-name>Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a proxy instance. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Object`.

For each operation in the interface, the proxy class has a member function of the same name.¹ For the preceding example, we find that the operation `op` has been mapped to the member function `op`. Also note that `op` is overloaded: the second version of `op` has a parameter `__ctx` of type `java.util.Map<String,String>`. This parameter is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it

Because all the `<interface-name>Prx` types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. The proxy references handed out by the Ice run time are always of type `<interface-name>Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

¹ Note that the operations are actually inherited from the `_SimpleOperations` and `_SimpleOperationsNC` base interfaces; for simplicity we show them as if they were part of the proxy interface.

4-18 Methods on Ice.ObjectPrx

Methods on Ice.ObjectPrx

`Ice.ObjectPrx` is defined as follows:

```
package Ice;

public interface ObjectPrx {
    boolean equals(Java.Lang.Object r);
    int hashCode();
    Identity ice_getIdentity();
    boolean ice_isA(String __id);
    String ice_id();
    String[] ice_ids();
    void ice_ping();
    // ...
}
```

Every Ice object supports these operations.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-18

Notes:

`Ice.ObjectPrx` provides a number of operations that are supported by all Ice objects:

- **equals**
This operation compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does not imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, **equals** returns **false** even though the proxies denote the same object.
- **hashCode**
This method returns a hash value for the proxy.
- **ice_getIdentity**
This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

```
module Ice {
    struct Identity {
        string name;
        string category;
    };
}
```

```
};
```

To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

```
Ice.ObjectPrx o1 = ...;
Ice.ObjectPrx o2 = ...;

Ice.Identity i1 = o1.ice_getIdentity();
Ice.Identity i2 = o2.ice_getIdentity();
if (i1.equals(i2))
    // o1 and o2 denote the same object
else
    // o1 and o2 denote different objects
```

- **ice_isA**

This method determines whether the object denoted by the proxy supports a specific interface. The argument to **ice_isA** is a type ID. (Refer to Section 2-39.)

For example, to see whether a proxy of type **ObjectPrx** denotes a **Printer** object, we can write:

```
Ice.ObjectPrx o = ...;
if (o != null && o.ice_isA("::Printer"))
    // o denotes a Printer object
else
    // o denotes some other type of object
```

Note that we are testing whether the proxy is null before attempting to invoke the **ice_isA** method. This avoids getting a **NullPointerException** if the proxy is null.

- **ice_id**

This method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type **BasePrx**, with a static type ID of **::Base**, the return value of **ice_id** might be **::Base**, or it might something more derived, such as **::Derived**.

- **ice_ids**

This method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports, including **::Ice::Object**.

- **ice_ping**

This method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as **ObjectNotExistException** or **ConnectTimeoutException**.

The `ice_isA`, `ice_id`, `ice_ids`, and `ice_ping` methods make invocations on the remote Ice object (asynchronous versions of these methods are also available). The remaining proxy methods, along with many others not listed above, operate on the local proxy object. See the Ice manual for more information on the methods supported by proxy objects.

4-19 Proxy Helpers

Proxy Helpers

For each interface, the compiler generates a helper class that allows you to do type-safe down-casts:

```
public final class SimplePrxHelper extends Ice.ObjectPrxHelper {
    public static SimplePrx checkedCast(Ice.ObjectPrx b);
    public static SimplePrx checkedCast(
        Ice.ObjectPrx b,
        java.util.Map<String, String> ctx);
    public static SimplePrx uncheckedCast(Ice.ObjectPrx b);
    // ...
}
```

Both casts test an is-a relationship.

- A **checkedCast** checks with the server whether the object actually supports the specified type and so requires sending a message.
- An **uncheckedCast** is a sledgehammer cast (so you had better get it right!) but does not require sending a message.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-19

Notes:

Given a proxy of any type, you can use a **checkedCast** to test whether the proxy supports a specific interface, serving a purpose similar to Java's **instanceof** operator. For example:

```
Ice.ObjectPrx obj = ...; // Get a proxy from somewhere...
```

```
SimplePrx simple = SimplePrxHelper.checkedCast(obj);
if (simple != null)
    // Object supports the Simple interface...
else
    // Object is not of type Simple...
```

A **checkedCast** returns a non-null proxy if the passed proxy supports the specified interface and a null proxy otherwise. Note that a **checkedCast** contacts the server, so the server must be running and the proxy must denote an existing object for the **checkedCast** to succeed. If the target object cannot be contacted, **checkedCast** throws an exception.

An **uncheckedCast**, on the other hand, is a local operation and does not contact the server, so it is not verified in any way. Use an **uncheckedCast** only if you are certain that a proxy denotes an object that supports the interface you are casting to. Incorrect use of an **uncheckedCast** causes undefined behavior, often resulting in a run-time exception. You can also end up with an operation invocation that succeeds but does something completely unexpected (such as being invoked on the wrong object).

Note: Normally, you use the single-argument version of **checkedCast**. You will rarely (if ever) need to use the version with the **ctx** parameter. Please consult the Ice manual for details.

4-20 Mapping for Operations

Mapping for Operations

Slice operations map to methods on the proxy interface.

```
interface Simple {  
    void op();  
};
```

Invoking a method on the proxy instance invokes the operation in the remote object:

```
SimplePrx p = ...;  
p.op(); // Invoke remote op() operation
```

The mapping is the same, regardless of whether an operation is a normal operation or has an **idempotent** qualifier.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-20

Notes:

As we saw in Section 4-174.19, each Slice operation maps to a pair of methods on the proxy interface. To invoke an operation, you call the method on the proxy instance like any other Java method. The generated code and the Ice run time take care of getting the invocation to the correct object in the server.

Normal and **idempotent** operations have the same signature.

4-21 Mapping for Return Values and In-Parameters

Mapping for Return Values and In-Parameters

Return values and In-parameters are passed either by value (for simple types), or by reference (for complex types).

```
interface Example {  
    string op(double d, string s);  
};
```

The proxy operation is:

```
String op(double d, String s);
```

You invoke the operation like any other Java method:

```
ExamplePrx p = ...;
```

```
String result = p.op(3.14, "Hello");
```

```
System.out.println(result);
```

- To pass a null proxy, pass a null reference.
- You can pass a null parameter for strings, sequences, and dictionaries to pass the empty string, sequence, or dictionary.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-21

Notes:

Return values and in-parameters are passed as you would expect: by value for simple types, and by reference for complex types. (Of course, you must initialize in-parameters for a sensible value to be sent to the server.)

The Java mapping uses a Java null reference as the null proxy, so you can pass a null proxy by passing a Java null reference.

It is legal to pass a null reference where a string, sequence, or dictionary is expected. In that case, the receiver gets an empty string, an empty sequence, or an empty dictionary, respectively. This feature is provided for convenience: especially for sequences of deeply nested data structures, it is useful not to have to recursively initialize every member before passing the sequence to a Slice operation.

4-22 Mapping for Out-Parameters

Mapping for Out-Parameters

Out-parameters are passed via a **Holder** type:

- Built-in types are passed as `Ice.ByteHolder`, `Ice.IntHolder`, `Ice.StringHolder`, etc. User-defined types are passed as `<name>Holder`.

All holder classes have a public `value` member, for example:

```
package Ice;
```

```
public final class StringHolder {
    public StringHolder() {}
    public StringHolder(String value) {
        this.value = value;
    }
    public String value;
}
```

You pass a holder instance where an out-parameter is expected; when the operation completes, the `value` member contains the returned value.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-22

Notes:

Because Java does not allow passing of parameters by reference, out-parameters are passed as a reference to a **Holder** class. The **Holder** class contains a `value` member that is assigned to by the operation. For the Slice built-in types, the Holder classes are `Ice.ByteHolder`, `Ice.IntHolder`, `Ice.StringHolder`, and so on. For user-defined types, the holder class is generated as `<name>Holder`.

Each holder class has a default constructor that default-initializes the `value` member, and a single-argument constructor that initializes the `value` member with the supplied value.

Note that holder classes are generated for *every* Slice type. For example, by default, the compiler does not generate a Java type for a Slice sequence because sequences map to arrays. However, the compiler *does* generate a holder type for each sequence; the holder class' `value` member is simple an array of the sequence's element type.

You can pass out-parameters as you would any other Java holder type, for example:

```
interface Example {
    void op(out string s);
};
```

You could call the operation as follows:

```
ExamplePrx p = ...;  
Ice.StringHolder s = new Ice.StringHolder();  
p.op(s);  
System.out.println(s.value);
```

4-23 Exception Handling

Exception Handling

Operation invocations can throw exceptions:

```
exception Tantrum { string reason; };
interface Child {
    void askToCleanUp() throws Tantrum;
};
```

You can call askToCleanUp like this:

```
ChildPrx child = ...; // Get proxy...
try {
    child.askToCleanUp(); // Give it a try...
} catch (Tantrum t) {
    System.out.println("The child says: " + t.reason);
}
```

Exception inheritance allows you to handle errors at different levels with handlers for base exceptions at higher levels of the call hierarchy.

The value of out-parameters if an exception is thrown is undefined.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-23

Notes:

If an operation throws an exception, you catch the exception as you would any other Java exception.

Of course, you can install exception handlers at different levels of the call hierarchy. For example, you can catch a specific exception at the point of call, and deal with other exceptions generically:

```
public class Client {
    static void run() {
        ChildPrx child = ...; // Get child proxy...
        try {
            child.askToCleanUp();
        } catch (Tantrum t) {
            System.out.print("The child says: ");
            System.out.println(t.reason);
            child.scold(); // Recover from error...
        }
        child.praise(); // Give positive feedback...
    }
}
```

```
public static void
main(String[] args)
{
    try {
        // ...
        run();
        // ...
    } catch (Ice.LocalException e) {
        e.printStackTrace();
    } catch (Ice.UserException e) {
        System.err.println(e.getMessage());
    }
}
```

Note that catching `Ice.LocalException` catches all run-time exceptions, and catching `Ice.UserException` catches all user exceptions.

If an operation has out-parameters and throws an exception, the values of the out-parameters are undefined: they may still have the previous value or may have been changed (and not necessarily to a value that was returned by the operation).

4-24 Mapping for Classes

Mapping for Classes

Slice classes map to Java classes:

- For each Slice member (if any), the class contains a corresponding public data member.
- If the class has operations, it is abstract and derives from the `_<name>Operations` and `_<name>OperationsNC` interfaces. These interfaces contain method definitions corresponding to the Slice operations.
- The class has a default constructor and a “one-shot” constructor with one parameter for each class member.
- Slice classes without a base class derive from `Ice.Object`.
- Slice classes with a base class derive from the corresponding base class.
- All classes support the operations on `Ice.Object`.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-24

Notes:

Slice classes map to Java classes. For example:

```
class TimeOfDay {
    short hour;           // 0 - 23
    short minute;        // 0 - 59
    short second;        // 0 - 59
    string format();     // Return time as hh:mm:ss
};
```

This maps to the following Java definitions:

```
public interface _TimeOfDayOperations {
    String format(Ice.Current current);
}

public interface _TimeOfDayOperationsNC {
    String format();
}
```

```

public abstract class TimeOfDay extends Ice.ObjectImpl
    implements _TimeOfDayOperations,
        _TimeOfDayOperationsNC
{
    public short hour;
    public short minute;
    public short second;

    public TimeOfDay() {}

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    public String format()
    {
        return format(null);
    }
    // ...
}

```

Note that the generated class is abstract. (For Slice classes without operations, the generated Java class is concrete.)

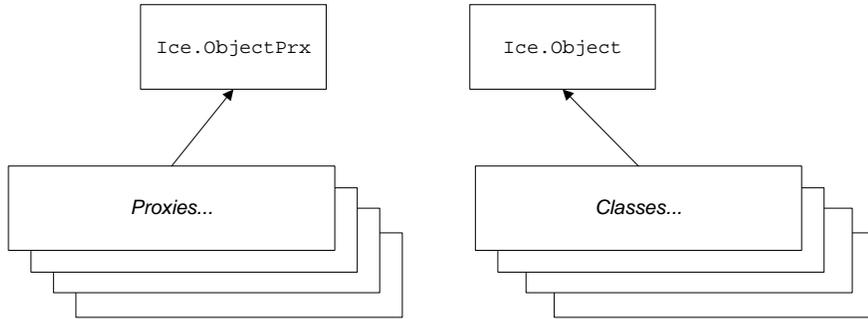
The `_<name>Operations` base interface contains one method for each Slice operation of the class. The operation signature is as for operations on proxies, with a trailing parameter of type `Ice.Current`.²

The `_<name>OperationsNC` base interface (**NC** stands for “no Current”) also contains a method for each Slice operation, but without the trailing `Ice.Current` parameter. As you can see from the example, the “no Current” versions of the operations are implemented by the compiler. They simply invoke the version with a `Current` parameter, supplying a null reference.

The upshot of this is that, to provide an implementation of the abstract class, you must implement the methods the class inherits from the `_<name>Operations` interface, but not the ones it inherits from the `_<name>OperationsNC` interface.

² We examine the purpose of this parameter in more detail in Chapter 12. For now, you can ignore it.

Also note that the generated class derives from `Ice.Object` (and *not* from `Ice.ObjectPrx`). This means that you cannot pass a class where a proxy is expected (or vice-versa) because classes and proxies have separate derivation hierarchies:



The default constructor explicitly initializes only those data members that declare default values, while the “one-shot” constructor accepts an initial value for each data member. If a class is derived from some other class, the one-shot constructor accepts an initial value for each data member of the base class(es) and the derived class, in base-to-derived order.

4-25 Inheritance from Ice.Object

Inheritance from Ice.Object

Classes support the methods on Ice.Object:

```
package Ice;
```

```
public interface Object
{
    void ice_ping(Current current);
    boolean ice_isA(String s, Current current);
    String[] ice_ids(Current current);
    String ice_id(Current current);

    int ice_hash();
    void ice_preMarshal ();
    void ice_postUnmarshal ();
}
```



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-25

Notes:

All classes and interfaces inherit a number of methods from **Ice.Object**. The first four methods (**ice_ping**, **ice_isA**, **ice_id**, and **ice_ids**) provide server-side functionality and are not discussed here. The remaining methods are described below:

- **int hashCode()**
This method returns a hash value so you can place classes into hash tables.
- **void ice_preMarshal()**
The Ice run time invokes this function prior to marshaling an instance. By overriding this function, you can, for example, validate the state of an object.
- **void ice_postUnmarshal()**
The Ice run time invokes this function after unmarshaling an instance. By overriding this function, you can, for example, perform additional initialization of the members of the implementation of an abstract class.

4-26 Abstract Classes

Abstract Classes

Classes that inherit methods from the `_<name>Operations` interface are abstract, so they cannot be instantiated.

To allow abstract classes to be instantiated, you must create a class that derives from the compiler-generated class. The derived class must provide implementations of the operations:

```
public class TimeOfDayI extends TimeOfDay {
    public String format(Ice.Current current) {
        DecimalFormat df
            = (DecimalFormat)DecimalFormat.getInstance();
        df.setMinimumIntegerDigits(2);
        return new String(df.format(hour) + ":" +
            df.format(minute) + ":" +
            df.format(second));
    }
}
```

By convention, implementations of abstract classes have the name `<class-name>I`.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-26

Notes:

The Slice compiler cannot generate an implementation of a class with operations for you because it does not know how to implement these operations. Instead, what is generated is an abstract base class from which you must derive a concrete implementation class. The concrete implementation class must provide an implementation of the methods of the `_<name>Operations` interface(s). (Note that, for Slice classes without operations, the compiler generates concrete classes, so you don't have to implement anything in that case.)

Note: The `Ice.Current` argument passed to every operation is a server-side construct that we'll explore later.

4-27 Class Factories

Class Factories

The Ice run time does not know how to instantiate an abstract class unless you tell it how to do that:

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
    // ...
};
```

You must implement the **ObjectFactory** interface and register a factory for each abstract class with the Ice run time.

- The run time calls **create** when it needs to create a class instance.
- The run time calls **destroy** when you destroy the communicator.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-27

Notes:

The Ice run time cannot know what Java class you have created to implement an abstract Slice class and so cannot instantiate the implementation of an abstract class directly. Instead, the Ice run time relies on the help of a factory instance that you implement: it is the factory's job to instantiate a class on behalf of the Ice run time.

Here is an implementation of the factory:

```
class ObjectFactory implements Ice.ObjectFactory {
    public Ice.Object create(String type) {
        assert(type.equals("::M::TimeOfDay")); // use ice_staticId here
        return new TimeOfDayI();
    }
    public void destroy() {
        // Nothing to do
    }
}
```

Note that the factory must implement the methods in **Ice.ObjectFactory**, namely **create** and **destroy**.

The object factory's **create** method is called by the Ice run time when it needs to instantiate a **TimeOfDay** class. The argument to **create** is the type ID of the class to be created.

The factory's **destroy** method is called by the Ice run time when the factory's **Communicator** is destroyed.

4-28 Factory Registration

Factory Registration

Once you have created a factory class, you must register it with the Ice run time for a particular type ID:

```
module Ice {
    local interface Communicator {
        void addObjectFactory(ObjectFactory factory,
                               string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

When the run time needs to unmarshal an abstract class, it calls the factory's **create** method to create the instance.

It is legal to register a factory for a non-abstract Slice class. If you do this, your factory overrides the one that is generated by the Slice compiler.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-28

Notes:

Once you have created a class factory, you must inform the Ice run time for which types it should call your factory, by calling **addObjectFactory** on the communicator. For example:

```
Ice.Communicator ic = ...;
Ice.ObjectFactory of = new TimeOfDayFactory();
ic.addObjectFactory(of, "::M::TimeOfDay");
```

If you try to register more than one factory for the same type ID, **addObjectFactory** throws **AlreadyRegisteredException**. It is legal to register the same factory for multiple type IDs. In this case, when the communicator is destroyed, **destroy** is called on your factory once for each type ID the factory is registered for.

findObjectFactory allows you to retrieve the factory for a type ID. The operation returns null if no factory is registered for the given type ID.

You can also register a factory for a non-abstract Slice class. If you do this, your factory will be used in preference to the Slice-generated factory.

Note that, instead of using a hard-wired type ID string, you should obtain the type ID of a class by calling its static **ice_staticId** function to avoid errors due to typos:

```
ic.addObjectFactory(of, M.TimeOfDay.ice_staticId());
```

4-29 Default Factory

Default Factory

You can register a factory for the empty type ID as a default factory. The Ice run time locates factories in the following order:

1. Look for a factory registered for the specific type ID. If one exists, call **create** on that factory. If the return value is non-null, return the result, otherwise try step 2.
2. Look for the default factory. If it exists, call **create** on the default factory. If the return value is non-null, return the result, otherwise try step 3.
3. Look for a Slice-generated factory (for non-abstract classes). If it exists, instantiate the class.
4. Throw **NoObjectFactoryException**.

If you have both a type-specific factory and a default factory, you can return null from the type-specific factory to redirect class creation to the default factory.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-29

Notes:

You can install a “catch-all” default factory by registering a factory for the empty type ID. The Ice run time first looks for a type-specific factory and falls back on the default factory if no type-specific factory is registered (or its **create** method returns null).

4-30 Stringified Proxies

Stringified Proxies

The simplest stringified proxy specifies:

- host name (or IP address)
- port number
- an object identity

For example:

```
fred: tcp -h myhost.dom.com -p 10000
```

General syntax:

```
<identity>: <endpoint>[: <endpoint>...]
```

For TCP/IP, the endpoint is specified as:

```
tcp -h <host name or IP address> -p <port number>
```

To convert a stringified proxy into a live proxy, use:

```
Communicator.stringToProxy.
```

A null proxy is represented by the empty string.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-30

Notes:

In order for a client to do anything with a server, it must hold a proxy to at least one object in the server. Once the client has its initial proxy (or proxies—usually a very small number of “bootstrap” proxies is sufficient), the client can obtain proxies to further objects by invoking operations on the initial proxies.

The simplest way to provide an initial proxy to a client is to specify it as a string containing the identity of the target object, the `tcp` protocol identifier, a host name (or IP address), and a port number.

You convert a string into a live proxy by calling `stringToProxy` on the communicator, for example:

```
Ice.Communicator ic = Ice.Util.initialize(args);

Ice.ObjectPrx o = ice.stringToProxy("fred:tcp -h myhost.com -p 10000");

// Assume that object has interface M::Example
M.ExamplePrx e = M.ExamplePrxHelper.checkedCast(o);

// Invoke operation foo on object
e.foo();
```

Note that the proxy returned from `stringToProxy` is of type `ObjectPrx`, so the code must downcast to `ExamplePrx` before it can invoke an operation of the `Example` interface.

Note: The syntax for stringified proxies supports several options as well as the specification of indirect proxies. Refer to Chapter 16 and the Ice manual for details.

4-31 Compiling and Running a Client

Compiling and Running a Client

To compile a client, you must:

- compile the Slice-generated source files(s)
- compile your application code

For Linux:

```
$ mkdir classes
$ javac -d classes -classpath \
> classes:$ICEJ_HOME/lib/Ice.jar \
> Client.java generated/Demo/*.java
```

To compile and run the client, `Ice.jar` must be in your `CLASSPATH`.



Client-Side Slice-to-Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

4-31

Notes:

To compile a client, you must compile the generated source code and your application code.

Note that you must have `Ice.jar` in your `CLASSPATH` to compile and run the client.

5 Assignment 2

Creating an Ice Client

5 Assignment 2: Creating an Ice Client

5-1 Exercise Overview

Exercise Overview

In this exercise, you will:

- create an Ice client to access a server that implements the filesystem developed in Assignment 1.

By the completion of this exercise, you will have gained experience in the Java language mapping, how to initialize and finalize the Ice run time, how to construct proxies, and how to invoke operations and handle exceptions.

Assignment 2 Creating an Ice Client
Copyright © 2005-2010 ZeroC, Inc.5-1

Notes:

In this exercise, you will create an Ice client to access a server that implements the file system we developed in Assignment 1.

5-1-1 Exercise Objectives

By the completion of this exercise, you will have gained experience in the Java language mapping, how to initialize and finalize the Ice run time, how to construct proxies, and how to invoke operations and handle exceptions.

5-2 Creating a Client for the Remote Filesystem

Creating a Client for the Remote Filesystem

- In your `lab2` directory, you will find a `build.xml` file to build a client and a server.
- The server is complete and implements the file system defined in `Filesystem.ice`.
- The server listens on port 10000 for incoming requests; the identity of the root directory object is “RootDir”.



Assignment 2 Creating an Ice Client
Copyright © 2005-2010 ZeroC, Inc.

5-2

Notes:

In your `lab2` directory, you will find a `build.xml` file to build a client and a server. The server is complete and implements the file system defined in `Filesystem.ice`. The server listens on port 10000 for incoming requests; the identity of the root directory object is “RootDir”.

5-3 What You Need to Do

What You Need to Do

The client code can be found in `Client.java`.

1. In the body of `main`, initialize the Ice run time, create a proxy to the root directory, and pass that proxy to the `listRecursive` function.
2. Following the call to `listRecursive`, shut down the Ice run time.
3. The body of `listRecursive` is empty, so you need to provide an implementation.
4. Test your client against the provided server.
5. Try running the client without first starting the server.
6. Change the client to use the identity "Fred" for the root directory.



Assignment 2 Creating an Ice Client
Copyright © 2005-2010 ZeroC, Inc.

5-3

Notes:

The client code can be found in `Client.java`. The code compiles as-is, but is incomplete. (The places in the code where you need to add something are marked with a `// MISSING` comment.)

1. In the body of `main`, initialize the Ice run time, create a proxy to the root directory, and pass that proxy to the `listRecursive` function.
2. Following the call to `listRecursive`, shut down the Ice run time. Be sure to handle and print any exceptions.
3. The body of `listRecursive` is empty, so you need to provide an implementation. `listRecursive` shows the contents of its `dir` argument: for each entry in the directory, it should print whether the entry is a file or a directory. If the entry is a file, `listRecursive` should show the name of the file and print the file's contents. If the entry is a directory, `listRecursive` should show the name of the directory and, recursively, show the contents of that directory.

The output of `listRecursive` should be somewhat like the output of an `ls -R` command, except that for files, `listRecursive` also prints the contents of each file. (You can use the `indent` variable to indent each line of output according to the depth of the recursion.)

Note that the `File::read` operation will occasionally raise an `IOError` exception. Your code should catch this exception and retry the `read` operation. (The server randomly raises `IOError` so the retry will eventually succeed.)

4. Test your client against the provided server. If your client works correctly, it should find two files and one directory underneath the root directory.
5. Try running the client without first starting the server. Does your client handle the situation correctly?
6. Change the client to use the identity “Fred” for the root directory. Is your client’s behavior acceptable in this case?

5-4 The main Method

The main Method

Note that the code catches and handles any exceptions, and that the communicator is destroyed only if it was successfully initialized.



Assignment 2 Creating an Ice Client
Copyright © 2005-2010 ZeroC, Inc.

5-4

Notes:

The main function for the client is as follows:

```
public static void
main(String[] args)
{
    int status = 0;
    Ice.Communicator ic = null;
    try
    {
        // Create a communicator
        //
        ic = Ice.Util.initialize(args);

        // Create a proxy for the root directory
        //
        Ice.ObjectPrx base = ic.stringToProxy("RootDir:default -p 10000");

        // Down-cast the proxy to a Directory proxy
        //
        DirectoryPrx rootDir = DirectoryPrxHelper.checkedCast(base);
    }
}
```

```
        if(rootDir == null)
            throw new RuntimeException("Invalid proxy");

        // Recursively list the contents of the root directory
        //
        System.out.println("Contents of root directory:");
        listRecursive(rootDir, 0);
    }
    catch(Ice.LocalException e)
    {
        e.printStackTrace();
        status = 1;
    }
    catch (Exception e)
    {
        e.printStackTrace();
        status = 1;
    }
    if(ic != null) {
        // Clean up
        //
        try
        {
            ic.destroy();
        }
        catch(Exception e)
        {
            e.printStackTrace();
            status = 1;
        }
    }
    System.exit(status);
}
```

Note that the code catches and handles any exceptions, and that the communicator is destroyed only if it was successfully initialized.

5-5 The listRecursive Method

The listRecursive Method

Note that the code, for each proxy returned by `list`, uses an `uncheckedCast` to down-cast the proxy. This avoids the overhead of using a `checkedCast`, which requires a remote message.



Assignment 2 Creating an Ice Client
Copyright © 2005-2010 ZeroC, Inc.

5-5

Notes:

```
static void
listRecursive(DirectoryPrx dir, int depth)
{
    char[] indentCh = new char[++depth];
    java.util.Arrays.fill(indentCh, '\t');
    String indent = new String(indentCh);

    NodeDetails[] contents = dir.list();

    for(int i = 0; i < contents.length; ++i)
    {
        System.out.println(indent + contents[i].name + "(" +
            contents[i].type == NodeType.DirT
                ? "directory" : "file") + "):");
        if(contents[i].type == NodeType.DirT)
        {
            DirectoryPrx subdir =
                DirectoryPrxHelper.uncheckedCast(contents[i].proxy);
            listRecursive(subdir, depth);
        }
        else
        {
            FilePrx file = FilePrxHelper.uncheckedCast(contents[i].proxy);

```

```
        boolean error = true;
        do
        {
            try
            {
                String[] text = file.read();
                for (int j = 0; j < text.length; ++j)
                {
                    System.out.println(indent + "\t" + text[j]);
                }
                error = false;
            }
            catch(IOException e)
            {
                System.out.println("Caught IO error: " + e.reason
                    + " Retrying...");
            }
        }
        while(error);
    }
}
```

Note that the code, for each proxy returned by `list`, uses an `uncheckedCast` to down-cast the proxy. This avoids the overhead of using a `checkedCast`, which requires a remote message.

6 Server-Side Java Mapping

6 Server-Side Slice-to-Java Mapping

6-1 Lesson Overview

Lesson Overview

- This lesson presents:
 - the mapping from Slice to Java relevant to the server side.
 - the relevant APIs that are necessary to initialize and finalize the Ice run time
 - how to implement and register object implementations.
- By the end of this lesson, you will be able to write a working Ice server.

Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.6-1

Notes:

This unit presents the mapping from Slice to Java relevant to the server side. It includes the relevant APIs that are necessary to initialize and finalize the Ice run time, and how to implement and register object implementations.

6-1-1 Lesson Objectives

By the end of this lesson, you will be able to write a working Ice server.

6-2 Server-Side Java Mapping

Server-Side Java Mapping

All of the client-side Java mapping also applies to the server side.

Additional server-side functionality you must know about:

- how to initialize and finalize the server-side run time
- how to implement servants
- how to pass parameters and throw exceptions
- how to create servants and register them with the run time



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-2

Notes:

The mapping of Slice types to Java types on the server side is identical to the client side, so everything that you have learned in Chapter 4 is valid for the server side as well.

However, for servers, we need to cover a few additional topics, such as the extra initialization steps and finalization steps that are required, as well as how to implement and create servants, how to register servants with the Ice run time, and how to pass parameters and throw exceptions.

6-3 Initializing the Ice Run Time

Initializing the Ice Run Time

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 1;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // server code here...
            status = 0;
        } catch (Exception e) {
        }
        if (ic) {
            try {
                ic.destroy();
            } catch (Java.Lang.Exception ex)
            {
            }
        }
        System.exit(status);
    }
}
```

Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.6-3

Notes:

The basic structure of the code in `main` for a server is identical to the client side: create a communicator by calling `Ice.Util.initialize` and, if initialization was successful, destroy the communicator again.

However, where the above code shows:

```
// server code here...
```

servers require some additional initialization and finalization steps.

6-4 Server-Side Initialization

Server-Side Initialization

Servers must create at least one object adapter, activate that adapter, and then wait for the Ice run time to shut down:

```
Ice = Ice.Util.initialize(args);
Ice.ObjectAdapter adapter
    = Ice.createObjectAdapterWithEndpoints(
        "MyAdapter", "tcp -p 10000");

// Instantiate and register one or more servants here...

adapter.activate();
Ice.waitForShutdown();
```

An object adapter provides one or more endpoints at which the server listens for incoming requests. An adapter has a name that must be unique within its communicator.

Adapters must be activated before they start accepting requests.

You must call `waitForShutdown` from the main thread to wait for the server to shut down (or otherwise prevent the main thread from exiting).



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-4

Notes:

Every server must create at least one object adapter. An object adapter has one or more endpoints at which the server listens for requests. Every adapter has a name (`MyAdapter`, in this example) which must be unique within the communicator, that is, no two adapters within a single communicator can have the same name. (In general, it is a good idea to ensure that adapter names are globally unique—see Section 12-9.)

Before an adapter can receive an incoming request from a client, you must activate it. Activating the adapter causes the server to listen for incoming connection requests at the endpoint specified for the adapter.

Internally, the Ice run time maintains one or more threads that accept and process incoming requests. To prevent your server's main thread from falling off the end of `main` and terminating the server process, you can pass the main thread to the Ice run time by calling `waitForShutdown` on the communicator. This call blocks the calling thread until the server has decided to terminate (possibly in response to a signal or in response to a request from a client). When `waitForShutdown` completes, you know that the server is idle, that is, that all incoming operation invocations have completed. Note however that calling `waitForShutdown` is not a requirement but rather simply a convenient way to prevent the main thread from terminating prematurely.

6-5 Mapping for Interfaces

Mapping for Interfaces

Interfaces map to skeleton classes with an abstract method for each Slice operation:

```
module M {
    interface Simple {
        void op();
    };
};
```

This generates:

```
package M;
public interface _SimpleOperations
{
    void op(Ice.Current current);
}
public interface Simple extends Ice.Object,
                               _SimpleOperations,
                               _SimpleOperationsNC;
public abstract class _SimpleDisp
    extends Ice.ObjectImpl
    implements Simple;
```



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-5

Notes:

Each Slice interface maps to a skeleton class with the name `_<name>Disp`. The skeleton class contains an abstract method for each operation in the Slice interface.

You create a servant class by deriving a concrete implementation from the abstract skeleton and implementing each inherited method. Once you have instantiated a servant and registered it with the Ice run time, a client calling operation `op` causes the server-side run time to transfer control to your concrete `op` method, thereby passing control from the Ice run time to your application. For example:

```
public final class SimpleI extends _SimpleDisp {
    public void op(Ice.Current c)
    {
        System.out.println("op was called");
    }
}
```

For the moment, you can ignore the trailing parameter of type `Ice.Current`. We discuss it in detail in Section 12-4.

Note that the `SimpleI` class is a fully functional servant class. If a client invokes the `op` operation on a proxy of type `Simple`, the server will print “op was called” on its standard output.

As for class implementations, servant classes use the interface name with an `I`-suffix. This is only a convention—you can give your servant classes any name you deem suitable.

6-6 Mapping for Interfaces (cont. 1)

Mapping for Interfaces (1)

You *must* implement all abstract methods that are inherited from the skeleton class.

You can add whatever else you need to support your implementation:

- constructors and finalizers
- public or private methods
- public or private data members
- other base interfaces



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-6

Notes:

As far as the Ice run time is concerned, you need only implement the methods that you inherit from the skeleton class. However, for all but the simplest servants, you will probably want additional functionality, such as constructors, member functions, data members, or additional base interfaces. You are free to add these as needed. For example, here is the previous servant implementation once more, using a constructor and a data member to customize the message that is printed when a client calls operation `op`:

```
public final class SimpleI extends _SimpleDisp {
    public SimpleI(String msg)
    {
        _msg = msg;
    }
    public void op(Ice.Current c)
    {
        System.out.println(_msg);
    }
    private String _msg;
}
```

6-7 Mapping for Parameters

Mapping for Parameters

Server-side operation signatures are identical to the client-side operation signatures (except for a trailing parameter):

- In-parameters are passed by value or by reference.
- Out-parameters are passed by holder types.
- Return values are passed by value or by reference.
- Every operation has a single trailing parameter of type `Ice.Current`.

```
string op(int a, string b, out float c, out string d);
```

Maps to:

```
String op(int a, String b,
          Ice.FloatHolder c, Ice.StringHolder d,
          Ice.Current __current);
```



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-7

Notes:

The parameter passing rules for the server side mirror those of the client side: in-parameters are passed by value or reference, out-parameters are passed as holder types, and return values are passed by value or reference.

An implementation of the above operation might look as follows:

```
public String op(int a, String b, Ice.FloatHolder c, Ice.StringHolder d,
                Ice.Current c)
{
    System.out.println("a = " + a); // In-parameters are initialized
    System.out.println("b = " + b); // by the client.

    c.value = 3.14;                // Return 3.14 to client.
    d.value = "param d";           // Return "param d" to client.

    return "return value"; // Return "return value" to client.
}
```

Again, there is nothing unusual here: you implement the operation just as you would write any other Java method with these parameter types.

6-8 Throwing Exceptions

Throwing Exceptions

```
exception GenericError { string reason; };
```

```
interface Example {
    void op() throws GenericError;
};
```

You could implement `op` as:

```
public void op(Ice.Current c) throws GenericError
{
    throw new GenericError("something failed");
}
```

Do not throw Ice run-time exceptions. You can throw **ObjectNotExistException**, **OperationNotExistException**, or **FacetNotExistException**, which are returned to the client unchanged. But these have specific meaning and should not be used for anything else.

If you throw any other run-time exception, the client will get an **UnknownLocalException** or **UnknownException**.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-8

Notes:

To throw an exception, you simply instantiate the exception and throw it.

Note that you must take care to only throw exceptions that are specified in an operation's exception specification. If you throw a user exception that is not part of the exception specification, the client receives an **UnknownUserException**.

Even though you can throw Ice run-time exceptions, you should avoid doing so. The only run-time exceptions that can be returned to a client are **ObjectNotExistException**, **OperationNotExistException**, and **FacetNotExistException**. However, these exceptions have very specific meaning and are normally thrown for you by the server-side run time when the corresponding error condition is detected. If you throw any other Ice run-time exception, it is returned to the client as an **UnknownLocalException**, which does not tell the client anything other than that something has failed.

If you allow a non-Ice exception (such as **ClassCastException**) to escape from an operation implementation, the client receives an **UnknownException**.

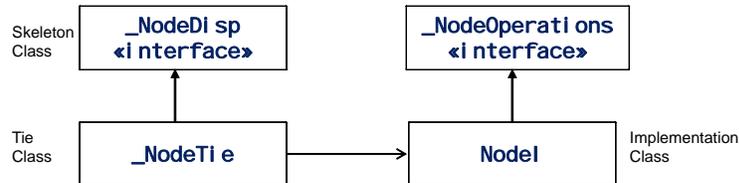
6-9 Tie Classes

Tie Classes

Tie classes are an alternative mechanism for implementing servants.

The `--tie` option for `slice2java` generates tie classes in addition to the normal server-side code.

Tie classes replace inheritance with delegation. This way, your implementation class need not inherit from the skeleton class:



Use the tie mapping when your implementation class must inherit from some other application class (and therefore cannot be derived from the skeleton class).



Notes:

If you add the `--tie` option to `slice2java`, the compiler generates an additional tie class for the server side. The tie class allows you to implement servants without having to derive the servant class from the skeleton class. The tie class delegates operation invocations to your implementation class, which, because it no longer derives from the skeleton class, is now free to derive from some other base class.

Here is the generated tie class for the `Simple` interface:

```

public class _SimpleTie extends _SimpleDisp implements Ice.TieBase
{
    public _SimpleTie();
    public _SimpleTie(_SimpleOperations delegate);

    public java.lang.Object ice_delegate();
    public void ice_delegate(java.lang.Object delegate);

    public boolean equals(java.lang.Object rhs);
    public int hashCode();

    public void op(Ice.Current __current);
}
  
```

```
}

```

As for the normal skeleton mapping, the tie contains one method for each Slice operation, and derives from `_SimpleDisp`. In other words, the generated tie class *is* the servant class; it forwards each operation invocation to your implementation class (the *delegate*). The constructor is overloaded so you can set the delegate at construction time. In addition, the `ice_delegate` methods allow you to set and get the delegate reference. When you set the delegate, you must pass a reference to an object that implements the `_SimpleOperations` interface, otherwise, the implementation throws a `ClassCastException`.¹

Using the generated tie, we can implement the delegate as follows:

```
public final class SimpleI implements _SimpleOperations
{
    public SimpleI(String msg)
    {
        _msg = msg;
    }

    public void op(Ice.Current __current)
    {
        System.out.println(_msg);
    }

    private String _msg;
}

```

Note that this is identical to the previous implementation, except that the delegate does not derive from `_SimpleDisp`.

To instantiate a tie and its delegate, you can write:

```
SimpleI s = new SimpleI("Hello");
M._SimpleTie servant = new M._SimpleTie(s);

```

Alternatively, you can default-construct the tie and set the delegate later:

```
_SimpleTie servant = new _SimpleTie(); // Create tie
// ...
SimpleI s = new SimpleI("Hello"); // Create delegate
// ...
servant.ice_delegate(s); // Set delegate

```

Note that, by default, there is no way to get back from the delegate to the tie. If you need to access the tie instance from within a delegate instance, you must arrange for this yourself, for example, by passing a reference to the tie to the constructor of the delegate.

¹ The formal parameter type is `java.lang.Object` because `ice_delegate` is defined in the `TieBase` class.

6-10 Creating an Object Adapter

Creating an ObjectAdapter

Each server must have at least one object adapter. You create an adapter with:

```
local interface ObjectAdapter;
local interface Communicator {
    ObjectAdapter createObjectAdapter(string name);
    ObjectAdapter createObjectAdapterWithEndpoints(
        string name,
        string endpoints);
    // ...
};
```

The endpoints at which the adapter listens are taken from configuration (first version), or from the supplied argument (second version).

Example endpoint specification:

```
tcp -p 10000:udp -p 10000:ssl -p 10001
```

Endpoints have the general form:

```
<protocol> [-h <host>] [-p <port>] [-t timeout] [-z]
```



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-10

Notes:

A server must have at least one object adapter. The adapter must have a name that is unique within its communicator. (No two adapters created on the same communicator can have the same name.)

`createObjectAdapter` looks at the configuration of the server to determine which endpoints to listen on. The endpoints are determined by the setting of the `<adapter-name>.Endpoints` property. (Refer to Chapter 8 for more information on properties.)

`createObjectAdapterWithEndpoints` uses the endpoints that are specified by the `endpoints` parameter.

The `-h` option in an endpoint specifies the interface on which the server listens for incoming requests. Omitting this option, or using `-h 0.0.0.0` or `-h *`, causes the server to listen on all interfaces, including the loopback interface (127.0.0.1).

The `-p` option specifies the port on which the server listens.

The `-t` option specifies a timeout for the object adapter's network activities as well as a default timeout for proxies created by this object adapter.

The `-z` option specifies that protocol compression can be used for invocations on this endpoint.

6-11 Object Adapter States

Object Adapter States

An object adapter is in one of three possible states:

- Holding (initial state after creation)
The adapter does not read incoming requests off the wire (for TCP and SSL) and throws incoming UDP requests away.
- Active
The adapter processes incoming requests. You can transition freely between the Holding and Active state.
- Inactive
This is the final state, entered when you initiate destruction of the adapter:
 - Requests in progress are allowed to complete.
 - New incoming requests are rejected with a `Connecti onRefusedExcepti on`.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-11

Notes:

After creation, an object adapter starts out in the holding state. While in that state, TCP and SSL requests are queued in the transport layer because the adapter does not read any requests off the wire. Note that leaving an adapter in the holding state for a long time can cause clients to eventually block, because TCP/IP flow control may eventually result in the client's transmit buffers filling up, with the client being suspended in a `write` system call. For UDP requests, it is operating system dependent how many UDP requests that were sent by clients are queued in the server's transport stack.

Typically, you first create an object adapter, then register your servants with it and perform any other necessary initialization, and then, once the server is fully initialized, activate the adapter to start the flow of requests.

The inactive state is the final state of an object adapter. It is not possible to transition from the inactive state to any of the other states. However, you can re-create a deactivated adapter after `destroy` has completed.

6-12 Controlling Adapter State

Controlling Adapter State

The following operations on the adapter relate to its state:

```
local interface ObjectAdapter {
    void activate();
    void hold();
    void deactivate();
    void waitForHold();
    void waitForDeactivate();
    void destroy();
    // ...
};
```

The operations to change state are non-blocking.

If you want to know when a state transition is complete, call `waitForHold` or `waitForDeactivate` as appropriate.

`destroy` blocks until deactivation completes.

You can re-create an adapter with the same name once `destroy` completes.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-12

Notes:

The adapter provides the `activate`, `hold`, and `deactivate` operations to allow you to control its state.

Note that the operations are non-blocking, that is, they initiate a state transition but do not wait for it to complete. If you want to know when a state transition is complete, you must call `waitForHold` or `waitForDeactivate`. These operations suspend the calling thread until the corresponding transition is complete. In both cases, the state transition completes once the last request in progress has completed, that is, once no more invocations are in progress.

You can call `waitForHold` or `waitForDeactivate` from multiple threads.

Calling `destroy` implicitly calls `deactivate` followed by `waitForDeactivate`, that is, `destroy` blocks until deactivation has completed. Once `destroy` returns, the object adapter no longer exists, so you can create another adapter with the same name thereafter.

6-13 Object Identity

Object Identity

Each Ice object has an associated object identity.

Object identity is defined as:

```
struct Identity {  
    string name;  
    string category;  
};
```

- The **name** member gives each Ice object a unique name.
- The **category** member is primarily used in conjunction with default servants and servant locators. If you do not use these features, the category is usually left as the empty string.

The identity must be unique within the object adapter: no two servants that incarnate an Ice object can have the same identity.

The *combination* of **name** and **category** must be unique.

An identity with an empty **name** denotes a null proxy.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-13

Notes:

Each Ice object has an object identity. The object identity is embedded in proxies for the object and, when clients make requests, is sent with the request. The object identity determines which particular servant a request is sent to and, therefore, must be unique within an object adapter.²

Unless you are using default servants or servant locators, the category field of an object identity will usually be empty. (See Chapter 18 for details on default servants and servant locators.)

² There are reasons for why you might want to have an object identity that is globally unique—we will discuss these in Chapter 12.

6-14 Stringified Object Identity

Stringified Object Identity

Two helper functions on the communicator allow you to convert between identities and strings:

```
interface Communicator
{
    Identity stringToIdentity(String id);
    String identityToString(Identity id);
    // ...
}
```

Stringified identities have the form `<category>/<name>`, for example:

`person/fred`

If no slash is present, the string is used as the name, with the category assumed to be empty.

The object adapter has a `getCommunicator` method that returns the communicator. You use the communicator to convert between strings and object identities.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-14

Notes:

Stringified identities have the general form `<category>/<name>`. If a stringified identity does not contain a slash, the string is used as the name, and the category is empty. If a category or name contains a slash, the slash must be escaped in the stringified identity. For example:

`Factories\ /Factory/Node\ /File`

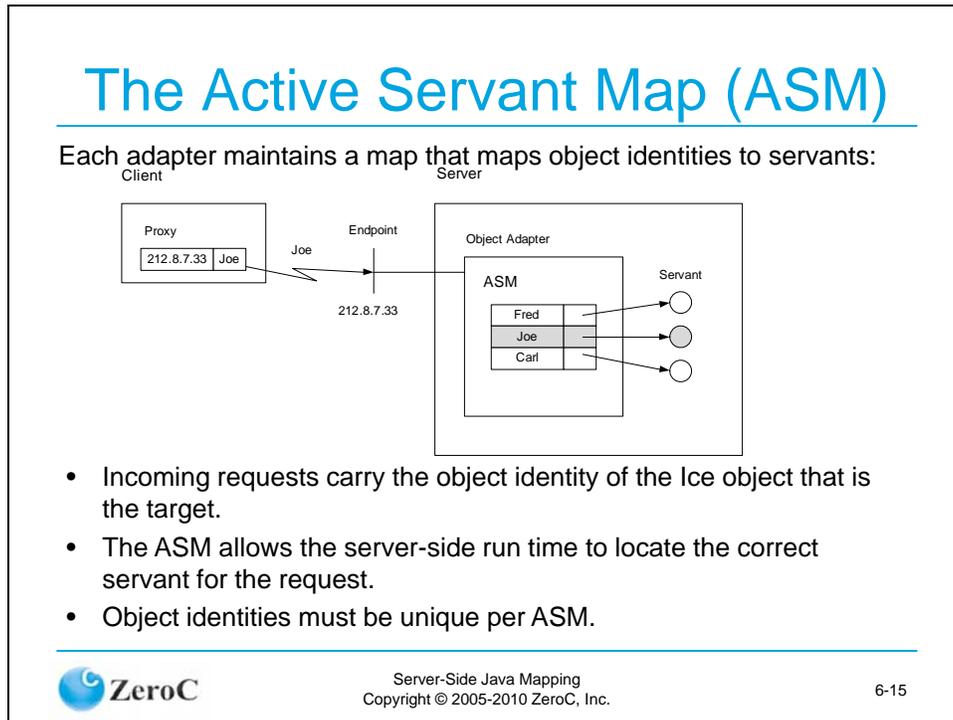
This is the identity with the category `Factories/Factory` and the name `Node/File`.

There are a number of other characters in stringified identities that must be escaped; you should use the `stringToIdentity` and `identityToString` helper functions for all identities that are not a simple sequence of printing characters.

Note that you can use the communicator to convert between strings and identities. The object adapter has a `getCommunicator` method that returns the communicator:

```
interface ObjectAdapter
{
    Communicator getCommunicator();
    // ...
}
```

6-15 The Active Servant Map (ASM)



Notes:

Each object adapter maintains a data structure known as the active servant map. The active servant map (or ASM, for short) is a lookup table that maps object identities to servants: the lookup value is a reference to the corresponding servant. When a client sends an operation invocation to the server, the request is targeted at a specific transport endpoint. Implicitly, the transport endpoint identifies the object adapter that is the target of the request (because no two object adapters can be bound to the same endpoint). The proxy via which the client sends its request contains the object identity for the corresponding object, and the client-side run time sends this object identity over the wire with the invocation. In turn, the object adapter uses that object identity to look in its ASM for the correct servant to dispatch the call to.

6-16 Activating Servants

Activating Servants

To make a servant available to the Ice run time, you must activate it. This adds an identity–servant pair to the ASM:

```
local interface ObjectAdapter {
    Object* add(Object servant, Identity Id);
    Object* addWithUUID(Object servant);
    // ...
};
```

Both operations return the proxy for the servant, for example:

```
SimplePrx sp = SimplePrxHelper.uncheckedCast(
    adapter.add(new SimpleI("hello"),
        adapter.getCommunicator().
            stringToIdentity("fred")));
```

As soon as a servant is added to the ASM, the run time will dispatch requests to it (assuming that the adapter is activated).

addWithUUID adds the servant to the ASM with a UUID as the name, and an empty category.



Notes:

Activating a servant means to add an identity–servant pair to the ASM. Requests are dispatched to the servant as soon as it is added (assuming that the adapter is active).

Both **add** and **addWithUUID** return a proxy to the servant, as type **ObjectPrx**. (You need not necessarily save the return value from these calls—you can create a proxy for an Ice object at any time, whether a servant is activated for that Ice object or not, but only if you know the object’s identity.) If you want to use the return value, you must either store it as type **ObjectPrx**, or down-cast it to the correct type (**SimplePrx** in this example).

Note that, because we know the type of the servant, there is no need to use a **checkedCast**—an **uncheckedCast** is sufficient and will always be correct (provided that you downcast to the correct type, of course).

addWithUUID adds an identity–servant pair with a UUID as the identity name, and the empty string as the identity category. Note that you can generate UUIDs with the **java.util.UUID.randomUUID()** method:

```
String uuid = java.util.UUID.randomUUID().toString();
```

6-17 Creating Proxies

Creating Proxies

You can create a proxy without activating a servant for the proxy:

```
interface ObjectAdapter {  
    // ...  
    Object* createProxy(Identity id);  
};
```

The returned proxy contains the passed identity and the adapter's endpoints.

Note that the return type is `Object*` so, typically, you need to downcast the proxy before you can use it.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-17

Notes:

`add` and `addWithUUID` return the proxy for a servant as you activate the servant. However, you can also create a proxy “out of thin air” without activating a servant, by calling `createProxy` on the object adapter.

The returned proxy contains the passed identity and the adapter's endpoints.

Creating proxies without activating a servant can be useful, for example, to implement collection manager operations that return sequences of proxies. `createProxy` is particularly useful if you use servant locators for lazy initialization (see Chapter 18).

6-18 The Ice::Application Class

The Ice.Application Class

Ice.Application is a utility class that makes it easy to initialize and finalize the Ice run time.

```
public abstract class Application
{
    public Application();
    public final int main(String appName, String[] args);
    public final int main(String appName, String[] args,
                          String configFile);
    public final int main(String appName, String[] args,
                          InitializationData id);
    public abstract int run(String[] args);
    public static Communicator communicator();
    public static String appName();
    // ...
}
```

You call **Application.main** from the real **main**, and implement the body of your client or server in the **run** method.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-18

Notes:

Initializing and finalizing the Ice run time is common to all Ice clients and servers, so Ice includes a utility class that provides this code. To use **Ice.Application**, you derive a class from it that implements the **run** method, for example:

```
public final class MyApplication extends Ice.Application {
    public int run(String[] args)
    {
        // Server code here...
    }

    public static void
    main(String[] args)
    {
        MyApplication app = new MyApplication();
        int status = app.main("MyApplication", args);
        System.exit(status);
    }
}
```

Using the `Application` class, your `main` function simply instantiates your derived class and returns the result of calling `Application.main` to the operating system. In turn, `Application.main` returns whatever status is returned by your implementation of `run`.

Before calling `run`, `Application.main` does the following:

1. It installs an exception handlers for `java.lang.Exception`:
If your `run` method fails to handle an exception and the stack gets wound back all the way to `Application.main`, `Application.main` catches the exception, prints an error message with the details of the exception on `stderr`, and returns a non-zero status.
`Application.main` ensures that the communicator is correctly destroyed before returning.
2. It initializes the Ice run time. You can get access to the communicator by calling the static `communicator` method.
3. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. (Ice command-line options start with `--Ice`, `--Freeze`, `--Glacier2`, etc.) The argument vector that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.
4. It stores the value of the `appName` parameter. You can get at that value by calling the static `appName` method from anywhere in your code. This is useful, for example, for error or trace messages that require the name of the program.
5. If instructed to handle signals, it creates a shutdown hook that behaves correctly in response to a signal.

We strongly recommend that you use `Ice.Application` in preference to writing your own code to initialize and finalize the Ice run time: `Ice.Application` ensures that the run time is finalized correctly under all circumstances, and it simplifies your code.

6-19 Shutdown Hook

Shutdown Hook

The `Ice.Application` provides control of the JVM shutdown hook:

```
package Ice;
public enum SignalPolicy { HandleSignals, NoSignalHandling }
public abstract class Application
{
    public Application();
    public Application(SignalPolicy signalPolicy);
    // ...
    synchronized public static void destroyOnInterrupt();
    synchronized public static void shutdownOnInterrupt();
    synchronized public static void defaultInterrupt();

    synchronized public static boolean interrupted();
}
```

The default behavior on interrupt is to destroy the communicator, allowing all currently running operation invocations to complete first.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-19

Notes:

`Ice.Application` also installs a JVM shutdown hook and provides methods that specify how the hook should behave:

- **`destroyOnInterrupt`**
This function causes the shutdown hook to first shut down the server-side run time (preventing new incoming invocations from being processed but allowing currently executing invocations to complete) and to then destroy the communicator. This is the default behavior when `Application` is instantiated using its default constructor, or when `HandleSignals` is passed to the constructor.
- **`shutdownOnInterrupt`**
This function causes the shutdown hook to shut down the server-side run time, but it does not destroy the communicator. New incoming invocations are rejected with an exception, but invocations that are executing when the signal arrives are allowed to complete.
- **`defaultInterrupt`**
This function restores the default behavior.
- **`interrupted`**

This function returns true if shutdown was caused by an exception or signal, false otherwise. This allows you to distinguish intentional shutdown from a forced shutdown, which is useful, for example, for logging purposes.

Note: `Ice.Application` is a singleton class: you can create only one instance of this class and, because `Ice.Application` creates a communicator, you cannot use it for applications that require more than one communicator. (Any additional communicators you create yourself would not be finalized correctly on receipt of a signal.)

6-20 Compiling and Running a Server

Compiling and Running a Server

To compile a client, you must:

- compile the Slice-generated source files(s)
- compile your application code

For Linux:

```
$ mkdir classes
$ javac -d classes -classpath \
> classes: $ICEJ_HOME/lib/Ice.jar \
> Server.java \
> generated/Demo/*.java
```

To compile and run the server, `Ice.jar` must be in your `CLASSPATH`.

Note that these commands are the same as for the client side—you need not supply server-specific options or use a server-specific class file or library.



Server-Side Java Mapping
Copyright © 2005-2010 ZeroC, Inc.

6-20

Notes:

To compile and run a server, you must compile the generated source code and your application code.

Note that you must have `Ice.jar` in your `CLASSPATH` to compile and run the server.

7 Assignment 3

Creating an Ice Server

7 Assignment 3: Creating an Ice Server

7-1 Exercise Overview

Exercise Overview

In this exercise, you will:

- create an Ice server that implements the filesystem we developed in Assignment 1.

By the end of this exercise, you will have gained experience in how to implement servants and how to use the `Ice.Application` class to initialize and finalize the Ice run time.

Assignment 3 Creating an Ice Server
Copyright © 2005-2010 ZeroC, Inc.7-1

Notes:

In this exercise, you will create an Ice server that implements the filesystem we developed in Assignment 1.

7-1-1 Exercise Objectives

By the end of this exercise, you will have gained experience in how to implement servants and how to use the `Ice.Application` class to initialize and finalize the Ice run time.

7-2 Creating a Server for the Remote Filesystem

Creating a Server for the Remote Filesystem

- In your `lab3` directory, you will find a `build.xml` file to build a client and a server.
- The client is complete and implements the solution shown in Assignment 2.
- Use this client to test your server.



Assignment 3 Creating an Ice Server
Copyright © 2005-2010 ZeroC, Inc.

7-2

Notes:

In your `lab3` directory, you will find a `build.xml` file to build a client and a server. The client is complete and implements the solution shown in Assignment 2. You will use this client to test your server.

7-3 What You Need to Do

What You Need to Do

1. Find `Server.java`, `Filesystem/FileI.java`, and `Filesystem/DirectoryI.java`. The places in the code where you need to add something are marked with a `// MISSING` comment.
2. Have a look at the code in `DirectoryI.java`. Add the new directory to the parent's `_contents` member and then add the new directory to the ASM.
3. Have a look at the code in `FileI.java`. Implement the `read` and `write` methods. Use the relevant member variable to store the contents of the file.
4. Implement the missing parts of `Server.java`.
5. Use the provided client to test your server and check that the contents of the file system are as expected.



Assignment 3 Creating an Ice Server
Copyright © 2005-2010 ZeroC, Inc.

7-3

Notes:

The server code can be found in `Server.java`, `Filesystem/FileI.java`, and `Filesystem/DirectoryI.java`. The places in the code where you need to add something are marked with a `// MISSING` comment.

1. Look at the code in `DirectoryI.java`.

Notice that the class implements the `NodeI` interface.

Each directory servant has the following member variables:

- `_name`
This variable stores the name of the directory. The name of the root directory is `"/`.
- `_parent`
This variable stores a reference to the parent directory. For the root directory (which does not have a parent directory), `_parent` is null.
- `_myID`
This variable stores the object identity of the servant.
- `_contents`

This variable stores a hash map that maps object identities to servants. For each file and directory contained in this directory, `_contents` contains a corresponding entry. The `list` operation uses this map so it can return the directory contents.

The constructor of `DirectoryI` initializes the `_name`, `_parent`, and `_myID` members. Note that the root directory has the fixed identity “RootDir”; all other directories use a UUID as the object identity.

Complete the body of the `activate` method by adding the directory as a child of the parent directory and adding the servant to the ASM. (The `addChild` method allows the child to add itself to the parent.)

Implement the body of `list`.

2. Look at the code in `FileI.java`.

Notice that the class implements the `NodeI` interface.

The implementation is analogous to `DirectoryI`. The constructor initializes the `_name`, `_parent`, and `_myID` member variables.

Implement the `activate` method.

The implementations of the `read` and `write` methods are missing. Implement these methods. Use the `_lines` member variable to store the contents of the file.

3. Implement the missing parts of `Server.java`:
 - Add the missing code to initialize the object adapter.
 - Add code to create a servant for the root directory and, within the root directory, create a `README` file and a `Coleridge` directory. Add some text to the `README` file. Inside the `Coleridge` directory, create a `Kubla_Khan` file, and add some text to that file. Remember to call `activate` on each servant.
4. Use the provided client to test your server and check that the contents of the file system are as expected.

7-4 The server Class

The Server Class

Assignment 3 Creating an Ice Server
Copyright © 2005-2010 ZeroC, Inc.7-4

Notes:

```
import Filesystem.*;
public class Server extends Ice.Application
{
    public int
    run(String[] args)
    {
        // Shut down cleanly on interrupt.
        //
        shutdownOnInterrupt();

        Ice.ObjectAdapter adapter = communicator().
            createObjectAdapterWithEndpoints("Lab3", "default -p 10000");

        // Create the root directory (with name "/" and no parent).
        //
        DirectoryI root = new DirectoryI("/", null);
        root.activate(adapter);

        // Create a file called "README" in the root directory.
        //
        File file = new FileI("README", root);
        String[] text = new String[]
        { "This file system contains a collection of poetry." };
    }
}
```

```
        try
        {
            file.write(text);
        }
        catch(IOException e)
        {
            // Implementation won't throw.
        }
        file.activate(adapter);

        // Create a directory called "Coleridge" in the root directory.
        //
        DirectoryI coleridge = new DirectoryI("Coleridge", root);
        coleridge.activate(adapter);

        // Create a file called "Kubla_Khan" in the Coleridge directory.
        //
        file = new FileI("Kubla_Khan", coleridge);
        text = new String[]{ "In Xanadu did Kubla Khan",
                             "A stately pleasure-dome decree:",
                             "Where Alph, the sacred river, ran",
                             "Through caverns measureless to man",
                             "Down to a sunless sea." };

        try
        {
            file.write(text);
        }
        catch(IOException e)
        {
            // Implementation won't throw.
        }
        file.activate(adapter);

        // All objects are created, allow client requests now.
        //
        adapter.activate();

        // Wait until we are done.
        //
        communicator().waitForShutdown();
        if(interrupted())
        {
            System.out.println(appName() + ": received signal, shutting down");
        }

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

7-5 The Directory Class

The Directory Class

Assignment 3 Creating an Ice Server
Copyright © 2005-2010 ZeroC, Inc.7-5

Notes:

```
package Filesystem;

public final class DirectoryI extends _DirectoryDisp implements NodeI
{
    public
    DirectoryI(String name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;
        _myID = parent != null ? java.util.UUID.randomUUID().toString() :
            "RootDir";
    }

    public String
    id()
    {
        return _myID;
    }

    public void
    activate(Ice.ObjectAdapter adapter)
    {

```

```

        Ice.Identity id = new Ice.Identity();
        id.name = _myID;
        adapter.add(this, id);
        if(_parent != null)
        {
            _parent.addChild(_name, this);
        }
    }

    public String
    name(Ice.Current c)
    {
        return _name;
    }

    public NodeDetails[]
    list(Ice.Current c)
    {
        NodeDetails[] s = new NodeDetails[_contents.size()];
        int i = 0;
        for(java.util.Map.Entry<Ice.Identity, NodeI> e :
            _contents.entrySet())
        {
            NodeI val = e.getValue();
            Ice.Identity id = new Ice.Identity();
            id.name = val.id();
            s[i] = new NodeDetails();
            s[i].type = val instanceof DirectoryI
                ? NodeType.DirT : NodeType.FileT;
            s[i].proxy = NodePrxHelper.uncheckedCast(
                c.adapter.createProxy(id));
            s[i].name = s[i].proxy.name();
            ++i;
        }
        return s;
    }

    public void
    addChild(String name, NodeI child)
    {
        _contents.put(name, child);
    }

    private String _name;
    private DirectoryI _parent;
    private String _myID;
    private java.util.HashMap<String, NodeI> _contents =
        new java.util.HashMap<String, NodeI>();
}

```

7-6 The FileI Class

The FileI Class

Assignment 3 Creating an Ice Server
Copyright © 2005-2010 ZeroC, Inc.7-6

Notes:

```
package Filesystem;

public class FileI extends _FileDisp implements NodeI
{
    public
    FileI(String name, DirectoryI parent)
    {
        _name = name;
        _parent = parent;
        _myID = java.util.UUID.randomUUID().toString();
    }

    public String
    id()
    {
        return _myID;
    }

    public void
    activate(Ice.ObjectAdapter adapter)
    {
        Ice.Identity id = new Ice.Identity();
    }
}
```

```
        id.name = _myID;
        adapter.add(this, id);
        if(_parent != null)
        {
            _parent.addChild(_name, this);
        }
    }

    public String
    name(Ice.Current c)
    {
        return _name;
    }

    public String[]
    read(Ice.Current c)
        throws IOError
    {
        return _lines;
    }

    public void
    write(String[] text, Ice.Current c)
        throws IOError
    {
        _lines = text;
    }

    private String _name;
    private DirectoryI _parent;
    private String _myID;
    private String[] _lines;
}
```

8 Properties and Configuration

8 Properties and Configuration

8-1 Lesson Overview

Lesson Overview

- This lesson presents:
 - how to use properties to control various aspects of the Ice run time.
 - how to use properties in your own applications.
- By completion of the chapter, you will know how the Ice run time can be configured using properties, how property values are evaluated, and how to use the property mechanism to configure your own applications.

Properties and Configuration
Copyright © 2005-2010 ZeroC, Inc.8-1

Notes:

Ice uses a simple configuration mechanism that allows you to control various aspects of the Ice run time. You can also use the same mechanism to configure your own applications.

8-1-1 Lesson Objectives

By completion of the chapter, you will know how the Ice run time can be configured using properties, how property values are evaluated, and how to use the property mechanism to configure your own applications.

8-2 Ice Properties

Ice Properties

The Ice run time and its various subsystems are configured using properties.

- Properties are name–value pairs, e.g.:
`Ice.UDP.SndSize=65535`
- By convention, Ice property names use the syntax
`<application>.<category>[.<sub-category>]`
For your own properties, you can use any number of categories and sub-categories (including none).
- Some property prefixes are reserved for Ice:
`Ice`, `IceBox`, `IceGrid`, `IcePatch2`, `IceSSL`, `IceStorm`, `Freeze`, and `Glacier2`.
- Property names are sequences of characters, excluding space (' '), hash ('#'), and ('=').
- Property values are sequences of characters, excluding hash ('#').



Notes:

Ice uses properties for its configuration. Properties are name–value pairs that are read by the Ice run time to configure various aspects of the run time, such as the maximum request size.

The property mechanism is extensible, so you can use it to configure your own applications.

By convention, Ice uses a period ('.') to separate property names into categories. This is a convention only—syntactically, the period has no special significance. For your own applications you can choose to follow the same convention and use categories for property names, or you can simply use a flat namespace for properties.

8-3 Configuration Files

Configuration Files

Properties are often set in a configuration file.

- Configuration files contain one property setting per line, e.g.:

```
# Example config file
Ice.MessageSizeMax = 2048 # 2MB message size limit
Ice.Trace.Network=3      # Trace all network activity
Ice.Trace.Protocol =    # No protocol tracing
```
- Leading and trailing white space around a property value are ignored, as are empty lines. Backslashes must be escaped as `\\`.
- The `#` character introduces a comment to the end of the current line.
- If a property is set more than once, the last setting takes effect.
- Assigning nothing to a property unsets the property.
- You can set the `ICE_CONFIG` environment variable to the path of a configuration file. The file is read when you create a communicator.
- Configuration files use UTF-8 encoding.



Properties and Configuration
Copyright © 2005-2010 ZeroC, Inc.

8-3

Notes:

You can define properties in a configuration file, using the syntax shown above. Note that leading and trailing white space for property values is ignored. (White space in the middle of a property value is preserved, however.)

If you set the `ICE_CONFIG` environment variable to the path of a configuration file, the file is read when you create a communicator. As a rule, you should use an absolute path name, otherwise the configuration file will not be found if the current working directory of an Ice process changes.

Note that you can create property names and values that contain spaces, `#`, or `=` characters. (See the Ice Manual for details on how to escape these characters.)

8-4 Setting Ice Properties on the Command Line

Setting Ice Properties on the Command Line

You can set Ice properties on the command line, e.g.:

```
java Server --Ice.UDP.SndSize=65535 --Ice.Trace.Network
```

- `--Ice.Trace.Network` is the same as `--Ice.Trace.Network=1`
- `--Ice.Trace.Network=` is the same as `--Ice.Trace.Network=' '`
- The `--Ice.Config` property determines the path of a configuration file: `--Ice.Config=/opt/Ice/default.config`
- `--Ice.Config` overrides the setting of the `ICE_CONFIG` environment variable.
- If you set properties on the command line, and the same properties are set in a configuration file, the properties on the command line override the ones in the configuration file.



Properties and Configuration
Copyright © 2005-2010 ZeroC, Inc.

8-4

Notes:

Apart from using a configuration file, you can also set Ice properties on the command line as shown.

If you simply mention the name of a property (that is, not follow the property name with an = character), the property is set to the value 1.

If you assign nothing to a property, the property is cleared.

The `--Ice.Config` property specifies the value of a configuration file. It overrides the setting of the `ICE_CONFIG` environment variable.

If you set properties in a configuration file and the same properties on the command line, the values on the command line take precedence.

If you specify several `--Ice.Config` options on the command line, only the last setting is used and the others are ignored.

If you want to use several configuration files, you can use the syntax

```
java Server --Ice.Config=/opt/Ice/default.config,/home/joe/Ice/default.config
```

(Note the comma separating the two configuration file names.) In this case, the run time reads the two configuration files in the order specified, so settings in the second file override settings in the first file. (You can use the same syntax for the `ICE_CONFIG` environment variable.)

8-5 Ice Initialization

Ice Initialization

`Ice.Util.Initialize` accepts an argument holder:
`Communicator Initialize(StringSeqHolder args);`

The function scans the argument vector for any Ice-specific options and returns an argument vector with those options removed.

Example:

```
java Server --Ice.Config=config --Ice.Trace.Network=3 -o f
```

After calling `Ice.Util.Initialize`, the cleaned-up vector contains:

```
-o f
```

You should parse the command line for your application *after* calling `Ice.Util.Initialize`. That way, you do not need to write code to skip Ice-related command-line options.

If you want the program name to appear in trace and log messages, set `Ice.ProgramName` before initializing the communicator.



Properties and Configuration
Copyright © 2005-2010 ZeroC, Inc.

8-5

Notes:

`Ice.Util.initialize` is overloaded: one version accepts a string array, whereas the other version accepts a `StringSeqHolder`. The latter version removes any Ice-specific options from the argument vector. By calling `Ice.Util.initialize` before parsing the command line, you avoid the need to write code to skip over Ice-specific command-line options.

Note that, if you want the name of the program to appear in trace and log messages, you must set `Ice.ProgramName` before initializing the communicator.

8-6 Reading Properties Programmatically

Reading Properties Programmatically

You can access property values programmatically:

```
dictionary<string, string> PropertyDict;
local interface Properties {
    string getProperty(string key);
    string getPropertyWithDefault(string key,
                                   string value);

    int getPropertyAsInt(string key);
    int getPropertyAsIntWithDefault(string key,
                                     int value);

    PropertyDict getPropertiesForPrefix(string prefix);
    // ...
};

local interface Communicator {
    Properties getProperties();
    // ...
};
```



Properties and Configuration
Copyright © 2005-2010 ZeroC, Inc.

8-6

Notes:

The communicator provides a `getProperties` operation that returns a smart pointer to a local `Properties` object. You can use this interface to read the value of properties (including properties for your own application). The operations behave as follows:

- **`getProperty`**
This operation returns the value of the specified property. If the property is not set, the operation returns the empty string.
- **`getPropertyWithDefault`**
This operation returns the value of the specified property. If the property is not set, the operation returns the supplied default value.
- **`getPropertyAsInt`**
This operation returns the value of the specified property as an integer. If the property is not set or contains a string that does not parse as an integer, the operation returns zero.
- **`getPropertyAsIntWithDefault`**
This operation returns the value of the specified property as an integer. If the property is not set or contains a string that does not parse as an integer, the operation returns the supplied default value.

- `getPropertiesForPrefix(string prefix)`

This operation returns all properties that begin with a specified prefix, such as “Filesystem”.

Note that the `Properties` interface contains a number of other operations for more advanced uses. (See the Ice manual for details.)

Here is how you could retrieve an application-specific property, `Filesystem.MaxFileSize` programmatically:

```
public static void
main(String[] args)
{
    Ice.StringSeqHolder ah = new Ice.StringSeqHolder(args);
    Ice.Communicator ic = Ice.Util.initialize(ah);
    args = ah.value;

    // Parse command line for application-specific options...

    Ice.Properties props = ic.getProperties();
    int ms = props.getPropertyAsInt("Filesystem.MaxFileSize");

    // ...
}
```

Note that, for an *application-specific* property to be retrieved with this code, it must be set in a configuration file. If the property is set on the command line instead, you need to explicitly parse the command line for application-specific properties first, as shown in Section 8-8.

8-7 Using InitializationData

Using InitializationData

Ice.Util.initialize is overloaded:

```
static Communicator initialize();
static Communicator initialize(String[] args);
static Communicator initialize(StringSeqHolder ah);
static Communicator initialize(InitializationData id);
static Communicator initialize(String[] args, InitializationData id);
static Communicator initialize(StringSeqHolder ah, InitializationData id);
```

```
final class InitializationData implements Cloneable
{
    public InitializationData();
    public java.lang.Object clone();
    public Properties properties;
    public Logger logger;
    public Stats stats;
    public ThreadNotification threadHook;
    public ClassLoader classLoader;
    public Dispatcher dispatcher;
}
```



Properties and Configuration
Copyright © 2005-2010 ZeroC, Inc.

8-7

Notes:

Ice.Util.initialize is overloaded to optionally accept an **InitializationData** instance. The fields of this structure allow you to customize a communicator, for example, with a specific logger or specific property values (by setting the **properties** member). You are required to do this when you create a communicator because, once set, the customizable aspects of the communicator remain in effect for the entire life time of the communicator, that is, they are immutable and cannot be changed once you have created a communicator.

You can pass application-specific properties to a communicator by setting the **properties** member of the **InitializationData** instance.

8-8 Command-Line Application Properties

Command-Line Application Properties

To be able to set application-specific properties on the command line, you must initialize a property set before you initialize the communicator:

```
public static void main(String[] args)
{
    Ice.InitializeData initData = new Ice.InitializeData();
    initData.properties = Ice.Util.createProperties();
    args = initData.properties.parseCommandLineOptions(
        "Filesystem", args);

    // Parse other application-specific options here...

    Ice.Communicator communicator =
        Ice.Util.initialize(args, initData);
}
```

- `createProperties` creates an empty property set.
- `parseCommandLineOptions` converts properties with the specified prefix, strips them from `args`, and returns the remaining arguments.



Properties and Configuration
Copyright © 2005-2010 ZeroC, Inc.

8-8

Notes:

If you use a property such as `Filesystem.MaxFileSize` and set the property value in a configuration file, the property will be set automatically when the configuration file is parsed.

However, if you want to be able to set the property on the command line, you must take extra steps:

1. Call `Ice.Util.createProperties` to create an empty property set.
2. Call `parseCommandLineOptions` on the property set, specifying the prefix of the property (or properties) you are interested in.

`parseCommandLineOptions` looks for option with the specified prefix. For example,

```
parseCommandLineOptions("Filesystem", args)
```

looks for command-line options beginning with `--Filesystem` and treats them as property settings. The values of these properties are added to the property set, and the returned string sequence contains any command-line options that were not converted to properties. You can call `parseCommandLineOptions` repeatedly to add properties with different prefixes.

Note that passing an empty prefix to `parseCommandLineOptions` causes it to parse *every* option with two leading dash (`--`) characters.

3. Pass an `InitializationData` instance with the initialized properties to `Ice.Util.initialize` to set the properties for the communicator.

8-9 Commonly-Used Ice Properties

Commonly-Used Ice Properties

- **Ice.Trace.Network** (0-3)
Trace network activity.
 - **Ice.Trace.Protocol** (0 or 1)
Trace protocol messages.
 - **Ice.Warn.Dispatch**
Print warnings for unexpected server-side exceptions.
 - **Ice.Warn.Connections** (0 or 1)
Print warnings if connections are lost unexpectedly.
 - **Ice.MessageSizeMax** (value in kB)
Set maximum size of messages that can be sent and received.
 - **Ice.ThreadPool.Server.Size**
Set the number of threads in the server-side thread pool.
- See the Ice manual for a complete list of properties.



Properties and Configuration
Copyright © 2005-2010 ZeroC, Inc.

8-9

Notes:

Above lists a few of the more commonly-used properties. Note that the trace and warning properties are particularly useful for debugging and should be the first thing you try if you find that a client cannot communicate with a server as expected.

8-10 Converting Properties to Proxies

Converting Properties to Proxies

A convenience operation on the communicator allows you to convert a property value to a proxy.

```
ObjectPrx p = communicator.propertyToProxy("App.Proxy");
```

This reads the stringified proxy from the property **App.Proxy**.

App.Proxy is the base name of the property. You can define additional aspects of the proxy in separate subordinate properties. For example:

- **App.Proxy.Coll ocati onOpti mi zed**
- **App.Proxy.Connecti onCached**
- **App.Proxy.Endpoi ntSel ecti on**

The subordinate properties of the property group define the local behavior of the proxy, such as how to select endpoints, prefer secure transports over non-secure ones, and so on.



Notes:

The `propertyToProxy` convenience operation on the communicator allows you to specify the name of a property from which to read a stringified proxy and to convert that string into a proxy.

This makes it easy to place stringified proxies into configuration files and convert them back to proxies without having to first read the property and then call `stringToProxy`.

Note that you can use subordinate properties to specify additional local behavior for a proxy, such as how the proxy selects its endpoints. (See the Ice manual for more details on the available options.)

8-11 Object Adapter Properties

Object Adapter Properties

Object adapters support a number of configuration properties.

The adapter's name is used as the prefix for its properties:

```
ObjectAdapter adapter =  
    communicator.createObjectAdapter("MyAdapter");
```

Commonly-used adapter properties:

- `MyAdapter.AdapterId`
- `MyAdapter.Endpoints`
- `MyAdapter.ProxyOptions`
- `MyAdapter.PublishedEndpoints`
- `MyAdapter.Router`
- `MyAdapter.ThreadPool`

Properties must be defined in the communicator's property set prior to calling `createObjectAdapter`.



Notes:

Object adapters support a number of configuration properties that allow you to customize their behavior. The name that you assign to an object adapter when calling `createObjectAdapter` is used as the prefix for these configuration properties. The most common adapter property is `<adapter>.Endpoints`, which specifies the address(es) on which the adapter listens for incoming connections. Please refer to the Ice manual for more information on configuring an object adapter.

Note that it is not strictly necessary to define your adapter's properties prior to initializing a communicator. However, the properties must be defined prior to calling `createObjectAdapter` and, once the adapter is created, subsequent changes to its properties have no effect.

9 Assignment 4 Using Properties

9 Assignment 4: Using Properties

9-1 Exercise Overview

Exercise Overview

In this exercise, you will:

- modify the client we created in Assignment 2 to use application-specific properties.

By the end of this exercise, you will have gained experience in how to use properties to configure the Ice run time as well as your own applications.

Assignment 4 Using Properties
Copyright © 2005-2010 ZeroC, Inc.9-1

Notes:

In this exercise, you will modify the client we created in Assignment 2 to use application-specific properties.

9-1-1 Exercise Objectives

By the end of this exercise, you will have gained experience in how to use properties to configure the Ice run time as well as your own applications.

9-2 Adding an Application-Specific Property

Adding an Application-Specific Property

- In your `lab4` directory, you will find a `build.xml` file to build a client and a server.
- Both client and server are complete.
- You will modify the client to use application-specific properties.



Assignment 4 Using Properties
Copyright © 2005-2010 ZeroC, Inc.

9-2

Notes:

In your `lab4` directory, you will find a `build.xml` file to build a client and a server. Both client and server are complete. You will modify the client to use application-specific properties.

9-3 What You Need to Do

What You Need to Do

1. Modify the client such that it picks up its property settings from a configuration file. Add the missing initialization of `base` to denote the proxy to the root directory.
2. Modify the `run` method such that it retrieves the property setting and sets the `_showSize` member variable accordingly.
3. Create a configuration file `config` and add a setting for both properties to it.
4. Modify `main` such that you can invoke the client.
5. Change the proxy for the root directory to port 9999 and run the client.
6. Change the proxy for the root directory to use port 10000 again. Now run the client with `--Ice.Trace.Protocol=1`.



Assignment 4 Using Properties
Copyright © 2005-2010 ZeroC, Inc.

9-3

Notes:

The Slice definitions in `Filesystem.ice` for this exercise have been modified by adding a `size` operation to the `Node` interface. For a file, this operation returns the number of characters in the file; for a directory, it returns the number of entries in the directory.

You will modify the client code to react to the setting of two new properties,

`Filesystem.RootDir` and `Filesystem.ShowSize`.

- `Filesystem.RootDir` contains the stringified proxy for the root directory. The server listens for incoming requests on port 10000, and the root directory has the object identity `RootDir`.
- The `Filesystem.ShowSize` property controls the behavior of `listRecursive`. If that property is set to zero, the behavior of `listRecursive` is unchanged. If the property is set to 1, `listRecursive` will, for each file and directory, also display its size.

The implementation of `listRecursive` uses the setting of member variable `_showSize` to determine whether or not to show the size information.

(`listRecursive` already contains this change, so you do not need to modify the function.)

1. Initially, you will modify the client such that it picks up its property settings from a configuration file. Add the missing initialization of `base` to denote the proxy to the root directory.
2. Modify the `run` method such that it retrieves the property setting and sets the `__showSize` member variable accordingly.
3. Create a configuration file `config` and add a setting for both properties to it. Run the client and verify that it does not show the size information if the property is set to zero, and that it does show the size information if the property is set to one. What options do you have for getting the client to load the configuration file?
4. As is, the client code cannot handle setting the filesystem properties on the command line. Modify `main` such that you can invoke the client as:

```
java Client --Filesystem.ShowSize=1 \  
--Filesystem.RootDir="RootDir:default -p 10000"
```

and have it react to the property settings.
5. Change the proxy for the root directory to port 9999 and run the client. You will see a message about a refused connection.
Stop the server and restart it with `--Ice.Trace.Network=2`. Now run the client again, also with `--Ice.Trace.Network=2`. On which run did you find it easier to work out what is going wrong?
6. Change the proxy for the root directory to use port 10000 again. Now run the client with `--Ice.Trace.Protocol=1`. Read the trace messages that are produced. Can you find any messages being exchanged that you did not expect? What part of the client is responsible for the exchange of these messages?

9-4 Using Properties

Using Properties

The missing line of code to initialize the `_showSize` member is:

```
_showSize = communicator().getProperties().
    getPropertyAsInt("Filesystem.ShowSize") != 0;
```

The code to add at the beginning of `main` so the properties can be set on the command line is:

```
Ice.InitializationData initData = new Ice.InitializationData();
initData.properties = Ice.Util.createProperties();
args = initData.properties.parseCommandLineOptions("Filesystem",
args);
```



Assignment 4 Using Properties
Copyright © 2005-2010 ZeroC, Inc.

9-4

Notes:

The missing line to initialize the proxy to the root directory is:

```
Ice.ObjectPrx base =
    communicator().propertyToProxy("Filesystem.RootDir");
```

The missing line of code to initialize the `_showSize` member is:

```
_showSize = communicator().getProperties().
    getPropertyAsInt("Filesystem.ShowSize") != 0;
```

The code to add at the beginning of `main` so the properties can be set on the command line is:

```
Ice.InitializationData initData = new Ice.InitializationData();
initData.properties = Ice.Util.createProperties();
args = initData.properties.parseCommandLineOptions("Filesystem", args);
```

When running the client with `--Ice.Trace.Protocol=1`, you will find `ice_isA` messages being sent by the client. These are caused by the calls to `checkedCast` in the client code: during a safe down-cast, the Ice run time sends a message to the server to confirm whether the object denoted by the proxy supports the requested interface.

10 Multi- Threaded Ice

10 Multi-Threaded Ice

10-1 Lesson Overview

Lesson Overview

- This lesson presents:
 - the threading models available with the Ice run time and how to configure them.
 - some general threading strategies that you can use in your servers
- By the completion of the chapter, you will understand how the Ice run time uses threads and how to implement a simple thread-safe server.

Multi-Threaded Ice
Copyright © 2005-2010 ZeroC, Inc.10-1

Notes:

This chapter presents the threading models available with the Ice run time and how to configure them. This chapter also presents some general threading strategies that you can use in your servers.

10-1-1 Lesson Objectives

By the completion of the chapter, you will understand how the Ice run time uses threads and how to implement a simple thread-safe server.

10-2 Ice Threading Model

Ice Threading Model

Ice uses a thread pool concurrency model.

For each communicator, Ice maintains:

- a client-side thread pool to process replies for outgoing requests and to dispatch incoming requests on bi-directional connections.
- a server-side thread pool to dispatch incoming requests.

You can create additional per-adapter thread pools.

The default size for both client- and server-side thread pools is 1.



Multi-Threaded Ice
Copyright © 2005-2010 ZeroC, Inc.

10-2

Notes:

Ice uses a thread pool concurrency model.

- The run time creates one client-side thread pool per communicator (by default, with a single thread). The client-side thread pool is used to dispatch incoming requests on bidirectional connections, so clients can receive callbacks. The client-side thread pool is also used to handle replies for outgoing invocations.
- The run time creates one server-side thread pool per communicator (by default, with a single thread) to dispatch incoming requests.

The default thread pools are shared by all object adapters created by the communicator. You can also configure a per-adapter thread pool that is used to dispatch requests to the corresponding object adapter.

10-3 Thread Pool Configuration

Thread Pool Configuration

By default, the client- and server-side thread pools contain a single thread.

You can configure the pool size:

- `Ice.ThreadPool.Client.Size=<num>`

The client-side thread pool can normally be left at 1, unless you need to support concurrent asynchronous or bi-directional callbacks (or if these callbacks might block).

- `Ice.ThreadPool.Server.Size=<num>`

The server-side thread pool determines how many requests can be processed concurrently by the server.

Both properties set the initial number of threads in the pool.



Multi-Threaded Ice
Copyright © 2005-2010 ZeroC, Inc.

10-3

Notes:

You can control the initial sizes of the client- and server-side thread pools with the properties `Ice.ThreadPool.Client.Size` and `Ice.ThreadPool.Server.Size`. The thread pools are created when you create a communicator and destroyed when you destroy a communicator, although the number of threads in a pool can change over time as directed by the pool's configuration.

Note that, unless you need asynchronous or bi-directional callbacks to run concurrently, you can leave the client-side thread pool size at 1.

10-4 Thread Pool Configuration (cont. 1)

Thread Pool Configuration (1)

Thread pools initially contains the number of threads specified by

`Ice.ThreadPool.Client.Size` and

`Ice.ThreadPool.Server.Size`.

You can also set a maximum size:

- `Ice.ThreadPool.Client.SizeMax=<num>`
- `Ice.ThreadPool.Server.SizeMax=<num>`

These properties allow a thread pool to temporarily grow larger than its initial size due to increased demand.

During idle periods, the size of a pool can shrink to just one thread. The idle timeout is specified by

`Ice.ThreadPool.Client.ThreadIdleTime` and

`Ice.ThreadPool.Server.ThreadIdleTime`.



Multi-Threaded Ice
Copyright © 2005-2010 ZeroC, Inc.

10-4

Notes:

The `Ice.ThreadPool.Client.SizeMax` and `Ice.ThreadPool.Server.SizeMax` properties permit a thread pool to grow larger than its initial size due to increased demand. The run time grows a thread pool immediately as the load increases up to the pool's maximum size, then reaps threads after they have been idle for a configurable period (the default is 60 seconds). The `Ice.ThreadPool.Client.ThreadIdleTime` and `Ice.ThreadPool.Server.ThreadIdleTime` properties determine how quickly idle threads are reaped. During idle periods, the size of a pool can shrink to just one thread.

10-5 Thread Pool Configuration (cont. 2)

Thread Pool Configuration (2)

- `Ice.ThreadPool.Client.SizeWarn=<num>`
`Ice.ThreadPool.Server.SizeWarn=<num>`
These properties log a warning once the number of threads in a pool exceeds the specified threshold.
- `Ice.ThreadPool.Client.StackSize=<bytes>`
`Ice.ThreadPool.Server.StackSize=<bytes>`
These properties set the stack size of the threads in a pool (byte units).
The default value is zero, which gives threads the default stack size as determined by the OS.



Multi-Threaded Ice
Copyright © 2005-2010 ZeroC, Inc.

10-5

Notes:

You can set `Ice.ThreadPool.Client.SizeWarn` and `Ice.ThreadPool.Server.SizeWarn` to log a warning once the number of threads in a pool exceeds the specified threshold. The warning is written using whatever logger is set on the corresponding communicator.

You can set `Ice.ThreadPool.Client.StackSize` and `Ice.ThreadPool.Server.StackSize` to change the stack size for threads in a pool to a value other than the OS-assigned default. This is useful for large thread pools because the default size (usually 64MB) can cause the process to exceed its virtual memory allocation.

10-6 Thread Safety

Thread Safety

All APIs in the Ice run time are thread safe:

- You never have to lock something against concurrent access on behalf of the run time.
- Ice run-time APIs are deadlock free, so you can call any Ice API at any time and from any thread without fear of deadlock.

Exception:

Do not call `waitForShutdown`, `waitForDeactivate`, or `waitForHold` from within an executing operation on the corresponding adapter. If you do, you *will* deadlock.

Access to collections (sequences and dictionaries) is *not* interlocked. If you manipulate the same collection concurrently from different threads, you must establish a critical region yourself.

For multi-threaded servers, you must protect your own application-specific data against concurrent access.



Multi-Threaded Ice
Copyright © 2005-2010 ZeroC, Inc.

10-6

Notes:

In general, the Ice run time is completely thread-safe, that is, you never need to lock anything on behalf of the run time. Similarly, the run time is free of deadlocks, except for nonsensical situations, such as calling `waitForShutdown` from within an operation implementation on the corresponding object adapter.

Sequence and dictionary accesses are not interlocked because they are mapped to language-specific collections that do not use concurrency control (such as Java arrays or hash maps). If you concurrently manipulate the same collection from different threads, you must interlock yourself as appropriate.

Of course, you must establish critical regions around accesses to your own application-specific data, as you would for any multi-threaded program.

11 Assignment 5

Thread Safety

11 Assignment 5: Thread Safety

11-1 Exercise Overview

Exercise Overview

In this exercise, you will:

- modify the server we created in Assignment 3 to be thread-safe.

By the completion of this exercise, you will have gained experience in how to use synchronization to make a server implementation thread-safe, and will know how to create threads to make concurrent invocations.

Assignment 5 Thread Safety
Copyright © 2005-2010 ZeroC, Inc.11-1

Notes:

In this exercise, you will modify the server we created in Assignment 3 to be thread-safe.

11-1-1 Exercise Objectives

By the completion of this exercise, you will have gained experience in how to use synchronization to make a server implementation thread-safe, and will know how to create threads to make concurrent invocations.

11-2 Thread Safety

Thread Safety

- The file system server is not thread-safe.
- Modify the server to support concurrent invocations by clients and modify the client to make concurrent invocations on the server.

Assignment 5 Thread Safety
Copyright © 2005-2010 ZeroC, Inc.11-2

Notes:

As it stands, the file system server is not thread-safe. In this step, you will modify the server to support concurrent invocations by clients and modify the client to make concurrent invocations on the server.

11-3 What You Need to Do

What You Need to Do

1. Modify the server to provide mutual exclusion.
2. Add trace statements to the beginning and end of `list`: your code should print the calling thread's ID as it enters and leaves `list`.
3. For testing purposes, add a statement to `list` that causes the calling thread to sleep for one second.
4. Modify the client to create three threads, each of which will call `listRecursive`.
5. At the end of `Client.run`, add code to create three threads of type `ListThread`.
6. Add code to join with the three threads you created in step 5.
7. Run client and server in separate windows and examine the trace produced by each program.



Assignment 5 Thread Safety
Copyright © 2005-2010 ZeroC, Inc.

11-3

Notes:

-
-
-
-
1. Modify the server to provide mutual exclusion. Use per-servant locking: clients can concurrently execute operations on different nodes in the file system, but concurrent invocations on the same node are serialized.
 2. Add a trace statement to the beginning and end of `list`: your code should print the calling thread's ID as it enters and leaves `list`. Be sure to add the trace *before* acquiring any lock so, instead of making the entire method synchronized, use a synchronized block inside the method.
 3. For testing purposes, add a statement to `list` that causes the calling thread to sleep for one second. Insert this statement *after* acquiring the lock. (We need to artificially slow down the execution so we can check by looking at the trace statement that concurrency works as expected.)
 4. In this step, you will modify the client to create three threads, each of which will call `listRecursive`. Examine `Client.java`. You will notice that the structure of the client has changed somewhat:
 - The client contains a new class, `ListThread`. This class is the thread class whose `run` method becomes the start frame of the threads created by the client.
 - The `listRecursive` method has moved from the `Client` class to the `ListThread` class.

Implement the `run` method of `ListThread`.

5. At the end of `Client.run`, add code to create three threads of type `ListThread`. Add a trace statement that prints the ID of each thread.
6. Add code to join with the three threads you created in step 5. Add trace statements to show the ID of each thread when you call `join`, and to show the ID again when `join` completes.
7. Run client and server in separate windows and examine the trace produced by each program. Check whether invocations of `list` in the server are indeed dispatched concurrently. (Use the thread IDs to determine whether more than one thread calls `list` while another thread is still sleeping inside `list`.) If you find that only one thread enters `list` at a time, fix your server such that it provides true concurrency.

11-4 Server Modifications

Server Modifications

- To make the server thread-safe, we need to make Slice operations synchronized.
- The operations for which this is necessary are **read**, **write** and **list**.
- It is also necessary to acquire a lock in **addChild**: without this lock, if the server concurrently instantiates nodes from different threads, the **_contents** member of the parent can be corrupted.



Assignment 5 Thread Safety
Copyright © 2005-2010 ZeroC, Inc.

11-4

Notes:

To make the server thread-safe, we need to make Slice operations synchronized. The operations for which this is necessary are **read**, **write**, and **list**. The **name** operation need not be synchronized because it returns an immutable value. (Strictly speaking, we could leave synchronization off the **list** operation because the **_contents** member is not modified after the server initializes the servants. However, if the server adds servants while it is running, the synchronization is necessary.) In addition, it is necessary to acquire a lock in **addChild**: without this lock, if the server concurrently instantiates nodes from different threads, the **_contents** member of the parent can be corrupted.

The **DirectoryI** implementation changes as follows.

```
public final class DirectoryI ...
{
    public
    DirectoryI(String name, DirectoryI parent)
    {
        // ...
    }

    public String
    id()
    {
        return _myID;
    }
}
```

```

    }

    public void
    activate(Ice.ObjectAdapter adapter)
    {
        // ...
    }

    public String
    name(Ice.Current c)
    {
        return _name;
    }

    public NodeDetails[]
    list(Ice.Current c)
    {
        Thread t = Thread.currentThread();

        System.out.println(t.getName() + " entering list");

        NodeDetails[] s;
        synchronized(this)
        {
            while(true)
            {
                try
                {
                    t.sleep(1000);
                    break;
                }
                catch(InterruptedException ex)
                {
                    // Ignored
                }
            }

            s = new NodeDetails[_contents.size()];
            int i = 0;
            for(java.util.Map.Entry<Ice.Identity, Node> e :
                _contents.entrySet())
            {
                Node val = e.getValue();
                Ice.Identity id = new Ice.Identity();
                id.name = val.id();
                s[i] = new NodeDetails();
                s[i].type = val instanceof DirectoryI ?
                    NodeType.DirT : NodeType.FileT;
                s[i].proxy = NodePrxHelper.uncheckedCast(
                    _adapter.createProxy(id));
                s[i].name = s[i].proxy.name();
                ++i;
            }
        }

        System.out.println(t.getName() + " leaving list");

        return s;
    }

    public synchronized void
    addChild(Ice.Identity id, Object child)
    {

```

```
        _contents.put(id, child);
    }
    // ...
}
```

Note that, normally, `list` would be a synchronized method. We use a synchronized block instead only so we can verify via the trace statements that the server indeed dispatches concurrent operation invocations in separate threads.

The `FileI` implementation changes as follows:

```
public class FileI ...
{
    public
    FileI(String name, DirectoryI parent)
    {
        // ...
    }

    public String
    id()
    {
        return _myID;
    }

    public void
    activate(Ice.ObjectAdapter adapter)
    {
        // ...
    }

    public synchronized String
    name(Ice.Current c)
    {
        return _name;
    }

    public synchronized String[]
    read(Ice.Current c)
        throws IOError
    {
        return _lines;
    }

    public synchronized void
    write(String[] text, Ice.Current c)
        throws IOError
    {
        _lines = text;
    }

    // ...
}
```

Note: For the server to actually run multi-threaded, you need to increase the thread pool size because the default size is 1. One way to do this is to run the server with `--Ice.ThreadPool.Server.Size=5`.

11-5 Client Modifications

Client Modifications

- The client simply creates three threads that each call `listRecursive` and joins with these threads.



Notes:

The client simply creates three threads that each call `listRecursive` and joins with these threads:

```
public class Client extends Ice.Application
{
    public int run(String[] args)
    {
        // ...

        Thread[] threads = new Thread[3];
        for(int i = 0; i < threads.length; ++i)
        {
            threads[i] = new ListThread(rootDir, "thread " + i);
            threads[i].start();
            System.out.println("Started " + threads[i].getName());
        }

        for(int i = 0; i < threads.length; ++i)
        {
            System.out.println("Joining with " + threads[i].getName());
            while(true)
            {
                try
                {
```

```
        threads[i].join();
        break;
    }
    catch(InterruptedException ex)
    {
        // Ignored
    }
}
System.out.println("Joined with " + threads[i].getName());
}
return 0;
}
// ...
}
```

12 Object Life Cycle

12 Object Life Cycle

12-1 Lesson Overview

Lesson Overview

- Object life cycle refers to the issues that surround creation and destruction of objects.
- This lesson shows you how you can create and destroy Ice objects in response to client requests, and how to ensure that these operations are thread-safe. The lesson also discusses issues regarding the uniqueness of object identities, and how to deal with objects that are abandoned by clients.
- By the completion of this lesson, you will have a thorough understanding of how to provide life cycle operations in a thread-safe manner.



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-1

Notes:

Object life cycle refers to the issues that surround creation and destruction of objects. This lesson shows you how you can create and destroy Ice objects in response to client requests, and how to ensure that these operations are thread-safe. The chapter also discusses issues regarding the uniqueness of object identities, and how to deal with objects that are abandoned by clients.

12-1-1 Lesson Objectives

By the completion of this lesson, you will have a thorough understanding of how to provide life cycle operations in a thread-safe manner.

12-2 Object Creation

Object Creation

Object creation typically relies on the factory pattern:

```
exception NameInUse {};
```

```
interface Directory extends Node {
    Directory* createDir(string name)
        throws NameInUse;
};
```

- The factory operation creates a new Ice object as a side-effect and returns the proxy to the newly-created object.
- As far as the Ice run time is concerned, a factory operation is no different from any other operation.
- The factory operation behaves like a constructor and can accept whatever arguments are necessary to create the new object.
- Often, factory operations also throw exceptions to indicate errors that might be caused by invalid arguments or that are detected by the operation implementation.



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-2

Notes:

The canonical way to create a new Ice object is to provide a factory operation on some interface.

The factory operation creates a new Ice object and returns its proxy.

You can have object factories with more than one create operation, for example, to cater for alternative ways of creating an Ice object (requiring different parameters), or to allow for the creation of more than one type of object. As far as the Ice run time is concerned, a factory operation is no different from any other operation. It is the *implementation* of that operation that causes a new Ice object to spring into existence as a side effect.

You should always write factory operations such that they completely initialize an object. Avoid interfaces that use one operation to create an object and another operation to initialize that object; such designs are brittle because they make it possible to end up with uninitialized or partially initialized objects.

Other variations on the factory pattern are possible. For example, you can define operations that create a number of objects instead of a single one, and you need not necessarily return the proxy to the newly-created objects, provided the client then has some other way to obtain the proxies to these objects later (for example, from a collection manager interface).

Implementation of the factory operation is usually trivial. The easiest way to create a new Ice object is to instantiate a new servant, initialize the servant, and add it to the active servant map:

```
public DirectoryPrx
createDir(String name, Ice.Current c)
{
    // Check parameters.
    //
    if(!nameIsValid(name))
    {
        throw new NameInUse();
    }

    // Instantiate a servant.
    //
    Directory d = new DirectoryI(name, this);

    // Add servant to ASM.
    //
    Ice.ObjectPrx o = c.adapter.addWithUUID(d);

    // Return proxy.
    //
    return DirectoryPrxHelper.uncheckedCast(o);
}
```

The crucial step in this code is to add a servant to the ASM. (This is known as *servant activation*.) Note that we use `addWithUUID` here to create a unique identity for the new Ice object. Alternatively, depending on your application, you may have some servant state that is unique for each Ice object and that can be used as the object identity instead of a UUID.

The code uses an `uncheckedCast` to down-cast the proxy for the new object to `DirectoryPrx`. This is guaranteed to work because we know that the object just added to the ASM is indeed a directory.

The `nameIsValid` method checks whether a file or directory with the same name exists already (not shown).

Note the use of the `Current` object that is passed to the operation. In this case, we use it to get access to the object adapter for the directory servant, in order to call `addWithUUID`. (We discuss the `Current` object in detail in Section 12-4.)

12-3 Object Creation and Thread Safety

Object Creation and Thread Safety

If clients can call `create` concurrently, you must interlock:

```
public synchronized FilePrx
createFile(String name, Ice.Current c)
    throws NameInUse
{
    if(!nameIsValid(name))
    {
        throw NameInUse();
    }

    // Instantiate servant, add to ASM,
    // and return proxy here...
}
```



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-3

Notes:

For threaded applications, the factory operation usually requires a lock because clients may attempt to create the same object with the same identity concurrently. It is usually easiest to retain the lock for the entire `create` operation because object creation tends to be rare.

If you need better concurrency (for example, because `create` is long-running), you can release the lock once you have established that object creation will succeed, and once you have taken steps to prevent other clients from creating a second object with the same identity (for example, by adding the object identity to a vector or map of existing objects).

12-4 The Current Object

The Current Object

Every operation invocation is passed an object of type `Ice::Current`:

```
dictionary<string, string> Context;
enum OperationMode {
    Normal, \Nonmutating, \Idempotent
};

local struct Current {
    ObjectAdapter adapter;
    Connection con;
    Identity id;
    string facet;
    string operation;
    OperationMode mode;
    Context ctx;
    int requestId;
};
```

The `Current` object provides information about the current invocation.



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-4

Notes:

Every operation implementation receives a trailing parameter of type `Ice::Current`. At run time, this parameter provides information about the current request:

- **adapter**
This field specifies the object adapter that was used to dispatch the request. As we saw in Section 12-2, it is useful to have this information available in order to add new entries to the ASM. (Without this information, you would have to store the adapter in a static or global variable, which would be cumbersome.) Note that the `ObjectAdapter` interface provides a `getCommunicator` operation that returns the communicator via which the invocation was dispatched. Again, this is useful because it avoids the need to store the communicator in a global variable.
- **con**
This field provides access to a `Connection` object that contains details of the connection via which the request was invoked. (See the Ice manual for details.)
- **id**
This field provides the object identity of the target object for the request.

- **facet**
This field provides the target facet of the operation. The facet is normally the empty string. (See the Ice manual for details on facets and versioning.)
- **operation**
This field contains the name of the operation that was invoked.
- **mode**
This field specifies whether the corresponding Slice operation is an ordinary operation or is qualified as **idempotent**.
- **ctx**
This field provides a map of name–value pairs that can be sent implicitly with every invocation.
Although you can set and get these name–value pairs from application code, we recommend that you do not use the **ctx** field. (It is intended mainly for the implementation of services and usually not used by applications; see the Ice manual for details.)
- **requestId**
This field provides the request ID that is used by the Ice protocol to associate a request with its response. For twoway operations, this is a positive integer. For oneway invocations, the request ID is zero and, for collocated invocations (that is, invocations where client and server share the same communicator within a single process), the request ID is -1.

12-5 Object Destruction

Object Destruction

To destroy an object, add an operation that instructs the object to commit suicide:

```
exception DirNotEmpty {};
```

```
interface Node {  
    void destroy() throws DirNotEmpty;  
};
```

- The implementation of **destroy** removes the servant from the ASM and destroys whatever resources are held by the servant.
- Clients invoking on the proxy for the destroyed object receive **ObjectNotExistException**.
- As far as the Ice run time is concerned, **destroy** is an ordinary operation without special significance.
- Do not add **destroy** to the factory. If you do, you need to keep track of which factory created what object.



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-5

Notes:

To enable clients to destroy an object, add a **destroy** operation to the object's interface. You can add whatever exceptions are necessary to indicate any application-specific error conditions that prevent the object from being destroyed.

As far as the Ice run time is concerned, your implementation of **destroy** need only remove the corresponding servant's entry from the ASM. This is known as *servant deactivation*. Clients that use the object's proxy after **destroy** has been called receive an **ObjectNotExistException**. However, it is likely that your implementation of **destroy** needs to do other application-specific cleanup, such as closing file descriptors or deleting a database record.

In general, you should add **destroy** to the object itself, instead of adding a **destroy** operation to the object's factory. Adding the operation to the factory forces clients to remember which factory created each object. For an application with several factories and object types, this rapidly becomes unwieldy.

12-6 Implementing Object Destruction

Implementing Object Destruction

The object adapter provides a **remove** operation that removes an entry from the ASM:

```
local interface ObjectAdapter {
    Object remove(Identity id);
    // ...
};
```

remove breaks the link between the object identity and the servant, effectively destroying the Ice object.

- The operation returns the servant that was removed.
- Calling **remove** on an object identity that is not in the ASM raises **NotRegisteredException**.
- If the server code does not hold a reference to the servant elsewhere, the servant becomes eligible for garbage collection as soon as the last executing operation leaves the servant.



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-6

Notes:

Object destruction involves breaking the link between an object's identity and its servant, by calling **remove** on the object adapter. As soon as the ASM entry for a servant is removed, any incoming invocations for the object are rejected with an **ObjectNotExistException**.

Typically, your server code will have only one reference to each servant, namely the reference that is part of the ASM entry. This means that, once you have called **remove**, the servant becomes eligible for garbage collection as soon as the last executing operation on the servant completes. (This is particularly important for threaded applications, in which several concurrent invocations may be active in the same servant.)

To implement a **destroy** operation, you must call **remove** on the object adapter. In addition, you must clean up whatever application-specific resources are held by the servant. This might involve closing file descriptors, deleting a database entry, or any other actions that are appropriate for your application.

Here is one way to implement a **destroy** operation:

```
public void
destroy(Ice.Current c)
    throws DirNotEmpty
```

```
{  
    c.adapter.remove(c.id);  
  
    // Servant is now inaccessible to clients.  
  
    // Remove any servant-specific state here...  
}
```

12-7 Object Destruction and Thread Safety

Object Destruction and Thread Safety

You must avoid a race condition if **destroy** can be called concurrently:

```
public synchronized void
destroy(Ice.Current c)
{
    if(!_destroyed)
        throw new Ice.ObjectNotExistException();
    // Remove any servant-specific state here...
    c.adapter.remove(c.id);
    _destroyed = true;
}

public synchronized void
write(String[] text, Ice.Current c) throws IOException
{
    if(!_destroyed)
        throw new Ice.ObjectNotExistException();
    // ...
}
```



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-7

Notes:

If operations can be called concurrently, you must ensure that all but the first call to **destroy** on the same object raise **ObjectNotExistException**. Moreover, all other operations must also lock the servant and check the **_destroyed** flag. To see why, assume for the moment that we do not have a **_destroyed** flag and that two clients concurrently call **destroy** and **write**. The following sequence of events can occur:

1. Thread A calls **write** and is suspended immediately on entry to the operation, *before* it can acquire the lock.
2. Thread B calls **destroy**. It finds the servant unlocked, locks it, destroys the servant state, and unlocks the servant again.
3. Thread A resumes and acquires the lock. However, **destroy** has already finished and the servant may no longer be in a usable state. For example, **destroy** could have closed network or database connections, deallocated memory, or otherwise changed the state of the servant such that **write** would fail or do something nonsensical.

To avoid this problem, the generic approach is:

- Add a boolean **_destroyed** flag to the servant.

- Make operations synchronized and, on entry to the operation, test the `_destroyed` flag. If the flag is set, throw `ObjectNotExistException`.
- This effectively ensures that, once `destroy` has been called by a client, no other operations can enter the servant. In addition, those operations that have already entered the servant, but have not yet acquired the lock, immediately terminate with `ObjectNotExistException`.

12-8 When to Remove Servant State?

When to Remove Servant State?

Avoid removing servant state in the servant's finalizer:

- **destroy** often must perform clean-up that can fail, such as closing network connections or flushing files.

If you delay physical removal of servant resources until the finalizer runs and something goes wrong, you end up with inconsistent state: **destroy** has completed successfully, but physical servant state is still there!

- If anything goes wrong in the finalizer, the finalizer cannot throw exceptions. (The best it can do is log the error.)



Notes:

You might be tempted to implement **destroy** as follows:

```
public void
destroy(Ice.Current c)
    throws ObjectNotExistException
{
    try
    {
        c.adapter.remove(c.id);
    }
    catch(Ice.NotRegisteredException)
    {
        throw new ObjectNotExistException();
    }
    _destroyed = true;
}
protected void
```

```
finalize() throws Throwable
{
    try
    {
        // Flush files, close transactions, etc...
        if(_destroyed)
        {
            // Remove persistent state, such as database records..
        }
    }
    catch(Exception ex)
    {
        // Log error here because throwing will do nothing.
        // Probably best to abort at this point...
    }
}
```

The idea here is to take advantage of the finalizer which (might) run eventually.

This works fine, as long as nothing goes wrong. However, if the clean-up actions do something that might fail, we end up with a big problem.

- **destroy** has already completed successfully so, as far as the client is concerned, the Ice object was destroyed; however, when the finalizer runs, we find that it is not possible to remove all the state of the object (for example, because a remote database has become inaccessible). Now the system is in an inconsistent state: as far as the client is concerned, the object was destroyed but, physically, the object's state still exists. Depending on exactly how your application works, this may be benign or disastrous.
- The finalizer has no way to report the failure, other than to log an error.

12-9 Object Identity and Uniqueness

Object Identity and Uniqueness

The Ice object model assumes that Ice objects have globally-unique identities.

- If you use UUIDs as object identities, this is guaranteed to be the case.
- If you use application-specific data as object identities, this is not guaranteed—the application must enforce sufficient uniqueness.

Technically, object identities must be unique per object adapter.

Object identity is embedded in the proxy for an object and sent over the wire with each invocation.

If object identities are globally unique, `ObjectNotExistException` is reliable:

- Once a client receives `ObjectNotExistException` from an object, all future attempts to contact the object will either fail, or also raise `ObjectNotExistException`.



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-9

Notes:

Each Ice object has an identity. The identity of an object is an arbitrary string. You can use UUIDs as object identities (for example, by calling `ObjectAdapter::addWithUUID`), or you can use application-specific identities (usually, by taking part of the object state and using it as the identity; for example, for a `Person` object, a social security number would be an obvious choice).

The uniqueness of object identity has implications with respect to object life cycle. In particular, with application-assigned identities, the uniqueness of the identities is controlled by the application. Minimally, object identities must be unique per object adapter, because the ASM uses the identity as the key to locate a servant: you cannot have two servants with the same object identity in the same ASM.

The object identity of an Ice object is stored in the proxy for the object; when a client invokes an operation, the identity is sent over the wire to the server as part of the request; the server uses the identity to locate the correct servant for the invocation.

If object identities are globally unique (that is, no two Ice objects ever use the same identity for all time), `ObjectNotExistException` is a reliable indication of object death: once destroyed, the object stays destroyed forever, because its identity is never reused.

12-10 Object Identity and Uniqueness (cont. 1)

Object Identity and Uniqueness (1)

```
interface File {
    void destroy();
    // ...
};
```

```
interface FileFactory {
    File* create(string pathname);
};
```

Assume that the path name is used as the object identity. A client can now do:

```
FileFactoryPrx ff = ...;
FilePrx f = ff.create("/fred");
// Write to new file...
// Pass f to some other process...
```

```
// Later:
f.destroy();
f = ff.create("/fred");
// Write to new file...
```



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-10

Notes:

If application-specific data is used as object identity, it can be difficult to enforce uniqueness of identities. As shown in the above example, the following sequence of events is possible:

1. A client creates a file with the identity **/fred** and initializes the file contents.
2. The client passes the proxy for the file to some other process.
3. Some time later, the same (or a different) client destroys the file.
4. Eventually, some client creates another file with the identity **/fred**.

As far as the Ice run time is concerned, this is perfectly legal. However, re-using an object identity in this manner can potentially be confusing:

- Normally, **ObjectNotExistException** is expected to be a death certificate: once one invocation raises **ObjectNotExistException**, all future invocations will also raise **ObjectNotExistException** (or fail to reach the server entirely). However, if object identities are reused, after having raised **ObjectNotExistException**, a future invocation via the same proxy can work again.
- If clients store proxies away (for example, in a database), a future invocation can end up in an entirely different and unexpected object, with possibly surprising results.

12-11 Object Identity and Uniqueness (cont. 2)

Object Identity and Uniqueness (2)

```
FileFactoryPrx ff = ...;
FilePrx f = ff.create("/fred");
```

```
// Pass proxy to some other process...
```

```
// Later...
```

```
f.destroy();
```

```
// Still later...
```

```
// Use same identity for different type of object:
```

```
ThingFactoryPrx tf = ...;
ThingPrx t = tf.create("/fred");
```

If a client invokes on the **File** proxy after the object is reincarnated as a **Thing**, it may get an **OperationNotExistException**, **MarshalException**, or even undefined behavior!



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-11

Notes:

The above example is more pathological with respect to re-use of object identities:

1. A client creates a file with the identity **/fred** and initializes the file contents.
2. The client passes the proxy for the file to some other process.
3. Some time later, the same (or a different) client destroys the file.
4. Still later, the identity **/fred** is used for a different type of object, **Thing**.
5. A client that still holds a proxy to the original **File** object and invokes on operation on the file will be confronted with rather surprising behavior:
 - Most likely, the client will receive an **OperationNotExistException**, assuming that the operation being invoked has a name that is valid for a **File**, but does not exist for a **Thing**.
 - If **File** and **Thing** happen to use common operation names, but use different parameters for these operations, the client will most likely receive a **MarshalException** (if the parameters for the **File** operation do not decode correctly as the parameter types expected by the **Thing** operation).

However, it is possible that the parameters happen to be similar enough to decode successfully, in which case the client does not get an exception, but has invoked a completely unrelated operation on a different object with essentially random parameter values!

The latter case can be particularly surprising if both **File** and **Thing** support a **destroy** operation. In that case, the client will believe that it has successfully destroyed the file (even though the file was destroyed already) and, to make things worse, has destroyed a completely unrelated object!

12-12 Uniqueness Recommendations

Uniqueness Recommendations

Consider using UUIDs as object identity. UUIDs are convenient because they make name clashes impossible.

If you use application-assigned object identity, pay attention to uniqueness:

- Ideally, do not ever re-use an identity.
- If you re-use identities, write your application to cope with this:
 - Avoid storing proxies in clients beyond their “use-by date.”
 - Do not build semantics into your application that expect **ObjectNotExistException** to be a definitive death certificate.
 - Use separate namespaces for object identities for different object types (for single object adapters), or
 - Use different object adapters for different types of objects.
- If you want to use IceGrid's well-known objects, you *must* use an identity that is unique within the IceGrid domain.



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-12

Notes:

As far as the Ice run time is concerned, there is no problem with re-using identities¹: from Ice's perspective, at any given time, an object identity either maps to a servant or it does not. If a servant can be found, a request can be dispatched successfully or not, depending on whether the operation name exists and the parameter values can be successfully unmarshaled and the results successfully marshaled back to the client. However, for the application, as we saw in the preceding example, reusing identities can potentially cause problems.

We suggest that you follow the above recommendations. If you decide to use application-defined identities, and there is a potential for an identity to be reused, you simply must write the application to cope with objects that get reincarnated (and somehow make sense of that).

However, in general, avoid using the same namespace for identities on the same object adapter if you re-use identities; the potential for sending invocations to the wrong type of object makes this a bad idea. We recommend using a unique prefix for the identities of objects of different type to avoid the potential name clash.

¹ The Ice run time never tracks the existence of Ice objects in order to remain stateless.

12-13 Dealing with Stale Objects

Dealing with Stale Objects

Consider stateful client–server interactions, such as for an online shop:

- The client creates a shopping cart object via a factory.
- Purchases are added to the cart by invoking operations on the cart.
- When the client is finished, and presses the “Buy” button, the order is processed and the cart is destroyed.

What happens if the client never finishes the purchasing process or crashes?

The server holds onto resources on behalf of the client so, unless the server does something in this case, it will eventually run out of resources.



Notes:

A common problem for distributed applications with life cycle operations is the question of how to guarantee that the server will reclaim resources. In the above example, the problem is caused by the client neglecting to call `destroy` (or `placeOrder`) as expected. The server allocates resources, such as memory, to the client. The expectation is that the client will eventually either purchase the items in the shopping cart or cancel the order (either way, thereby destroying the cart) so the server can reclaim its resources. However, if the client crashes or simply forgets to complete the purchasing process, the server is left sitting on the allocated memory with no way of knowing that the client has crashed or has forgotten to complete the process. (A dead client is indistinguishable from a slow client, as far as the server is concerned.)

Typically, the situation is made more complex by the fact that several objects may be involved: the server may, for example, create a number of objects on behalf of the client, all of which must eventually be destroyed.

12-14 Dealing with Stale Objects (cont. 1)

Dealing with Stale Objects (1)

Basic approach for cleaning up stale objects:

- Instead of creating objects directly, each client creates a single session object.
- The session object is the object factory that allows the client to create all the objects it needs.
- The session keeps track of which objects were created.
- The session offers a **refresh** operation. The client is expected to call refresh every n seconds.
- If the client fails to call **refresh** for more than n seconds, the server destroys the session object, and all objects created by that session.

This approach guarantees:

- resources will be reclaimed if a client crashes
- resources are not reclaimed prematurely (while still being used)



Notes:

There are various ways to deal with the stale object problem. For example, the server can arbitrarily cap the number of live sessions and use an evictor (see Chapter 18). However, the problem with this approach is that, once the limit is reached, the server must destroy an existing session for each new session that is created. In turn, this risks destroying a session that may have been around for quite a long time, but is still being used.

Another approach is to use a timeout: if an object has not been used for n seconds, the server destroys it. The drawback is the same one as for the previous approach: the server might destroy an object that is still being used (albeit rarely).

You might also consider implementing a distributed reference counting mechanism. If so, matters are likely to get worse than better: the reference count must reliably be updated if a client crashes, which is non-trivial. And distributed reference counting does not scale (as evidenced by DCOM). We therefore strongly discourage this approach.

The above slide presents a pragmatic approach to solving the problem. It guarantees not to destroy resources prematurely, and it guarantees that resources will be reclaimed if a client crashes.

Note that the approach does not deal with the case of a client that has a bug and forgets to either complete a purchase or to empty the shopping cart: as long as the client process stays alive, its objects will not be destroyed by the server. However, the above approach effectively deals with the most common problem: how to recover in the face of network failure or crashed clients.

In addition, the approach is remarkably non-intrusive to existing object implementations. This means that you can back-patch it into existing implementations with little effort.

12-15 Dealing with Stale Objects (cont. 2)

Dealing with Stale Objects (2)

```

interface SomeObject {
    // Lots of operations here...

    void destroy();
};

interface Session { // One session per client
    SomeObject* create(/* params */);
    idempotent string getName();
    void refresh();
    idempotent void destroy();
};

interface SessionFactory { // Singleton
    Session* create(string name);
};

```



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-15

Notes:

12-15-1 Interface Definitions

The above slide shows the Slice definitions to support reaping of stale objects.

SomeObject is the interface for the objects that clients create as part of their interactions with the server. (There is no need to limit this design to a single interface—you can just as easily support several interfaces.)

The **Session** interface acts as the factory for application objects. (Again, you could have several factory operations for different types of objects in this interface.) The session has a name, so it can be identified (for example, in log messages); that name is returned by the **getName** operation.

The **refresh** operation is expected to periodically be called by the client to keep its session alive. If the client neglects to call **refresh** before the session timeout expires, the session destroys itself and all objects that were created by its factory operations.

In addition, the session provides a **destroy** operation, so clients can explicitly destroy a session (instead of just letting the timeout expire).

The **SessionFactory** interface allows clients to create a named session. (The session factory is a singleton object—all clients use the same session factory to create their sessions.)

12-16 Dealing with Stale Objects (cont. 3)

Dealing with Stale Objects (3)

The reaper thread:

- maintains a list of existing sessions
- provides an **add** operation so sessions can be added
- runs an infinite loop:
 - sleep for n seconds
 - get the current time
 - for each existing session, if the session's timestamp is older than n seconds, call **destroy** on the session and remove the session from the list
 - if a call to **destroy** raises **ObjectNotExistException**, remove the session from the list



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-16

Notes:

12-16-1 Implementing the Reaper Thread

The reaper thread is run in the server. It is responsible for identifying and destroying stale sessions. Here is the class definition:

```
class ReaperThread extends Thread
{
    public void
    run();

    public synchronized void
    terminate();

    public synchronized void
    add(SessionPrx proxy, SessionI session);

    private boolean _terminated = false;
    private long _timeout = 60000;    // 60 seconds
}
```

```
        private java.util.HashMap<SessionPrx, SessionI> _sessions =
            new java.util.HashMap<SessionPrx, SessionI>();
    }
```

The **add** operation adds a proxy and servant to the **_sessions** map. The map contains both the proxy and the servant reference so we can invoke Slice operations via the proxy, as well as invoke other member functions via the reference.

```
public synchronized void
add(SessionPrx proxy, SessionI session)
{
    _sessions.put(proxy, session);
}
```

The **terminate** method allows us to ask the reaper thread to shut down. (We need to do this when the server shuts down.) Note that this also destroys all existing sessions.

```
public synchronized void terminate()
{
    try
    {
        for(SessionPrx sp : _sessions.keySet())
        {
            try
            {
                sp.destroy();
            }
            catch(Ice.ObjectNotExistException ex)
            {
                // Ignore.
            }
        }
    }
    catch(Ice.ConnectionRefusedException ex)
    {
    }
    _sessions.clear();
    _terminated = true;
    notify();
}
```

When the server wants to shut down, it calls **terminate** from its main thread. **terminate** iterates over the sessions and calls **destroy** on each session. Note that the code ignores **ObjectNotExistException** when it tries to destroy a session. This is because clients may have explicitly called **destroy** on a session since the reaper thread last ran, so it is possible for a session to be destroyed already, but to still be in the reaper's list of sessions.

Also note the code stops trying to destroy sessions if it encounters a `ConnectionRefusedException`. This exception can be raised if the server calls `terminate` after shutting down the communicator. In that case, the server is on the way out so there is no point in continuing to destroy sessions.

Finally, `terminate` sets the `_terminated` flag and then calls `notify` to wake up the reaper thread, and then waits for the reaper thread to exit.

Here is the `run` method, which implements the reaper thread:

```
public void
run()
{
    synchronized(this)
    {
        while(!_terminated)
        {
            try
            {
                wait(_timeout);
            }
            catch(InterruptedException ex)
            {
            }
            if(!_terminated)
            {
                Iterator< Map.Entry<SessionPrx, SessionI> > i =
                    _sessions.entrySet().iterator();
                while(i.hasNext())
                {
                    Map.Entry<SessionPrx, SessionI> e = i.next();
                    SessionPrx sp = e.getKey();
                    SessionI s = e.getValue();
                    try
                    {
                        if(System.currentTimeMillis() -s.timestamp() >
                            _timeout)
                        {
                            sp.destroy();
                            i.remove();
                        }
                    }
                    catch(Ice.ObjectNotExistException ex)
                    {
                        i.remove();
                    }
                }
            }
        }
    }
}
```

```
    }  
  }  
}
```

The thread sits in a loop that runs as long as `_terminate` is false. The first action inside the loop is to lock the monitor and enter a timed wait. This releases the lock and puts the reaper thread to sleep. The reaper thread returns from `wait` when one of two events occurs:

- The server's main thread has called `terminate`, which sets `_terminate` to true and calls `notify`.
In this case, when the reaper thread wakes up, it finds that `_terminate` is now true and exits.
- The timeout of the timed wait has expired.
In this case, when the reaper thread wakes up, it finds that `_terminate` is still false and executes the body of the loop.

In the loop body, the reaper thread iterates over the entries in the `_sessions` map. It gets the current system time and compares it to the timestamp of the session. If the time that has elapsed since the session was last refreshed is greater than the timeout, the reaper thread calls `destroy` on the session. (The reaper thread reacquires the current time for each entry in the map because destroying a session could potentially take a while.) Note that, as for terminating, the reaper thread ignores `ObjectNotExistException` because a client may have explicitly called `destroy` on a session that is still in the reaper's map. If the reaper destroys a session (or finds that the session was destroyed previously), it also removes it from the `_sessions` map.

12-17 Dealing with Stale Objects (cont. 4)

Dealing with Stale Objects (4)

```
class SessionFactoryI extends _SessionFactoryDisp
{
    public SessionFactoryI (ReapThread r)
    {
        _reaper = r;
    }
    public SessionPrx create(String name, Ice.Current c)
    {
        SessionI session = new SessionI ();
        SessionPrx proxy = SessionPrxHelper.uncheckedCast(
            c.adapter.addWithUUID(session));
        _reaper.add(proxy, session);
        return proxy;
    }
    private ReapThread _reaper;
}
```



Notes:

12-17-1 Implementing the Session Factory

The session factory implementation is very simple. The constructor initializes the `_reaper` member with the reaper thread, and the `create` method instantiates a new session and adds it to the reaper's list of sessions by calling `add` on the reaper.

12-18 Dealing with Stale Objects (cont. 5)

Dealing with Stale Objects (5)

```

class Session extends _SessionDisp
{
    public synchronized SomeObjectPrx
    create(Ice.Current c);

    public synchronized void
    destroy(Ice.Current c);

    public void
    refresh(Ice.Current c);

    public long
    timestamp();

    private long _timestamp = System.currentTimeMillis();
    private boolean _destroyed = false;
    private java.util.LinkedList<SomeObjectPrx> _objs =
        new java.util.LinkedList<SomeObjectPrx>();
}

```



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-18

Notes:

12-18-1 Implementing the Session

The session (which acts as the factory for `SomeObject` objects) provides the (application-specific) create factory operation. It creates a new servant and adds it to the ASM as usual, and also adds the proxy for the servant to the list of servants it has created:

```

public synchronized SomeObjectPrx
create(Ice.Current c)
{
    if(_destroyed)
    {
        throw Ice.ObjectNotExistException();
    }
    SomeObjectI o = new SomeObjectI();
    SomeObjectPrx proxy = SomeObjectPrxHelper.uncheckedCast(
        c.adapter.addWithUUID(o));
    _objs.add(proxy);
    return proxy;
}

```

```
}
```

The `destroy` operation destroys all objects created by this session.

```
public synchronized void
destroy(Ice.Current c)
{
    if(!_destroyed)
    {
        throw new Ice.ObjectNotExistException();
    }
    _destroyed = true;

    try
    {
        c.adapter.remove(c.id);
        for(SomeObjectPrx p : _objs)
        {
            c.adapter.remove(p.ice_getIdentity());
        }
    }
    catch(Ice.ObjectAdapterDeactivatedException ex)
    {
        // This method is called on shutdown of the server,
        // in which case this exception is expected.
    }
    _objs.clear();
}
```

The only noteworthy point here is that the implementation ignores `ObjectAdapterDeactivatedException`. This prevents exceptions from propagating out of the operation when a session is destroyed as part of server shut-down.

The `refresh` operation is called by the client to keep the session alive and stores the current time in the session's `_timestamp` member:

```
public synchronized void
refresh(Ice.Current c)
{
    if(!_destroyed)
    {
        throw new Ice.ObjectNotExistException();
    }
    _timestamp = System.currentTimeMillis();
}
```

12-19 Dealing with Stale Objects (cont. 6)

Dealing with Stale Objects (6)

The server's main program:

- instantiates an object adapter
- creates the reaper thread and starts it
- creates the session factory and adds it to the ASM
- activates the object adapter
- waits for shut-down

Once shut-down is complete, the server:

- calls `terminate` on the reaper thread
- joins with the reaper thread



Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

12-19

Notes:

12-19-1 Server main Implementation

The server's main implementation is very simple. After creating and starting the reaper thread, the server creates the session factory and then waits for shut-down. Once `waitForShutdown` returns, the server calls `terminate` on the reaper thread and waits for the reaper thread to finish by joining with it:

```
Ice.ObjectAdapter adapter =
    communicator().createObjectAdapter("SessionFactory");
ReapThread reaper = new ReapThread();
reaper.start();
SessionFactoryI sf = new SessionFactoryI(reaper);
SessionFactoryPrx sfproxy = SessionFactoryPrxHelper.uncheckedCast(
    adapter.add(sf, communicator().stringToIdentity("SessionFactory")));
// ...

adapter.activate();
// ...
```

```
communicator().waitForShutdown();

reaper.terminate();
while(true)
{
    try
    {
        reaper.join();
        break;
    }
    catch(InterruptedException ex)
    {
    }
}
```

12-20 Dealing with Stale Objects (cont. 7)

Dealing with Stale Objects (7)

The client must call `refresh` every n seconds to keep the session alive.

Instead of arbitrarily sprinkling calls to `refresh` through the code, in the hope that they get executed often enough, run a background thread:

- The `refresh` thread sits in a loop and calls `refresh` periodically. To be safe, make the refresh interval a little bit shorter than the server's reap interval.
- The refresh thread provides a `terminate` method so the client's main thread can join with it when the time comes to shut down.



Notes:

12-20-1 Client-Side Implementation

The client must keep the session alive by calling `refresh` periodically. You could sprinkle your code with calls to `refresh` in the hope that they will be executed often enough. But that is not only messy, it also fails if the client's main thread blocks for some reason. A much better approach is to run a separate thread that takes care of calling `refresh` periodically. To be safe, the refresh interval for the client should be somewhat shorter than the server's reap interval. For example, if the reap interval is one minute, the client could call `refresh` every 50 seconds to account for the occasional network delay.

The implementation of the refresh thread is very simple and shown in full here.

```
class SessionRefreshThread extends Thread
{
    SessionRefreshThread(long timeout, SessionPrx session)
    {
        _timeout = timeout;
        _session = session;
    }
}
```

```
    }

    public void
    run()
    {
        synchronized(this)
        {
            while(!_terminated)
            {
                try
                {
                    wait(_timeout);
                }
                catch(InterruptedException ex)
                {
                }
                if(!_terminated)
                {
                    try
                    {
                        _session.refresh();
                    }
                    catch(Ice.LocalException ex)
                    {
                        _terminated = true;
                    }
                }
            }
        }
    }

    public synchronized void
    terminate()
    {
        _terminated = true;
        notify();
    }

    private boolean _terminated = false;
    private long _timeout;
    private SessionPrx _session;
}
```

As for the server-side reaper thread, the refresh thread sleeps for the timeout interval and calls `refresh` each time the timer expires. The `terminate` method allows the client's main thread to terminate the refresh thread.

The only changes to the normal client code are to create a proxy to the session factory, create a session, and to pass the session to the refresh thread's constructor. Before going about its normal business, the client starts the refresh thread and, when the client is ready to shut down, it calls `terminate` on the refresh thread and joins with it:

```
SessionFactoryPrx factory = ...;

SessionPrx session = factory.create(name);

SessionRefreshThread refresh =
    new SessionRefreshThread(50000, session);
refresh.start();
try
{
    // Main client logic here...
    //
    // The refresh thread must be terminated before destroy
    // is called, otherwise it might get
    // ObjectNotExistException. refresh is set to null so
    // that if session.destroy() raises an exception the
    // thread will not be re-terminated and re-joined.
    //
    refresh.terminate();
    while(true)
    {
        try
        {
            refresh.join();
            break;
        }
        catch(InterruptedException ex)
        {
        }
    }

    refresh = null;
    session.destroy();
}
catch(Exception ex)
{
    //
    // The refresh thread must be terminated in the event of
    // a failure.
    //
    if(refresh != null)
```

```
{
    refresh.terminate();
    while(true)
    {
        try
        {
            refresh.join();
            break;
        }
        catch(InterruptedException ex)
        {
        }
    }
}
```

13 Assignment 6 Object Life Cycle

13 Assignment 6: Object Life Cycle

13-1 Exercise Overview

Exercise Overview

In this exercise, you will:

- add life cycle operations to the file system server.

By the completion of this exercise, you will have gained experience in how to implement thread-safe life cycle operations.

Assignment 6 Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.13-1

Notes:

In this exercise, you will add life cycle operations to the file system server.

13-1-1 Exercise Objectives

By the end of this exercise, you will have gained experience in how to implement thread-safe life cycle operations.

13-2 Life Cycle

Life Cycle

- In this exercise, you will add life cycle operations to the file system server.
- The server is the thread-safe server you developed in Assignment 5, so your implementation will need to be thread-safe.



Assignment 6 Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

13-2

Notes:

In this exercise, you will add life cycle operations to the file system server. The server is the thread-safe server you developed in Assignment 5, so your implementation will need to be thread-safe.

13-3 What You Need to Do

What You Need to Do

1. Examine the code in `Server.java`.
2. Look at the implementations of `makeRootDir` and the `DirectoryI` constructors in `Filesystem/DirectoryI`.
3. Implement the `createDir` method.
4. Implement the `createFile` method.
5. Use the test client that is contained in `Client.java` to test your create operations.
6. Implement the `DirectoryI.destroy` method.
7. Implement the `FileI.destroy` method.
8. Edit `Client.java` and enable the commented-out section marked `PART_2`.



Assignment 6 Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

13-3

Notes:

-
-
-
-
1. Examine the code in `Server.java`. Note that the server now uses a static function `DirectoryI.makeRootDir` to create the root directory.
 2. Look at the implementations of `makeRootDir` and the `DirectoryI` constructors in `Filesystem/DirectoryI`. Make sure you understand how the `DirectoryI` constructor works—you will need to use it in the implementations of `createDir` and `createFile`.
 3. `DirectoryI` provides three helper function that you can use to implement the life cycle operations:
 - `checkNameInUse`
This function checks if a node with the same name exists already in a directory. If so, it throws a `NameInUse` exception. Use this function in your create implementations to throw an exception if a client attempts to create a node with the same name as an existing node.
 - `addChild`
As before, this function can be called by the create implementations in order to add a newly-created node to its parent directory.
 - `removeChild`

This function performs the inverse operation, namely, it removes the specified node from the parent's list of nodes. This function is for use by the **destroy** implementations.

Implement the **createDir** method.

4. Implement the **createFile** method.
5. **Client.java** contains a test client that you can use to test your create operations. Look through the client code (up to the point marked **PART_2**) to see what it does. Compile the client and run it with your server. If you have implemented the create operations correctly, the client will list the contents of the file system without error.
6. Implement the **DirectoryI.destroy** method. Make sure that you prevent attempts to destroy a directory that is not empty, and that you prevent attempts to destroy the root directory. (Throw a **DirNotEmpty** exception in either case.)

Pay attention to the thread safety of your implementation. In particular, consider what happens if a client performs a **list** operation on a directory at the same time as another client destroys a node in the same directory. Also make sure that any operations that have been dispatched, but have not yet entered the body of the operation implementation, correctly raise **ObjectNotExistException** if a client concurrently destroys the operation's node.

7. Implement the **FileI.destroy** method.
8. Edit **Client.java** and enable the commented-out section marked **PART_2**. Compile the client and run it against your server. If you have implemented your **destroy** operations correctly, the client will complete without error.

13-4 Thread Safety Modifications

Thread Safety Modifications

- To prevent a race condition, we need to test, on entry to every operation, whether the object has been previously destroyed. Rather than repeat the same code in every operation, we can bundle the test into a helper function.



Assignment 6 Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

13-4

Notes:

To prevent the race condition shown in Section 12-7, we need to test, on entry to every operation, whether the object has been previously destroyed. Rather than repeat the same code in every operation, we can bundle the test into a helper function:

```
class DirectoryI extends _DirectoryDisp
{
    // ...
    private void
    checkDestroyed()
    {
        if(_destroyed)
        {
            throw new Ice.ObjectNotExistException();
        }
    }
    private boolean _destroyed = false;
    // ...
}
```

The helper function tests the `_destroyed` member and throws `ObjectNotExistException` if it is set. Note that this method is not synchronized because it is called only from methods that themselves are synchronized.

On entry to all other operations, we call `checkDestroyed`:

```
public final class DirectoryI extends _DirectoryDisp
{
    // ...
    public synchronized String
    name(Ice.Current c)
    {
        checkDestroyed();

        // ...
    }

    public synchronized FilePrx
    createFile(String name, Ice.Current c)
    throws NameInUse
    {
        checkDestroyed();

        // ...
    }

    public synchronized DirectoryPrx
    createDir(String name, Ice.Current c)
    throws NameInUse
    {
        checkDestroyed();

        // ...
    }

    public void
    destroy(Ice.Current c)
    throws DirNotEmpty
    {
        synchronized(this)
        {
            checkDestroyed();
            // ...
        }
        // ...
    }

    public synchronized NodeDetails[]
    list(Ice.Current c)
    {
        checkDestroyed();

        // ...
    }

    // ...
}
```

If an object is destroyed while another operation on the same object has already been dispatched, but the operation has not entered the operation body yet, the client correctly receives an **ObjectNotExistException** (rather than allowing the operation to run on a destroyed Ice object which, depending on the implementation, may cause problems).

Note that the **FileI** implementation uses the same strategy for its operations (not shown).

13-5 createDir Implementation

createDir Implementation

- `createDir` calls `checkDestroyed` in case this directory has been destroyed before the operation body started to execute and then calls `checkNameInUse` to make sure that clients cannot create two directories with the same name.



Notes:

`createDir` calls `checkDestroyed` in case this directory has been destroyed before the operation body started to execute and then calls `checkNameInUse` to make sure that clients cannot create two directories with the same name. Note that this code is free from race conditions because `createDir` is a synchronized method.

The code then instantiates a new `DirectoryI` and creates a proxy for the new object.

Most of the work is done by the `DirectoryI` constructor, which adds the new servant to the `_contents` member and the ASM:

```
public synchronized DirectoryPrx
createDir(String name, Ice.Current c)
    throws NameInUse
{
    checkDestroyed();

    checkNameInUse(name);
    DirectoryI d = new DirectoryI(name, this, c.adapter);
    return DirectoryPrxHelper.uncheckedCast(c.adapter.createProxy(
        c.adapter.getCommunicator().stringToIdentity(d._myID)));
}

public
DirectoryI(String name, DirectoryI parent, Ice.ObjectAdapter adapter)
```

```
{
    assert(parent != null);

    _name = name;
    _parent = parent;
    _myID = java.util.UUID.randomUUID().toString();
    parent.addChild(_myID, this);
    adapter.add(this, adapter.getCommunicator().stringToIdentity(_myID));
}
```

13-6 createFile Implementation

createFile Implementation

- The `createFile` implementation is analogous to `createDir`.
- The `FileI` constructor does much of the work



Assignment 6 Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

13-6

Notes:

The `createFile` implementation is analogous to `createDir`. Again, the `FileI` constructor does much of the work:

```
public class DirectoryI extends _DirectoryDisp
{
    // ...

    public synchronized FilePrx
    createFile(String name, Ice.Current c)
        throws NameInUse
    {
        checkDestroyed();

        checkNameInUse(name);
        FileI f = new FileI(name, this, c.adapter);
        return FilePrxHelper.uncheckedCast(c.adapter.createProxy(
            c.adapter.getCommunicator().stringToIdentity(f._myID)));
    }
    // ...
}

public class FileI extends _FileDisp
{
    public
```

```
FileI(String name, DirectoryI parent, Ice.ObjectAdapter adapter)
{
    assert(parent != null);

    _name = name;
    _myID = java.util.UUID.randomUUID().toString();
    _parent = parent;
    parent.addChild(_myID, this);
    adapter.add(this, adapter.getCommunicator().stringToIdentity(_myID));
}

// ...
}
```

13-7 DirectoryI.destroy Implementation

DirectoryI.destroy Implementation

```
public void
destroy(Ice.Current c)
{
    synchronized(this)
    {
        checkDestroyed();
        _destroyed = true;
    }
    c.adapter.remove(c.id);
    _parent.removeChild(_myID);
}
```



Assignment 6 Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

13-7

Notes:

```
public void
destroy(Ice.Current c)
{
    synchronized(this)
    {
        checkDestroyed();
        _destroyed = true;
    }
    c.adapter.remove(c.id);
    _parent.removeChild(_myID);
}
```

As for the other operations, **destroy** first checks whether the object was previously destroyed, and it prevents attempts to destroy a non-empty directory or the root directory. It then removes the ASM entry for the object and removes the destroyed node from the parent directory's **_contents** map by calling **removeChild**.

Note that **destroy** is not a synchronized method, but uses a synchronized block instead. This is necessary to prevent a potential deadlock with **list.list** is a synchronized method because, while it iterates over the **_contents** map, it must prevent concurrent modification of that map.

However, to assign to the `name` member of each of the returned `NodeDetails` classes, `list` invokes the `name` operation on each child node. In turn, `name` is a synchronized method. If `destroy` would be a synchronized method (instead of using a synchronized block inside the method), the following scenario could arise:

1. Client A calls `list` on a directory, which locks the directory.
2. Client B concurrently calls `destroy` on one of the child directories of the directory being listed by Client A. If `destroy` were a synchronized method, that would lock the child directory, so client A holds a lock on the parent, and client B holds a lock on the child.
3. The `list` implementation, on behalf of client A, invokes the `name` operation on the directory that is locked by client B.
4. The `destroy` operation, on behalf of client B, calls `removeChild` on the directory locked by client A.

At this point, the code deadlocks because each thread holds a lock that is required by the other thread in order to proceed.

The separate synchronized block in `destroy` prevents this deadlock: the `destroy` implementation releases the lock on itself before it attempts to lock the parent directory as part of `removeChild`, which makes the deadlock impossible.

An alternative way of preventing the deadlock would be to make the `_name` member public, in which case `list` could assign the `name` member of each `NodeDetails` class without invoking the name operation, and thereby avoid having to lock a child node.

Either approach is possible. The main point to keep in mind here is that `destroy` and `list` can easily cause circular lock dependencies, which complicates the code. For this reason, a reaping approach is often the better solution.

13-8 FileI.destroy Implementation

FileI.destroy Implementation

The `FileI.destroy` implementation is analogous to `DirectoryI.destroy`:

```
public void
destroy(Ice.Current c)
{
    synchronized(this)
    {
        checkDestroyed();
        _destroyed = true;
    }
    c.adapter.remove(c.id);
    _parent.removeChild(_myID);
}
```



Assignment 6 Object Life Cycle
Copyright © 2005-2010 ZeroC, Inc.

13-8

Notes:

The `FileI.destroy` implementation is analogous to `DirectoryI.destroy`:

```
public void
destroy(Ice.Current c)
{
    synchronized(this)
    {
        checkDestroyed();
        _destroyed = true;
    }

    c.adapter.remove(c.id);
    _parent.removeChild(_myID);
}
```

14 Glacier2

14 Glacier2

14-1 Lesson Overview

Lesson Overview

- Glacier2 is the Ice firewall traversal service. It allows clients and servers to communicate even if they are separated by a firewall.
- By the completion of this lesson, you will understand how Glacier2 works, how to configure it correctly, and how to modify your applications to work with Glacier2.

Glacier2
Copyright © 2005-2010 ZeroC, Inc.14-1

Notes:

Glacier2 is the Ice firewall traversal service. It allows clients and servers to communicate even if they are separated by a firewall.

14-1-1 Lesson Objectives

By the completion of this lesson, you will understand how Glacier2 works, how to configure it correctly, and how to modify your applications to work with Glacier2.

14-2 Running an Ice Server Behind a Firewall

Running an Ice Server Behind a Firewall

If a server is behind a firewall, clients can access the server if:

- The firewall opens an incoming port for clients.
- The firewall port-forwards incoming connections on that port to the real server port.
- The server is configured to advertise the firewall's host name and port in its proxies instead of its own name and port by setting the `<adapter-name>.PublishedEndpoints` property.

Problems of this approach:

- Each server requires a separate hole in the firewall.
- If clients need to connect to the server from the inside network as well as the outside network, either:
 - traffic is routed from the inside network to the firewall and back into the inside network again (inefficient), or
 - the server must publish internal and external addresses in its proxies.



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-2

Notes:

The above slide presents one way to run an Ice server behind a firewall. To allow clients to connect to the server, the firewall must open a port and forward all traffic on that port to the server's machine on the inside network. In addition, the server must be configured to publish the firewall's host name and port in its proxies, instead of its own host name and port. (Ice supports this with the `<adapter-name>.PublishedEndpoints` property. If that property is set, the adapter publishes the specified endpoints in its proxies, instead of the server's actual endpoints.)

A significant problem with this approach is that, if the server only publishes the firewall's endpoints in its proxies, traffic from clients that are inside the network is routed via the firewall: from the inside client to the external interface of the firewall, and from there back into the inside network to the server. (This assumes that the firewall will accept incoming connections from the inside network, which may not be the case.)

An alternative is to have the server publish its own host name and port as well as the firewall's, so the proxies that are created by the server are usable by both external and internal clients. However, we strongly discourage this solution. When a proxy contains multiple endpoints, Ice's default behavior is to select one at random. Unless you take great care in constructing and configuring your proxies, it is quite likely that clients will randomly choose an inappropriate endpoint, resulting in potential delays during connection establishment, arbitrary connection refusals, or routing inefficiencies.

Finally, by far the biggest problem is typically that each Ice server that must be accessible from the outside network requires a separate hole in the firewall. Even if an organization's security policy allows this, it creates an administrative problem because the firewall requires its rules to be updated each time a server is added or removed.

14-3 Glacier2

Glacier2

Glacier2 is the Ice firewall-traversal service. It provides:

- firewall traversal for servers with no change to code or configuration
- firewall traversal for clients with minimal code and configuration changes
- callbacks from servers to clients via a bidirectional connection (with minimal changes)
- authentication via user name and password (among others)
- session management
- secure communication via SSL
- request batching and filtering

Glacier2 requires only a single port to be opened in the firewall to support an arbitrary number of clients and servers.

Alternatively, Glacier2 can also *be* the firewall for Ice servers. (No port forwarding is required in this case.)



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-3

Notes:

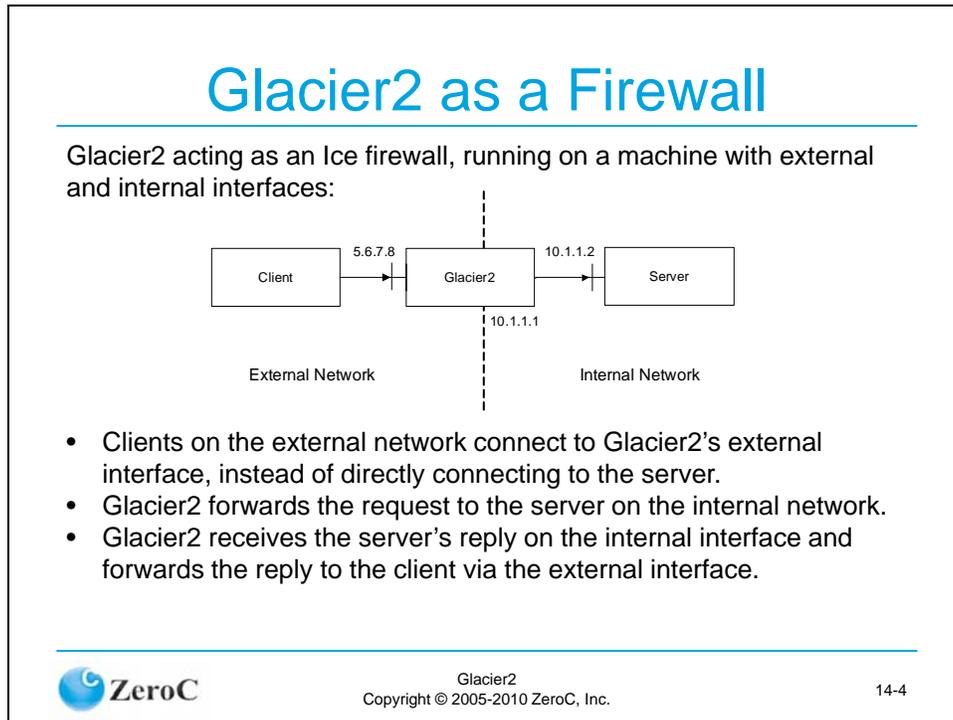
Glacier2 secures traffic from clients on an external, insecure network to servers on an internal, secure network. Glacier2 can be used with minimal disturbance to existing applications: for servers (unless callbacks are used), Glacier2 is effectively invisible; for clients, only minimal code and configuration changes are necessary in order to work with Glacier2.

Glacier2 eliminates the problem of having to open a separate port for each Ice server on the internal network: an arbitrary number of clients and servers can communicate via that single port. If Glacier2 runs behind a corporate firewall, the firewall must be configured to open a single port and forward the traffic on that port to Glacier2. Alternatively, if Glacier2 runs on a machine with two interfaces, one for the external network and one for the internal network, it can act as the firewall for Ice servers, so no port forwarding by the corporate firewall is necessary.

In addition, Glacier2 provides a few other services, such as authentication, session management, request batching and filtering, and secure communications via SSL.

Note that Glacier2 does not work with UDP, only TCP and SSL.

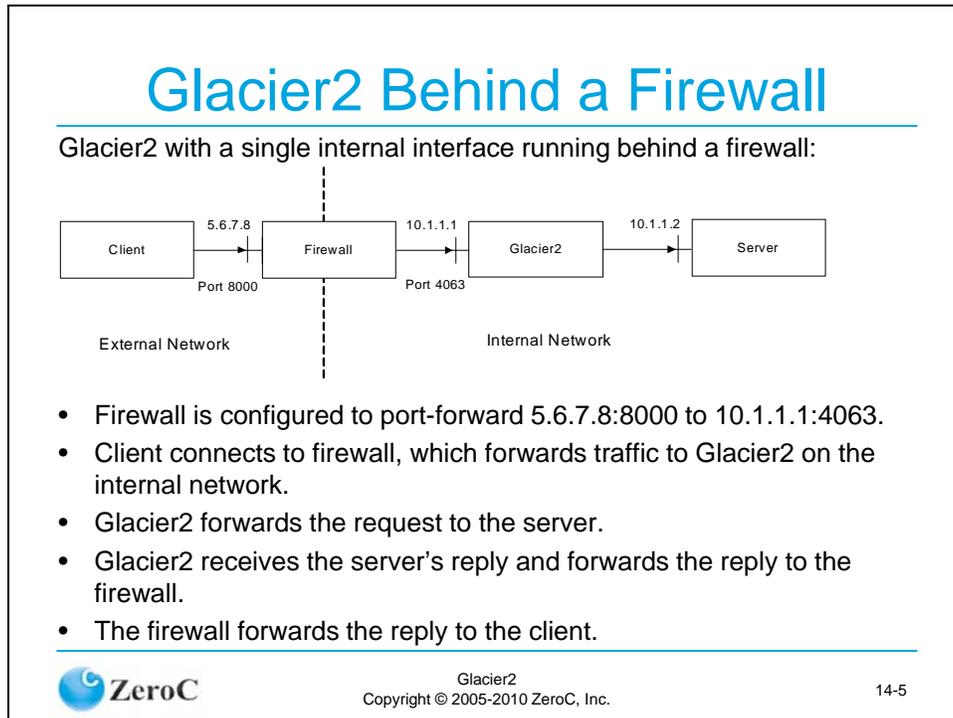
14-4 Glacier2 as a Firewall



Notes:

If Glacier2 runs on a machine with external and internal interfaces, it can act as a firewall for Ice servers. Glacier2 is configured to listen on a specific port of the external interface for incoming client requests and forwards these requests via the internal interface to the server. The server does not know that anything unusual is going on: as far as the server is concerned, Glacier2 *is* the client. When the server's operation completes, it returns the results to Glacier2 via the same connection on which it received the incoming request; in turn, Glacier2 forwards the reply to the client.

14-5 Glacier2 Behind a Firewall



Notes:

If Glacier2 runs on the internal network only and is behind a firewall, the scenario is almost the same as if Glacier2 acts as the router/firewall itself. The only difference is that the firewall must be configured to port-forward incoming requests on a single port to Glacier2's client port.

14-6 Running Glacier2

Running Glacier2

Glacier2's executable is called `glacier2router`.

UNIX daemon options:

- `--daemon`
- `--noconsole`
- `--nochildr`

Use the `iceserviceinstall` utility to configure it as a Windows service.

Glacier2
Copyright © 2005-2010 ZeroC, Inc.14-6

Notes:

Glacier2 is packaged as the `glacier2router` executable. The program has only a few options to allow it to be run as a UNIX daemon, and you can use the `iceserviceinstall` tool to configure it as a Windows service. (See the Ice manual for details.)

The configuration of Glacier2, such as setting endpoints and timeouts, is achieved by setting properties.

14-7 Glacier2 Configuration

Glacier2 Configuration

To configure Glacier2 to be usable by clients, you must minimally set the `Glacier2.Client.Endpoints` property.

It specifies the endpoint at which Glacier2 listens for incoming client requests, for example:

```
Glacier2.Client.Endpoints=tcp -p 4063
```

- If Glacier2 cannot be accessed by hostile clients, TCP is usually the appropriate protocol.
- If Glacier2 should allow access only for specific clients or if you require secure communications, you should specify SSL as the protocol.

If you specify SSL, you must also configure the IceSSL plugin by setting:

```
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=. . .
IceSSL.CertAuthFile=. . .
IceSSL.CertFile=. . .
IceSSL.KeyFile=. . .
```



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-7

Notes:

The only mandatory configuration item for Glacier2 is provided by the `Glacier2.Client.Endpoints` property. It specifies the port number and IP address (or host name) at which Glacier2 listens for client requests. (If the property is not set, `glacier2router` terminates with an error message.)

14-8 Glacier2 Sessions

Glacier2 Sessions

For each client, Glacier2 maintains a session. Clients must obtain a Router proxy and use it to create a session:

```

module Glacier2 {
    exception CannotCreateSessionException {
        string reason;
    };
    exception PermissionDeniedException {
        string reason;
    };
    interface Session {
        void destroy();
    };
    interface Router extends Ice::Router {
        // ...
        Session* createSession(string userId,
                               string password)
            throws PermissionDeniedException,
                CannotCreateSessionException;
    };
};

```



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-8

Notes:

Glacier2 maintains a session on behalf of each client. To use Glacier2, each client must create a session, supplying a user name and password.

The session is created by calling `createSession` on the `Glacier2::Router` interface.

Here is the canonical code for doing this:

```

Ice.RouterPrx defaultRouter = communicator().getDefaultRouter();
if(defaultRouter == null)
{
    System.err.println("No default router configured");
    return;
}
Glacier2.RouterPrx router = Glacier2.RouterPrxHelper.checkedCast(defaultRouter);

String username = ...
String password = ...
try
{
    router.createSession(username, password);
}

```

```
catch(Glacier2.PermissionDeniedException ex)
{
    System.err.println("Permission denied: " + ex.reason);
}
catch(Glacier2.CannotCreateSessionException ex)
{
    System.err.println("Cannot create session: " + ex.reason);
}
```

The communicator is configured with a router. (We will see how to do this shortly.) `getDefaultRouter` returns that configured router.

A router is an object that is interposed into the invocation path of the client: any invocations made by the client are sent to the router instead; the router then forwards the invocation to a specific endpoint, such as the configured Glacier2 endpoint.

Note that the code ignores the return value from `createSession`. This is because, unless you arrange otherwise, `createSession` returns a null proxy: the session that is created in this way is known as an *internal session*, that is, it is a session that is solely maintained by Glacier2 and inaccessible to the client. (We will see how to create external sessions in Section 14-15.)

Once the client has created the session, it can communicate with the server as usual, that is, the preceding code is the *only* code change that is necessary to make an existing client work with Glacier2.

If clients connect to Glacier2 via a public network and you care about security, you should configure Glacier2 and the client to communicate via SSL; if you use TCP, the user name and password are sent over the wire in clear text. Note that you can also use SSL credentials to create a session by calling `createSessionFromSecureConnection`—see the Ice Manual for details.

14-9 Client Configuration

Client Configuration

You must set two properties for the client to use Glacier2:

- `Ice.Default.Router=Glacier2/router: tcp \`
 `-h 5.6.7.8 -p 4063`
- `Ice.ACM.Client=0`

`Ice.Default.Router` specifies which Glacier2 router the client will use. The endpoint details must match the configuration of Glacier2.

`Ice.ACM.Client` must be disabled by explicitly setting it to zero (or set to a value larger than Glacier2's session timeout). (ACM is enabled by default.)

You should also disable retries by setting:

- `Ice.RetryIntervals=-1`



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-9

Notes:

For the client to transparently communicate with its servers via Glacier2, it must be configured by setting the `Ice.Default.Router` property. Once set, this property takes care of transparently redirecting all client requests to Glacier2. The default identity of the router is `Glacier2/router` (that is, the category of the object identity is `Glacier2` and its name is `router`.) You can change the identity used by the Glacier2 router by setting the `Glacier2.InstanceName` property in the Glacier2 configuration.

You must disable automatic connection management (ACM). ACM is enabled by default and closes connections that have been idle for more than one minute. This is transparent to the client: if a client makes another invocation after more than a minute of idle time, the Ice run time automatically re-establishes a connection. However, with Glacier2, ACM must be disabled because, once a connection is lost, the Ice run time cannot transparently re-establish it because doing so requires re-authentication and re-creating a session object. (It is okay to leave ACM enabled, provided its timeout value is larger than Glacier2's session timeout. Refer to Section 14-14.)

For the same reason, you should disable retries by setting the `Ice.RetryIntervals` property to `-1`. Normally, the Ice run time will attempt to transparently re-establish a lost connection before raising a `ConnectionLostException` in the client. But, in the presence of sessions, connections cannot be re-established. Not disabling retries does no real harm, but delays the `ConnectionLostException` that is raised in the client until after the attempt to re-establish the connection has failed. Disabling retries avoids this delay.

14-10 Creating a Password File

Creating a Password File

Glacier2 uses a password file to authenticate clients.

The password file contains one line for each user, with the user name and encrypted password:

```
Joe ZpYd5t1p4.d0Y
```

```
marc G8Y0Z670gnlwl
```

You must configure the name and location of the password file by setting **Glacier2.CryptPasswords** to the path name of the file.

You can use the **openssl** utility to create encrypted passwords:

```
$ openssl
```

```
OpenSSL> passwd
```

```
Password: openSesame
```

```
Verifying - Password: openSesame
```

```
ZpYd5t1p4.d0Y
```



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-10

Notes:

Before clients can use Glacier2, you must allocate user IDs and passwords to clients. By default, Glacier2 uses the UNIX crypt algorithm for passwords. The password file that Glacier2 uses is configured with the **Glacier2.CryptPasswords** property. (A relative path name for this property is interpreted as a path relative to Glacier2's working directory.) The password file contains one line for each user, with the user name and password separated by white space.

As shown above, you can use the **openssl** tool to generate the encrypted passwords.

14-11 Custom Authentication

Custom Authentication

You can implement a custom authentication mechanism by implementing the `PermissionsVerifier` interface:

```
module Glacier2 {
  interface PermissionsVerifier
  {
    Idempotent bool checkPermissions(
      string userId,
      string password,
      out string reason);
  };
};
```

Set `Glacier2.PermissionsVerifier` to the proxy of this object.

If set, Glacier2 uses the specified verifier instead of the default password mechanism.

`checkPermissions` must return true if authentication is successful, false otherwise.



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-11

Notes:

You can integrate Glacier2 with a pre-existing authentication mechanism by implementing an Ice object of type `PermissionsVerifier` and setting `Glacier2.PermissionsVerifier` to the proxy of that object. If set, this property causes Glacier2 to delegate authentication to the specified object, instead of using the default password file mechanism. Your implementation of `checkPermissions` must return true if authentication is successful, false otherwise. If the return value is false, you return an explanation of why authentication failed in the out-parameter `reason`. That string is made available to the client in the corresponding `PermissionDeniedException` that is raised from the client's call to `createSession`.

Keep in mind that, if Glacier2 communicates with the permissions verifier over an unsecure network, you should configure the verifier's proxy for SSL, otherwise the user name and password will be sent over the network in clear text.

14-12 The Admin Interface

The Admin Interface

The Admin interface allows remote shut-down of Glacier2:

```
module Glacier2 {  
  interface Admin  
  {  
    void shutdown();  
  };  
};
```

The default identity of this interface is `Glacier2/admin`.

The endpoint at which this object listens is determined by the property `Glacier2.Admin.Endpoints`.

If the property is not set, Glacier2 does not enable this interface.

Do not make this object available on a public network or, if you do, only use an SSL endpoint!



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-12

Notes:

The `Admin` interface permits remote shut-down of Glacier2.¹ Make sure to secure this interface correctly (if you enable it at all).

¹ Currently, this interface is rather bare. ZeroC will add to it over time, according to customer demand.

14-13 Custom Object Identities

Custom Object Identities

If you are running several Glacier2 processes, you will need to use different object identities for each one.

- `Glacier2.InstanceName`

This property changes the identity of the **Router** and **Admin** objects in Glacier2. For example:

- `Glacier2.InstanceName=Fred`

results in **Fred/router** and **Fred/admin** as the identities of the router and admin objects.

If you change the identity, the client configuration must be changed accordingly:

- `Ice.Default.Router=Fred/router: -h 5.6.7.8 -p 4063`

Glacier2
Copyright © 2005-2010 ZeroC, Inc.14-13

Notes:

You may need to run several Glacier2 instances, for example, to permit clients to access two distinct corporate servers in two different networks. You can assign a custom identity to Glacier2's Router and Admin objects by setting the property **Glacier2.InstanceName**.

14-14 Session Timeouts

Session Timeouts

Unless you configure a timeout, session state is maintained by Glacier2 indefinitely.

To configure a timeout, set the property

`Glacier2.SessionTimeout` to the timeout value in seconds.

Any client activity resets the timeout. If there is no activity on the session for the specified timeout, Glacier 2 destroys the session.

If a session is destroyed, the client must create a new session, reauthenticating itself.

A destroyed session results in a `ConnectionLostException` in the client.



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-16

Notes:

In general, you should set a session timeout when using Glacier2. If you do not, session state is maintained indefinitely for each client. This means that, if a client crashes or otherwise gets disconnected, Glacier2 retains the state for that client's session until the client creates a new session (in which case the new session replaces the old one).

The `Glacier2.SessionTimeout` property sets the timeout value in seconds. Any inactivity on the client's session for longer than the configured timeout results in a `ConnectionLostException` in the client when the client invokes another operation.

If you do not configure a session timeout, session state is retained indefinitely by Glacier2. An easy way to ensure that the client's session stays alive is to use a similar approach as outlined in Chapter 12: run a background thread in the client that periodically calls `refreshSession` on the `Router` object. Note that the helper class `Glacier2.Application` (see Section 14-19) handles this task for you.

14-15 Explicit Session Management

Explicit Session Management

If you need to track session activity of clients, you can create an external session:

- Implement the Glacier2 **SessionManager** interface
- Configure Glacier2 to use your session manager by setting `Glacier2.SessionManager` to the session manager's proxy.
- Implement the Glacier2 **Session** interface.

If you use explicit session management, your **create** operation can return the session's proxy to the client. In that case, the client receives a non-null proxy (instead of the null proxy it gets for an internal session).

Explicit session management is useful to, for example, log when clients create and destroy a session.

Your **create** operation must handle re-creating a dropped session.

Note that, for SSL, there is also an **SSLSessionManager**.



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-15

Notes:

You can use explicit session management for Glacier2 (for example, to log when sessions are created or destroyed, or to set up other contextual information).

Glacier2 delegates session creation to a **SessionManager**. You can change the session manager that is used by Glacier2 by setting the `Glacier2.SessionManager` property to the session manager's proxy. Here are the relevant interfaces:

```
module Glacier2 {
    exception CannotCreateSessionException {
        string reason;
    };
    interface Session {
        void destroy();
    };
    interface SessionControl {
        ...
    };
    interface SessionManager {
        Session* create(string userId, SessionControl* control)
            throws CannotCreateSessionException;
    };
}
```

```
    };  
};
```

Your session manager must provide an implementation of the `create` operation. Note that `create` must be able to handle creation of a session even if a session for the same user already exists. This is necessary because, for example, a client may have lost connectivity with Glacier2 and attempts to re-create its session. (This is especially important if you do not set a session timeout—if your `create` operation cannot create a session for a user who already has an existing session, each user can create a session exactly once, but never again until you restart Glacier2.)

The `create` operation can (but need not) return the proxy for the new session. If you do return a non-null proxy, the client can explicitly destroy its session by calling the session's `destroy` operation. (If you return a null proxy, clients can still destroy an internal session by calling the router's `destroySession` operation.)

Note that you can independently replace Glacier2's `PermissionsVerifier` as well; Glacier2 calls your `create` operation only *after* it has authenticated the client.

If you use Glacier2 with SSL, you can replace the corresponding `SSLPermissionsVerifier`.

14-16 Supporting Callbacks

Supporting Callbacks

The diagram shows a Client on the left and a Server on the right, with a Glacier2 server in the center. The Client and Glacier2 are connected via an External Network using the address 5.6.7.8:4068 over a Bidirectional Connection. The Glacier2 server is also connected to an Internal Network. The Glacier2 server has an endpoint at 10.1.1.1 in the Internal Network. The Server is connected to Glacier2 via the Internal Network using the address 10.1.1.2.

To support callbacks from server to client, Glacier2 must have an endpoint in the internal network.

The `Glacier2.Server.Endpoints` property configures that endpoint. The property does not require a port, only a host name or IP address:

```
Glacier2.Server.Endpoints=tcp -h 10.1.1.1
```

No code changes are required in the server for callbacks.

 ZeroC
Glacier2
Copyright © 2005-2010 ZeroC, Inc.
14-16

Notes:

A server that makes callbacks on objects provided to it by the client cannot directly invoke the operation on the client. Instead, the invocation must be routed back to the client via Glacier2. For this to be possible, Glacier2 must provide an endpoint in the internal network so the server can connect to Glacier2 for its outgoing callback invocations. The endpoint for server callbacks is set via the `Glacier2.Server.Endpoints` property. Note that this property does not require a port number. (A host name or IP address is sufficient.)

On the client side, Glacier2 forwards the server's invocation to the client over a bidirectional connection. This is necessary because the client may be behind a firewall of its own, and that firewall may not permit incoming connections. By forwarding the server invocation to the client over the same connection that the client uses to connect to the router, Glacier2 can ensure that callbacks work even if the client is behind such a firewall.

14-17 Supporting Callbacks (cont. 1)

Supporting Callbacks (1)

Client requirements for callbacks:

- The client must have an object adapter (but no local endpoint is necessary).
- Callback proxies created by this object adapter must use the *router's* server endpoint so that callback invocations from back-end servers are sent to the router, and not sent directly to the client.
- To achieve this, the client must configure its callback object adapter with a proxy for the router, using the `<adapter-name>.Router` property or by calling `createObjectAdapterWithRouter`.



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-17

Notes:

If you look back at the picture in Section 14-16, you will see that, in order for the server to invoke an operation on a client's callback object, the server must connect to Glacier2's internal server endpoint instead of connecting directly to the client's endpoint. This means that any proxies that the client creates for its callback objects must contain Glacier2's internal server endpoint instead of the client's endpoint (as would normally be the case).

To achieve this, the object adapter that the client uses for its callback objects must be configured with Glacier2's router proxy by setting the `<adapter-name>.Router` property:

```
<adapter-name>.Router=Glacier2/router:tcp -h 5.6.7.8 -p 4063
```

`<adapter-name>` must match the adapter name that the client uses for its callback adapter. (Instead of using a property, you can also call `createObjectAdapterWithRouter` when creating the adapter—see the Ice manual for details.)

Internally, the Ice run time ensures that, when a client's adapter is configured with a router, any proxies created by the client contain the router's internal server endpoint. (The client's object adapter obtains that information by invoking an operation on the router that informs it of the internal server endpoint.) This also explains why it is not necessary to specify a port number for the `Glacier2.Server.Endpoints` property: Glacier2 simply uses a free port (assigned by the operating system) and, when the client's object adapter asks the router for the server endpoint, the correct port number is returned to the client-side run time.

14-18 Supporting Callbacks (cont. 2)

Supporting Callbacks (2)

When a server makes a callback, Glacier2 has to work out which client the callback should go to.

For each client session, Glacier2 generates a unique category. That category *must* be used by a client in the identity of its callback objects.

```
Ice.RouterPrx r = communicator.getDefaultRouter();
Glacier2.RouterPrx router =
    Glacier2.RouterPrxHelper.checkedCast(r);
String category = router.getCategoryForClient();

Ice.Identity id = new Ice.Identity();
id.category = category;
id.name = java.util.UUID.randomUUID().toString();
SomeObject p = new SomeObject();
adapter.add(p, id);
```

Because each client uses a different category, Glacier2 can examine the category to determine to which client to forward a callback made by the server.



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-18

Notes:

In order for callbacks to work correctly, Glacier2 must ensure that a callback that is made by a server ends up in the correct client. However, the proxy for a callback object contains Glacier2's server endpoint instead of the client's endpoint. This means that Glacier2, when it receives a callback invocation from a server, cannot use the endpoint on which the server makes its callback to identify the client that should receive that invocation.

This means that the only other information available for Glacier2 to identify the correct client is the object identity or, specifically, the **category** member of the object identity:

- For each client session, Glacier2 generates a unique category string.
- Clients must use that category string for their callback objects. (The **name** member of the object identity can be freely chosen by the client.)
- Clients can obtain the correct category to use by calling **getCategoryForClient** on the Glacier2 router.

14-19 Helper Classes

Helper Classes

Ice includes helper classes that provide functionality commonly needed by Glacier2 clients:

- **Glacier2.Application** is a subclass of **Ice.Application** that simplifies the use of Glacier2 in command-line applications.
- **Glacier2.SessionFactoryHelper** and **Glacier2.SessionHelper** offer greater flexibility for graphical clients.

Glacier2
Copyright © 2005-2010 ZeroC, Inc.14-19

Notes:

Ice includes a number of helper classes to help you build robust Glacier2 clients.

The **Ice.Application** class, which we discussed in Chapter 6, encapsulates some basic Ice functionality such as communicator initialization, communicator destruction, and proper handling of signals and exceptions. The **Glacier2.Application** class extends **Ice.Application** to add functionality that is commonly needed by Glacier2 clients:

- Keeps a session alive by periodically calling **refreshSession** from a background thread
- Automatically restarts a session if a failure occurs
- Optionally creates an object adapter for callbacks
- Destroys the session when the application completes

The **Glacier2.Application** class is designed primarily for command-line applications. Ice also includes a separate collection of classes that offer functionality similar to **Glacier2.Application** but with greater flexibility to suit the needs of graphical clients (see the Ice manual for details).

14-20 The Glacier2.Application Class

Glacier2.Application

The `Glacier2.Application` class should look familiar to users of `Ice.Application`:

```
package Glacier2;
public abstract class Application extends Ice.Application {
    public class RestartSessionException extends Exception { }
    public Application();
    public Application(SignalPolicy signalPolicy);
    public abstract Glacier2.SessionPrx createSession();
    public abstract int runWithSession(String[] args)
        throws RestartSessionException;
    public static Glacier2.RouterPrx router();
    public static Glacier2.SessionPrx session();
    // ...
}
```

Subclasses must implement `createSession` and `runWithSession`.



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-20

Notes:

As for `Ice.Application`, the application's `main` method must instantiate the `Glacier2.Application` subclass and invoke its `main` method. In turn, `main` calls `createSession` followed by `runWithSession`, both of which must be implemented by the subclass.

As its name implies, `createSession` is where the subclass creates a new session with the Glacier2 router. For example, the implementation might prompt the user for a user name and password, and then invoke `createSession` on the router proxy (the router proxy is available via the static `router` method). The implementation must return the proxy for the new session (if any); the application can obtain this proxy at any time using the static `session` method.

The `runWithSession` method contains the application's main loop and is analogous to the `run` method in `Ice.Application`. The argument vector passed to `runWithSession` contains the arguments passed to `Application.main` with all Ice-related options removed. The implementation must return zero to indicate success and non-zero to indicate failure; this value becomes the return value of `Application.main`.

Here is an example of a minimal subclass:

```
public class Glacier2App extends Glacier2.Application
{
    public Glacier2.SessionPrx createSession()
    {
        String user = ...;
        String password = ...;
        try
        {
            return router().createSession(user, password);
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
            System.exit(1);
        }
    }

    public int runWithSession(String[] args)
        throws Glacier2.RestartSessionException
    {
        // main loop goes here ...
        return 0;
    }

    public static void main(String[] args)
    {
        Glacier2App app = new Glacier2App();
        int status = app.main(args);
        System.exit(status);
    }
}
```

14-21 The Glacier2.Application Class (cont. 1)

Glacier2.Application (1)

Additional convenience methods simplify callbacks and session management:

```
package Glacier2;
public abstract class Application extends Ice.Application {
    // ...
    public void sessionDestroyed();
    public void restart()
        throws RestartSessionException;
    public String categoryForClient()
        throws SessionNotExistException;
    public Ice.Identity createCallbackIdentity(String name);
    public Ice.ObjectPrx addWithUUID(Ice.Object servant);
    public Ice.ObjectAdapter objectAdapter()
        throws SessionNotExistException;
}
```



Glacier2
Copyright © 2005-2010 ZeroC, Inc.

14-21

Notes:

Glacier2.Application provides a number of additional methods that Glacier2 applications will find useful:

- **sessionDestroyed**
This is a callback method that **Application** invokes after destroying the current session. A subclass can override this method to take action when connectivity with the Glacier2 router is lost.
- **restart**
At any time, **runWithSession** can raise **RestartSessionException** to restart the current session. The **restart** method is provided as a convenience for you to call when you wish to restart the session; this method always raises **RestartSessionException** and therefore it never returns. **main** traps the exception and restarts the session by invoking **createSession** followed by another call to **runWithSession**.
- **categoryForClient**

This method returns the category to be used in the identities of all of the client's callback objects. The method raises `SessionNotExistException` if it is called when no session has been established. The category string is cached to minimize unnecessary remote invocations on the router.

- `createCallbackIdentity`

This method creates a new Ice identity for a callback object with the given identity name. The category of the identity is set to the return value of `categoryForClient`.

- `addWithUUID`

This method adds a servant to the callback object adapter's Active Servant Map using a UUID for the identity name and the return value of `categoryForClient` for the identity category.

- `objectAdapter`

This method returns the object adapter used for callbacks, creating it if necessary.

15 Assignment 7 Using Glacier2

15 Assignment 7: Using Glacier2

15-1 Exercise Overview

Exercise Overview

In this exercise, you will

- modify the file system application to work with Glacier2.

By the end of this exercise, you will have gained experience in how to configure Glacier2 and your applications, create Glacier2 sessions, and communicate via Glacier2.

Assignment 7 Using Glacier2
Copyright © 2005-2010 ZeroC, Inc.15-1

Notes:

In this exercise, you will modify the file system application to work with Glacier2.

15-1-1 Exercise Objectives

By the completion of this exercise, you will have gained experience in how to configure Glacier2 and your applications, create Glacier2 sessions, and communicate via Glacier2.

15-2 Using Glacier2

Using Glacier2

- In this exercise, you will modify the application you developed in Assignment 6 to communicate via Glacier2.



Assignment 7 Using Glacier2
Copyright © 2005-2010 ZeroC, Inc.

15-2

Notes:

In this exercise, you will modify the application you developed in Assignment 6 to communicate via Glacier2.

15-3 What You Need to Do

What You Need to Do

1. The client needs to communicate with the server via Glacier2. Change the client to use `Glacier2.Application` and add the missing code to create a Glacier2 session for the client.
2. Create a configuration file for Glacier2. For this exercise, because we do not have a real firewall, you will run the client, server, and Glacier2 on the same machine. Use the loopback address (127.0.0.1) for the configuration. Glacier2 should listen for client requests on port 4063. Configure a session timeout of 30 seconds.
3. Create a password file for Glacier2 and modify the client source code to use the correct user name and password.
4. Create a configuration file for the client to work with Glacier2.
5. Run Glacier2, the client, and the server. If you have set things up correctly, the client will list the contents of the server's file system.
6. Run Glacier2, the client, and the server with network tracing enabled. Examine the port numbers that are used to convince yourself that the client indeed communicates via Glacier2.
7. Change the client to use an invalid password and verify that Glacier2 correctly rejects session creation.



Notes:

1. The client needs to communicate with the server via Glacier2. Change the client to use `Glacier2.Application` and add the missing code to create a Glacier2 session for the client.
2. Create a configuration file for Glacier2. For this exercise, because we do not have a real firewall, you will run the client, server, and Glacier2 on the same machine. Use the loopback address (127.0.0.1) for the configuration. Glacier2 should listen for client requests on port 4063. Configure a session timeout of 30 seconds.
3. Create a password file for Glacier2 and modify the client source code to use the correct user name and password.
4. Create a configuration file for the client to work with Glacier2.
5. Run Glacier2, the client, and the server. If you have set things up correctly, the client will list the contents of the server's file system.
6. Run Glacier2, the client, and the server with network tracing enabled. Examine the port numbers that are used to convince you that the client indeed communicates via Glacier2.
7. Change the client to use an invalid password and verify that Glacier2 correctly rejects session creation.

15-4 Client Modifications

Client Modifications

Assignment 7 Using Glacier2
Copyright © 2005-2010 ZeroC, Inc.15-4

Notes:

```
public class Client extends Glacier2.Application
{
    public Glacier2.SessionPrx createSession()
    {
        String username = "joe";
        String password = "joe";
        try
        {
            return router().createSession(username, password);
        }
        catch(Glacier2.PermissionDeniedException ex)
        {
            System.err.println("cannot create session: " + ex.reason);
            ex.printStackTrace();
            System.exit(1);
        }
        catch(Glacier2.CannotCreateSessionException ex)
        {
            System.err.println("cannot create session: " + ex.reason);
            ex.printStackTrace();
        }
    }
}
```

```
        System.exit(1);
    }
}

public int runWithSession(String[] args)
{
    // ...
    return 0;
}
}
```

15-5 Client Configuration

Client Configuration

```
Ice.Default.Router=Glacier2/router:tcp -h 127.0.0.1 -p 4063
Ice.ACM.Client=0
Ice.RetryIntervals=-1
```

Assignment 7 Using Glacier2
Copyright © 2005-2010 ZeroC, Inc.15-5

Notes:

```
Ice.Default.Router=Glacier2/router:tcp -h 127.0.0.1 -p 4063
Ice.ACM.Client=0
Ice.RetryIntervals=-1
```

15-6 Glacier2 Configuration

Glacier2 Configuration

```
Glacier2.Client.Endpoints=tcp -h 127.0.0.1 -p 4063
Glacier2.CryptPasswords=passwords
Glacier2.SessionTimeout=30
```

Assignment 7 Using Glacier2
Copyright © 2005-2010 ZeroC, Inc.15-6

Notes:

```
Glacier2.Client.Endpoints=tcp -h 127.0.0.1 -p 4063
Glacier2.CryptPasswords=passwords
Glacier2.SessionTimeout=30
```

15-7 Glacier2 Password File

Glacier2 Password File

You need one line in the password file with a user name and encrypted password, for example:

```
joe 0WULk8FE9fmwo
```

The encrypted password in this file is “joe”.

Assignment 7 Using Glacier2
Copyright © 2005-2010 ZeroC, Inc.15-7

Notes:

You need one line in the password file with a user name and encrypted password, for example:

```
joe 0WULk8FE9fmwo
```

The encrypted password in this file is “joe”.

16 IceGrid

16 IceGrid

16-1 Lesson Overview

Lesson Overview

- IceGrid is the location and server activation service for Ice.
- In this lesson you will learn to:
 - use IceGrid to start servers on demand
 - avoid hard-coding addresses and port numbers into proxies
 - advertise application objects
 - monitor the status of servers.
- By the completion of this lesson, you will understand how IceGrid works, how to configure clients and servers to take advantage of indirect binding and automatic activation, and how to administer and troubleshoot IceGrid.

IceGrid
Copyright © 2005-2010 ZeroC, Inc.16-1

Notes:

IceGrid is the location and server activation service for Ice. Using IceGrid, you can start servers on demand, avoid hard-coding addresses and port numbers into proxies, advertise application objects, and monitor the status of servers.

16-1-1 Lesson Objectives

By the completion of this lesson, you will understand how IceGrid works, how to configure clients and servers to take advantage of indirect binding and automatic activation, and how to administer and troubleshoot IceGrid.

16-2 IceGrid

IceGrid

IceGrid is a location and activation service:

- IceGrid allows clients to use indirect proxies that do not contain host names (or IP addresses) and port numbers.
- IceGrid can activate servers on demand, when clients first issue a request.
- IceGrid allows well-known proxies to be advertised. Clients can bootstrap using proxies that contain the name of well-known objects, instead of endpoint information.

IceGrid also provides advanced features:

- Replication and load balancing with automatic failover
- Allocation of servers to clients
- Status monitoring
- Application distribution
- Centralized application deployment

IceGrid
Copyright © 2005-2010 ZeroC, Inc.16-2

Notes:

IceGrid is the location and activation service for Ice. The main purpose of IceGrid is to provide location transparency: using IceGrid, proxies no longer need to contain endpoint information (which gets out of date if a server is moved to a different machine). Instead, proxies contain symbolic names that are transparently resolved to the correct endpoint. This also means that it is no longer necessary to run servers at a fixed port. Instead, you can allow the OS to assign an unused port to each server and rely on IceGrid to work out where the server is currently running.

IceGrid also can activate servers on demand: instead of having to start each server manually and run it permanently, IceGrid can activate servers “just in time”, when the first request for a server is sent by a client.

IceGrid provides a registry of well-known objects. Typically, these objects are the ones that clients require to bootstrap (such as the root directory for the file system application). Proxies to well-known objects do not contain endpoint information or an adapter ID, but only a symbolic name.

These features of IceGrid make it easier and more flexible to configure an application. In addition, IceGrid provides a number of advanced features, such as replication and load balancing, allocation of servers to specific clients, application distribution and deployment, as well as status monitoring. These features make it easy to configure and deploy an application on a number of remote machines without having to separately configure the application on each machine.

16-3 IceGrid Components

IceGrid Components

IceGrid consists of a single registry and one or more nodes:

- The IceGrid registry is a database that keeps track of known applications and the servers that make up each application. The registry also knows details such as how to start each server, what command-line options to provide at server start-up, and the values of environment variables to be set for each server. A single registry is used for a number of machines.
- An IceGrid node is essentially a server start-up and monitoring process. On each machine on which IceGrid-aware servers run, an IceGrid node process is required. Each IceGrid node communicates with its IceGrid registry to keep it informed of the status of servers.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-3

Notes:

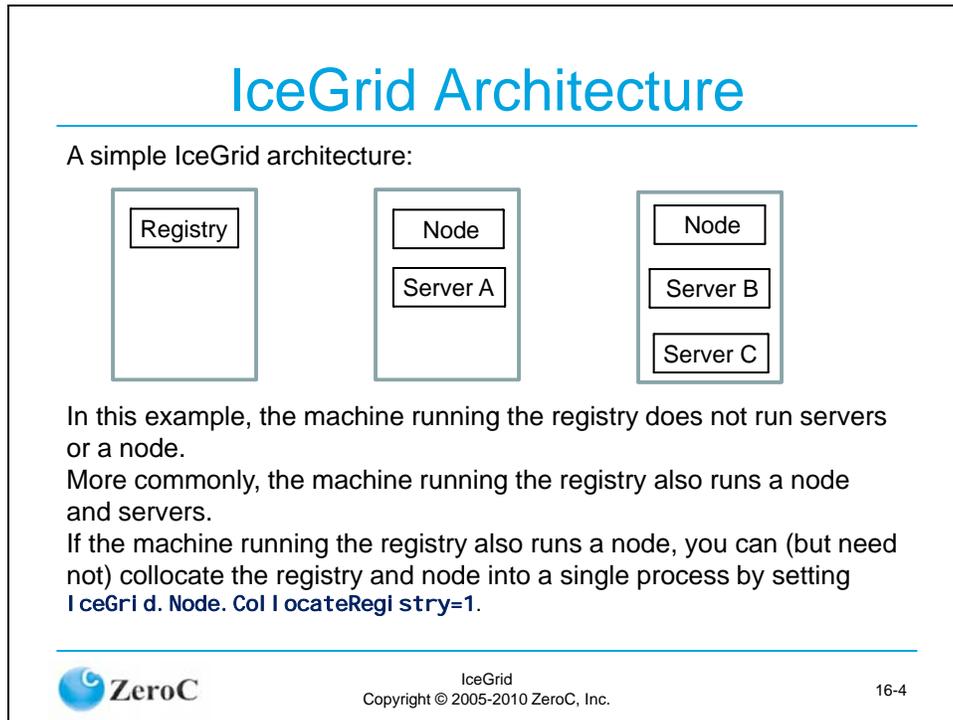
IceGrid uses a single registry, plus one or more nodes. You must run an IceGrid node on each machine on which you are running IceGrid-aware servers.

The job of the registry is to store the names of servers, how to activate them, where each server is currently running, and so on.

The job of each node is to start servers on the node's machine, and to monitor their status.

Each node communicates with the registry and informs the registry of status changes (such as server shutdown).

16-4 IceGrid Architecture



Notes:

One machine in an IceGrid domain must run an IceGrid registry. Every machine on which you want to run servers that are aware of IceGrid, you must run an IceGrid node. If you also want to run application servers on the machine that runs the registry, you must run a node on the registry machine as well. In that case, you can collocate the registry and the node by setting the property `IceGrid.Node.ColloccateRegistry`. If set, the node automatically provides the registry functionality as well; if not set, you must manually start separate registry and node processes.

Clients can be located anywhere, either on one of the machines that are part of the IceGrid domain, or on a completely unrelated machine.

16-5 Indirect Proxies

Indirect Proxies

An indirect proxy has the form

<object-identity>@<adapter-id>

For example:

RootDir@fsadapter

The adapter *ID* is different from the adapter *name* that is used by the server. The adapter ID is configured with the adapter property

<adapter-name>. AdapterId.

The advantage of indirect proxies is that they do not contain endpoint information. If a server is moved to a different machine or port, the client need not be updated.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-5

Notes:

Indirect proxies do not contain endpoint information. Instead, the endpoint information is replaced by a symbolic adapter ID, such as **fsadapter**. When a client uses an indirect proxy, the Ice run time transparently contacts the IceGrid registry to find out what endpoints are used by the server with the specified adapter ID. The registry returns the endpoints to the client-side run time, which then dispatches the request to the correct endpoint.

Note that the IceGrid registry gets involved only the first time a client uses an indirect proxy. Thereafter, all communications between the client and the server are direct, and not via IceGrid.

The main advantage of indirect proxies is that they allow you to move a server to a different machine without having to update the client configuration.

16-6 Client Configuration

Client Configuration

To work with IceGrid, clients require only a single configuration item:

Ice.Default.Locator must be set to the proxy of the IceGrid registry:

```
Ice.Default.Locator=IceGrid/Locator: tcp -h registryhost \
-p 4061
```

This property tells the client-side run time where it can obtain endpoint information for indirect proxies.

IceGrid/Locator is the default identity of the registry's locator service.

Note that the proxy for the locator cannot be an indirect proxy: the run time requires one fixed endpoint at which it can resolve addresses.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-6

Notes:

For clients to work with IceGrid, you must define a single property, **Ice.Default.Locator**. This property informs the client-side run time where it can contact the IceGrid location service in order to resolve indirect proxies. (Setting this property is analogous to configuring the IP address of a DNS server for a machine.) You must set the value of this property to the proxy for the registry's location service. The location service, by default, has the object identity **IceGrid/Locator**.

16-7 Registry Endpoints

Registry Endpoints

The registry provides three endpoints:

```

graph LR
    Registry[Registry] --- Client[Client endpoint  
IceGrid.Registry.Client.Endpoints]
    Registry --- Server[Server endpoint  
IceGrid.Registry.Server.Endpoints]
    Registry --- Internal[Internal endpoint  
IceGrid.Registry.Internal.Endpoints]

```

- Client endpoint
Used by the administrative tools and by clients to resolve indirect proxies
- Server endpoint
Used by servers for status updates and registration
- Internal endpoint
Used by nodes and registry replicas

Two additional endpoints are used if IceGrid runs in conjunction with Glacier2.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-7

Notes:

The IceGrid registry provides three separate endpoints for different purposes.

- Client endpoint
This endpoint provides the location service to clients.
- Server endpoint
This endpoint is used by servers to inform the registry of important events, such as adapter activation and deactivation.
This endpoint is also used by administrative commands to update the registry, for example, with the details of a new application.
- Internal endpoint
This endpoint is used for internal purposes. In particular, the IceGrid nodes use this endpoint to communicate with the registry.

Two additional endpoints can be configured for Glacier2 to create sessions and administrative sessions—see the Ice Manual for details.

16-8 Registry Configuration

Registry Configuration

You must set the following properties for the registry to work:

- `IceGrid.Registry.Client.Endpoints`
- `IceGrid.Registry.Server.Endpoints`
- `IceGrid.Registry.Internal.Endpoints`
- `IceGrid.Registry.Data`

Only the client endpoint requires a port number.

The server and internal endpoints only require a protocol, but not a host or port.

The `IceGrid.Registry.Data` property defines the path to a directory in which the registry places its database files.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-8

Notes:

For the registry to work correctly, you must set the four properties shown above. Note that it is not necessary to specify a host and port for the server and internal endpoints; for these, you need to specify only a protocol. (The registry acts as its own location service for these endpoints, so the port number can be assigned dynamically for these endpoints by the operating system.) The client endpoint, however, requires a port number.

Note that the client's `Ice.Default.Locator` endpoint must match the setting of `IceGrid.Registry.Client.Endpoints`.

The `IceGrid.Registry.Data` property defines the location of a directory in which the registry stores its database files. (Alternatively, the registry can be configured to use a SQL database—see the Ice manual for details.)

Here is a minimal configuration for the registry:

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.Data=registry
```

16-9 Node Configuration

Node Configuration

Each node requires at least the following configuration:

- **Ice.Default.Locator**
Defines the registry's location service proxy.
- **IceGrid.Node.Endpoints**
Defines the node's endpoint for communication with the registry.
- **IceGrid.Node.Name**
A unique name for the node within the IceGrid domain.
- **IceGrid.Node.Data**
The location of the configuration files of servers started by the node.

IceGrid.Node.Name must be different for each node!

If you want to collocate the registry, you can set **IceGrid.Node.CollocateRegistry=1** on exactly one of the nodes in the IceGrid domain.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-9

Notes:

Each node requires four properties to be set:

- **Ice.Default.Locator**
This property defines the proxy for the registry's location service. Its value must match the setting of **IceGrid.Registry.Client.Endpoints**. This property must be defined so the node knows where to find its registry.
- **IceGrid.Node.Endpoints**
This property defines the node's endpoint for communication with the registry. It is not necessary to specify a port number.
- **IceGrid.Node.Name**
Each node has a name that must be unique within the registry. This property defines the name by which the registry identifies the node.
- **IceGrid.Node.Data**
This property defines the directory in which the node stores server configuration files.

A minimal node configuration would be as follows:

```
Ice.Default.Locator=IceGrid/Locator:tcp -p 4061
```

IceGrid.Node.Endpoints=tcp

IceGrid.Node.Name=Node1

IceGrid.Node.Data=node

If you set **IceGrid.Node.CollocateRegistry=1**, the node will also provide registry functionality. In that case, the node's configuration must also include the registry properties discussed in Section 16-8.

16-10 Starting an IceGrid Node

Starting an IceGrid Node

--nowarn: Don't print security warnings.
--readonly: Start the master registry in read-only mode.

UNIX:

- daemon**: Run as UNIX daemon
- noclose**: Don't close open file descriptors for daemon
- nochdir**: Don't change directory to /
- pidfile *file***: Write process ID into specified file.

Windows:

Use the **iceserviceinstall** utility to configure it as a Windows service.

IceGrid
Copyright © 2005-2010 ZeroC, Inc.16-10

Notes:

The **icegridnode** executable implements the IceGrid node. To successfully start the node, you must set the properties described in Section 16-9.

The **--nowarn** option suppresses security warnings. (Refer to Section 16-31.)

The **--readonly** option runs the master registry in read-only mode.

You can run the node as a UNIX daemon by providing the appropriate options, or configure it to run as a Windows service using the **iceserviceinstall** utility. (Usually, in a production environment, the node will run as a Windows service that is automatically activated, or as a UNIX daemon that is activated from **/etc/rc**.)

Note that if you set **IceGrid.Node.CollocateRegistry**, the node will also include a registry. Otherwise, you must start a separate registry process.

16-11 Starting the Registry

Starting the Registry

The registry command is `icegridregistry`.

It supports the `--nowarn` option as well as the same UNIX daemon options as `icegridnode`:

`--daemon, --noconsole, --nochild, --pidfile file`

Use the `iceserviceinstall` utility to configure it as a Windows service.

If you run a separate registry, and start nodes before starting the registry, the nodes will periodically attempt to contact the registry and establish a connection once the registry is running.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-11

Notes:

The registry command is `icegridregistry`. The command provides the same options as `icegridnode`.

You must run exactly one registry per IceGrid domain. (If you run a collocated registry within a node, do not start a separate registry process.)

To successfully start the registry, you must set the properties described in Section 16-8. Each node periodically attempts to contact its registry. Each node will try to contact the registry indefinitely, so starting a node before starting its registry does not cause any problems.

16-12 Server Configuration

Server Configuration

Server configuration is accomplished via XML files.

The XML descriptor at a minimum describes:

- the application name
- the node(s) on which the server(s) run
- for each server:
 - a server ID
 - the server executable file name

Additional descriptor elements can specify:

- the adapter name and protocol
- activation mode (manual, on-demand, etc.)
- command-line options
- property settings
- environment variables



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-12

Notes:

For each application, the registry must know the details of the servers that make up the application. To configure the registry, you write an XML *deployment descriptor* that contains the relevant details. At a minimum, the descriptor must provide an application name, the names of the nodes on which the application's servers run, and, for each server, a server ID and the executable to be run.

Here is a minimal XML descriptor for an application:

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fssserver" exe="/usr/bin/myserver"/>
        <adapter name="Lab8" endpoints="tcp"/>
    </node>
  </application>
</icegrid>
```

The application name must be unique within the registry.

The node name specifies on which node the server is to run; it must match the corresponding node's setting of the `IceGrid.Node.Name` property. An application can have several node elements. If two node elements have the same name, their contents are merged.

Each `node` element typically contains at least one `server` element. Each `server` element must provide a server ID, which uniquely identifies that server within the registry. Each `server` element must also provide an executable path. Note that, in general, you should specify an absolute path name for the executable. (Relative pathnames are interpreted relative to the node's working directory.)

The `server` element performs several important functions:

- It provides the node with information on when and how to start the server executable.
- It associates object adapters and Ice objects with their server.
- It includes whatever additional configuration that the server requires, such as environment variables, property settings, and command-line options.

Using the information in the `server` element, the node automatically generates an Ice configuration file for the server. When the node starts the server, it appends an appropriate `--Ice.Config` option to the server's command line, instructing the server to load this configuration file.

16-13 Server Configuration (cont. 1)

Server Configuration (1)

A server element can have several **option** child elements:

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fserver" exe="/usr/bin/fserver">
        <option>--myoption</option>
        <option>myoptarg</option>
        <adapter name="Lab8" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

Command-line arguments are appended to the executable in the order specified.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-13

Notes:

16-13-1 Command-Line Arguments

You can pass any number of command-line arguments to a server by adding **option** child elements to the **server** element.

When starting a server, the node adds the arguments to the server's executable path in the order in which they are specified.

16-14 Server Configuration (cont. 2)

Server Configuration (2)

You can set properties as part of a server descriptor:

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fsserver" exe="/usr/bin/fsserver">
        <option>Server</option>
        <property name="Ice.ServerIdleTime" value="20"/>
        <property name="Ice.GC.Interval" value="60"/>
        <adapter name="Lab8" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

Property settings are written into a configuration file that is passed to the server on start-up.



Notes:

16-14-1 Properties

You can set properties for a server by adding **property** elements to the **server** element. The property settings are transferred directly to the configuration file that the node generates on behalf of the server. (The properties are set in the same order as specified in the XML descriptor; **property** elements can precede or follow **option** elements.)

You can also use property sets to share settings defined at the application or node level. (See the Ice manual for details.)

16-15 Server Configuration (cont. 3)

Server Configuration (3)

You can set environment variables for the server:

```
<i cegri d>
  <appl i cati on name="fi l esystem">
    <node name="Node1">
      <server i d="fssserver" exe="/opt/app1/bi n/server">
        <env>LD_LI BRARY_PATH="/opt/app1/li b"</env>
        <adapter name="App1" endpoi nts="tcp"/>
      </server>
    </node>
  </appl i cati on>
</i cegri d>
```

For UNIX, use Bourne shell syntax for environment variables.

For Windows, use Windows syntax:

```
<env>PATH=%PATH%; C: /opt/I ce/I l i b</env>
```

`$PATH` (and `$${PATH}`) substitute the setting of an environment variable.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-15

Notes:

16-15-1 Environment Variables

The `env` element can appear as a child of the `server` element and specifies the setting of an environment variable. The IceGrid node ensures that the corresponding environment variable setting is passed to the server on start-up.

You can use more than one `env` element if you need to set several environment variables.

As for the shell, the syntax `$VAR` substitutes the setting of an environment variable. The alternative syntax, `$${VAR}` is useful if you need a delimiter. (Do not use `${VAR}`—that syntax refers to IceGrid variables, not environment variables.)

16-16 Server Configuration (cont. 4)

Server Configuration (4)

Each **server** element must have an **adapter** element for each adapter to which clients bind indirectly.

```
<icegrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fsserver" exe="/usr/bin/fsserver">
        <adapter name="Lab8" endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

The **adapter** element must minimally specify the adapter name (as used by the server).

The endpoint usually only specifies a protocol, so the operating system can assign a port. However, you can specify a port as well, if you want the server to use a specific port.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-16

Notes:

16-16-1 Object Adapter Configuration

For each adapter that is used by a server (and to which you want clients to be able to bind indirectly), you must add an **adapter** element to the **server** element.

Minimally, the **adapter** element must specify the object adapter name as used by the server.

In addition, you can specify endpoints for the adapter. Typically, the endpoints only specify a protocol, but no port number, so the server uses whatever port is available. However, you can also specify a port if you want the server to use a specific port.

If you do not specify a protocol, IceGrid uses the default protocol as specified by the **Ice.Default.Protocol** property. (If that property is not set, the default protocol is TCP/IP.)

If you only specify an adapter's name, but no adapter ID (refer to Section 16-17-1), the server's adapter identifier for clients becomes **<server-id>.<adapter-name>**.

For the above example, the adapter will be known to clients as **fsserver.Lab8**. Therefore, given this deployment, the stringified proxy to the server's root directory would be

RootDir@fsserver.Lab8

16-17 Server Configuration (cont. 5)

Server Configuration (5)

If you specify an `id` attribute, the server's externally visible adapter ID becomes that ID:

```
<IceGrid>
  <application name="filesystem">
    <node name="Node1">
      <server id="fsserver" exe="/opt/app1/server">
        <adapter name="Lab8" id="fsa" endpoints="tcp"/>
      </server>
    </node>
  </application>
</IceGrid>
```

The client's indirect proxy now becomes:
`RootDir@fsa`



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-17

Notes:

16-17-1 Server Configuration (cont.)

You can optionally specify an adapter ID with the `id` attribute. If you do this, the specified ID becomes the ID that clients must use to bind indirectly to that adapter.

16-18 Command-Line Administration

Command-Line Administration

You can maintain the registry from the command line with `icegridadmin`.

The program requires the property `Ice.Default.Locator` to be set so it can find the registry.

`icegridadmin` allows you to:

- Add, update, and remove applications
- Start, stop, and check the status of servers
- Add, remove, and check the status of adapters
- Add, remove, and list well-known objects



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-18

Notes:

You populate the registry with the `icegridadmin` tool.¹ This allows you to add, modify, remove, and list the registry entries, and to check the status of servers.

The tool is interactive and prompts for input similar to the shell. Typing `help` presents the available commands.

Note that the tool requires `Ice.Default.Locator` to be set to the registry's client endpoint; without that setting, the tool does not know where to find the registry.

¹ There is also a GUI version of this tool—see Section 16-28.

16-19 icegridadmin Application Commands

icegridadmin Application Commands

- `application add file.xml`
Add the application described in *file.xml*.
- `application remove name`
Remove the application name.
- `application update file.xml`
Update an already deployed application with *file.xml*.
- `application describe name`
List details of application name.
- `application list`
List all deployed applications.
- `application diff file.xml`
Show differences between deployed application descriptor and *file.xml*.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-19

Notes:

The `application` command allows you to configure the registry with new applications. `file.xml` is the name of the file containing the application's deployment descriptor. For example:

```
$ icegridadmin --Ice.Default.Locator="IceGrid/Locator:tcp -h 127.0.0.1 -p 4061"
>>> application add filesystem.xml
>>> application list
filesystem
>>> application describe filesystem
application `filesystem'
{
  node `Node1'
  {
    servers
    {
      fsserver
    }
  }
}
>>>
```

16-20 icegridadmin Node Commands

icegridadmin Node Commands

- **node list**
List all nodes.
- **node describe *name***
Show information about node *name*.
- **node ping *name***
Test whether node *name* is running.
- **node show *name* [stdout | stderr]**
Show the node's **stdout** and/or **stderr** output.
- **node load *name***
Show the load of node *name*.
- **node shutdown *name***
Shut down the node *name*.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-20

Notes:

The **node** command allows you to monitor and administer nodes.

The registry maintains a heartbeat with its nodes, so if a node dies, the registry becomes aware of that within a few seconds. You can use **node ping** to check whether a node is responding to registry commands.

Under UNIX, the load of a node is shown as the load average. (The load average is the number of runnable processes in the process table over the preceding one, five, and fifteen minutes.) Under Windows, the load of a node is shown as the CPU utilization over the preceding one, five, and fifteen minutes.

16-21 icegridadmin Server Commands

icegridadmin Server Commands

- **server list**
List all server IDs.
- **server describe *id***
Show details of server *id*.
- **server enable *id***
Enable server *id*.
- **server disable *id***
Disable server *id*. (A disabled server cannot be started, either on demand or explicitly.)
- **server stdout *id message***
Write *message* on server *id*'s standard output.
- **server stderr *id message***
Write *message* on server *id*'s standard error.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-21

Notes:

The **server** command allows you to check the details of a server. You can set the **Ice.StdOut** and **Ice.StdErr** properties to the name of a file to redirect standard output and standard error to a file.

Alternatively, on UNIX you can set **Ice.UseSyslog** to a non-zero value to cause the run time to send messages to **syslogd**.

Note that the **server stdout** and **server stderr** commands depend on the server implementing a Process object (Refer to Section 16-25).

16-22 icegridadmin Server Commands (cont. 1)

icegridadmin Server Commands (1)

- **server state *id***
Show the state of the server *id* (running, inactive, enabled or disabled).
- **server pid *id***
Show the process ID of server *id*.
- **server signal *id signal***
Send signal *signal* to server *id* (UNIX only).
- **server start *id***
Start server *id*.
- **server stop *id***
Stop server *id*.
- **server remove *id***
Remove server *id*.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-22

Notes:

The **server** command provides a few other options that allow you to control a server.

Note that the **server signal** command is available only under UNIX. You can specify the signal either using its number or its symbolic name, such as **SIGTERM**.

To start a server, the server must be deployed with the registry. If you start the server using the **icegridadmin** command, the server will be monitored by IceGrid. However, if you start the server by hand (from the command line), its status will not be monitored by IceGrid (but you can still bind to it indirectly, provide that the **Ice.Default.Locator** property is set for the server).

16-23 icegridadmin Server Commands (cont. 2)

icegridadmin Server Commands (2)

- `server show [options] id [stdout | stderr | log]`
Print text from the server's stdout, stderr, or specified log file.
- `server properties id`
Show the run-time properties of the server *id*.
- `server property id name`
Show the setting of the property *name* for the server *id*.
- `server patch id`
Apply updates to the server *id* via IcePatch2.



Notes:

The **show** command displays a server's output, and the **properties** and **property** commands allow you to check the property settings of a server.

The **patch** command allows you to apply software updates via IcePatch2. (See the Ice manual for details.)

16-24 Server Activation and Deactivation

Server Activation and Deactivation

The **activation** attribute of a server element can be set to “manual”, “on-demand”, “session”, or “always”. (The default is “manual”.)

```
<server id="fserver" exe="java" activation="on-demand">
```

- If set to “manual”, the server must be started using the **icegridadmin server start** command.
- If set to “on-demand”, IceGrid transparently activates the server when a client resolves the first indirect proxy to an object in the server.

The easiest way to deactivate a server is to set **Ice.ServerIdleTime** to a timeout in seconds.

If the server is idle for the specified timeout, its object adapters shut down and **waitForShutdown** completes.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-24

Notes:

If you set the **activation** attribute of a **server** element to “on-demand”, the server will be transparently started by IceGrid when a client first resolves an indirect reference to the server.

If you set **Ice.ServerIdleTime**, the server’s object adapters shut down once the server has been idle for the specified amount of time, and the communicator’s **waitForShutdown** method returns.

Automatically terminating idle servers is useful to conserve operating system resources, such as memory, sockets, and file handles. (A server consumes resources even if it is completely idle.)

16-25 The Process Object

The Process Administrative Facet

```

module Ice {
    interface Process {
        idempotent void shutdown();
        void writeMessage(string message, int fd);
    };
};

```

IceGrid adds a **Process** facet to an adapter that runs at the server's **Ice.Admin.Endpoint** (127.0.0.1 by default). The **server stop** command calls **shutdown** on the facet and the implementation of that operation calls **shutdown** on the communicator.

This allows the server to shut down when asked to do so by **icegridadmin**.

You can specify a different admin endpoint for the server by setting the **Ice.Admin.Endpoints** property for the server.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-25

Notes:

The **server stop** command invokes the **shutdown** operation on the **Process** administrative facet. By default, the endpoint for this facet is 127.0.0.1 (for security reasons). This endpoint is used only by the IceGrid node. If the loopback interface on the machine is not secured (that is, the machine has multiple users, not all of whom you can trust), you should use an SSL endpoint instead of a TCP endpoint.

The server informs IceGrid of the proxy to its **Process** facet, so **icegridadmin** can (via the IceGrid node) shut down a server.

If a server disables the administrative facet (by not setting **Ice.Admin.Endpoint**), the node sends a signal to the server in an attempt to shut it down gracefully. Depending on the environment, this may or may not work; for example, on Windows and for a Java server, **server stop** sends a Ctrl-Break to the server's JVM, which causes the JVM to dump its threads and stop executing (so the server does not shut down cleanly). If the server does not terminate within the activation timeout, the node kills the server.

The **writeMessage** operation writes to the server's standard output or standard error and is used by the **server stdout** and **server stderr** commands. (These commands do not work if the server's **Process** facet is disabled.)

16-26 Server Environment

Server Environment

When you use `application add` or `application update`, `icegridadmin` writes a configuration file for a server.

The configuration file is stored in
`<node-dir>/servers/<server-id>/config/config`

For example:

`Node1/servers/fserver/config/config`

This configuration file contains any property settings you specified in the deployment descriptor.

When IceGrid starts a server, it passes

`--Ice.Config=Node1/servers/fserver/config/config`

as an option to the server, so the server gets the correct configuration.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-26

Notes:

When a server is started by IceGrid, it is informed of its configuration via command-line options: IceGrid appends the `--Ice.Config` property settings to the command line of the server.

The remaining configuration (in the form of additional property settings) is passed to the server in the configuration file that is specified by `--Ice.Config`. (That configuration file is written in the node's data directory every time you use the `application add` or `application update` command.)

16-27 Server Code Changes

Server Code Changes

The server's object adapter obtains its endpoints from the property settings generated by the IceGrid node.

When creating an object adapter, use:

```
communicator.createObjectAdapter("<adapter-name>")
```

without specifying any endpoints.

The adapter name must match the **name** attribute of the **adapter** element in the server's deployment descriptor.

The adapter will listen on the endpoints specified by the **adapter** element, which are transferred to the **<adapter-name>.Endpoints** property.

The adapter informs IceGrid of its endpoints so the registry can resolve indirect proxies.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-27

Notes:

For a server to create indirect references and cooperate with Ice, you need to change the code that creates the object adapter to not specify an endpoint directly (that is, use **createObjectAdapter** instead of **createObjectAdapterWithEndpoints**). That way, the adapter retrieves its endpoints from property settings (specifically, the **<adapter-name>.Endpoints** property) and reports these endpoints to IceGrid to facilitate the resolution of indirect proxies.

External to the code, the only other change for a server is that **Ice.Default.Locator** must be set (which is passed to the server on the command line).

16-28 The Graphical Admin Tool

The Graphical Admin Tool

Ice provides a GUI tool that provides most of the functionality of `icegridadmin`.

The tool is provided as a stand-alone jar file in the Ice distribution.

To start the tool:

```
java -jar IceGridGUI.jar
```

The tool prompts for the value of `Ice.DefaultLocator` so it can find the registry.

For the GUI tool to work, either:

- set `IceGrid.Registry.CryptPasswords`
- or set one of the following properties to a custom verifier:
 - `IceGrid.Registry.AdminPermissionsVerifier` (for TCP)
 - `IceGrid.Registry.AdminSSLPermissionsVerifier` (for SSL)



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-27

Notes:

You can perform most of the functions of `icegridadmin` with a graphical tool. The tool provides a tree view of applications, nodes, servers, and adapters, and makes it easy to change the configuration of an application.

For the GUI to be able to access the registry, the registry must be configured with a permissions verifier. You do this by setting either:

`IceGrid.Registry.AdminPermissionsVerifier` or

`IceGrid.Registry.AdminSSLPermissionsVerifier`, depending on whether the registry is configured to use TCP or SSL.

The property value must be the proxy to an object that implements the `Glacier2::PermissionsVerifier` interface (see Chapter 14). IceGrid provides a built-in null permissions verifier with the following object identity:

`IceGrid/NullPermissionsVerifier`

Note that this permissions verifier permits any user name and password, so use it only for testing!

Alternatively, IceGrid provides a built-in permissions verifier that uses the crypt algorithm. If you want to use this built-in permissions verifier, set **IceGrid.Registry.CryptPasswords** to the path name of the password file (as for Glacier2). If you want to use a password file, you must leave the **AdminPermissionsVerifier** and **AdminSSLPermissionsVerifier** properties undefined because they take precedence over **IceGrid.Registry.CryptPasswords**.

16-29 Well-Known Proxies

Well-Known Objects

The registry maintains a table of well-known objects. The table stores a name-proxy pair. You can populate the table

- via the deployment descriptor
- via the `IceGridAdmin` or `IceGridGUI.jar` tools
- programmatically, via the registry's `Slice` interface

A proxy for a well-known object consists of only an identity.

Minimally (using the default protocol), a proxy is:

`RootDir`

You can add an object element as a child of an adapter element to declare a well-known object:

```
<adapter name="Lab8" id="fsadapter" endpoints="tcp">
  <object identity="RootDir"/>
</adapter>
```

`RootDir` and `RootDir@fsadapter` are now equivalent.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-29

Notes:

Typically, clients require a number of proxies to “bootstrap” themselves (such as a proxy to the root directory for the file system application). You can advertise such proxies with IceGrid as well-known objects. When a client uses a proxy for a well-known object, the client-side run time first asks the locator to resolve the object’s identity. (In the above example, this returns the indirect proxy `RootDir@fsadapter` to the client-side run time.) If the proxy that is returned is indirect, the client-side run time then resolves that proxy to the corresponding adapter’s endpoint as usual.

Well-known objects are useful to simplify the configuration of clients. (The only configuration item that is needed by a client is the `Ice.Default.Locator` property, so configuration can be reduced to a single property.)

With `icegridadmin`, you can use the following commands:²

- `object add proxy`

Add a well-known object to the registry.

² IceGridGUI provides equivalent functionality.

- `object remove identity`
Remove the well-known object with *identity* from the registry.
- `object [expr]`
List all well-known objects matching *expr*. (*expr* can contain a trailing * as a wildcard.)
- `object describe [expr]`
Show details of the well-known objects matching *expr*.

16-30 Well-Known Proxies (cont. 1)

Well-Known Proxies (1)

```
module IceGrid {
    interface Admin {
        void addObject(Object* obj)
            throws ObjectExistsException,
                DeploymentException;
        void updateObject(Object* obj)
            throws ObjectNotRegisteredException,
                DeploymentException;
        void addObjectWithType(Object* obj, string type)
            throws ObjectExistsException,
                DeploymentException;
        void removeObject(Ice::Identity id)
            throws ObjectNotRegisteredException,
                DeploymentException;
        idempotent ObjectInfoSeq getAllObjectInfos(
            string expr);
        idempotent ObjectInfo getObjectInfo(Ice::Identity id)
            throws ObjectNotRegisteredException;
    };
};
```



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-29

Notes:

Programmatically, you can manipulate the well-known proxy table via the `IceGrid::Admin` interface.

To obtain a proxy to the `IceGrid::Admin` interface, a client must first create an administrative session with the IceGrid registry and call `IceGrid::AdminSession::getAdmin`. (See the Ice manual for more information.)

16-31 Security Considerations

Security Considerations

Do not permit the server, node, and internal endpoints to be accessible in hostile environments.

In hostile environments, you must use SSL and appropriate certificates to secure these endpoints.

- Under UNIX, if you run the node as a user other than `root`, servers are started with that user ID.
- If you run the node as `root`:
 - If you do not specify a user attribute for the server descriptor, the server runs as `nobody`.
 - Otherwise, it runs as the specified user.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-31

Notes:

You need to pay careful attention to the security implications of using IceGrid. One thing to keep in mind is that anyone who can access the internal interface can run any process as the user ID that is used to start `icegridnode`. Similarly, anyone who can access the server endpoint can substitute a different server for the real one. The node endpoint must be kept secure as well because it allows a hostile client to manipulate a node and execute arbitrary processes on the node.

Under UNIX, if the node does not run as `root`, any servers started by the node run as that user. Otherwise, if you run the node as `root`, but do not specify a `user` attribute with the server descriptor, the server runs as user `nobody`. If the node runs as `root`, you can use the `user` attribute of the server descriptor to specify a user ID for a server. (See the Ice manual for details.)

16-32 Troubleshooting

Troubleshooting

You can set various tracing properties to diagnose problems:

`IceGrid.Registry.Trace.Adapter=3`

`IceGrid.Registry.Trace.Node=2`

`IceGrid.Registry.Trace.Server=1`

`IceGrid.Registry.Trace.Object=1`

`IceGrid.Registry.Trace.Locator=2`

`IceGrid.Node.Trace.Activator=3`

`IceGrid.Node.Trace.Adapter=3`

`IceGrid.Node.Trace.Server=3`

These properties produce trace messages for the corresponding area of interest.

If you run the registry/node in a window from the command line, trace output is written to the terminal.

Beware of relative pathnames for executables and files.

Failure to start a server can be related to `LD_LIBRARY_PATH`.

Check for core files in the node's working directory.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-32

Notes:

You can set the properties shown above to obtain trace information about IceGrid activity. These properties are useful especially when a server does not start as expected. If a client's indirect proxy does not bind, use `icegridadmin adapter list` and `object list` to verify that adapter name and object identity match. You can also set `Ice.Trace.Locator` to trace the activities of the locator on the client side. Running the client with `Ice.Trace.Protocol=1` is also useful because it allows you to see the operation names and object identities that are used, and to pinpoint exactly which operation invocation causes the problem.

During debugging, it can be useful to run the registry, a node, the client, and the server each in a separate window, so you get separate trace from each process. If you can successfully start a server using `server start`, but the server fails to activate on-demand, the problem is likely to be a mismatch of the adapter ID.

16-33 Other Features

Other Features

IceGrid provides a number of other features (not further covered here):

- **Templates**
Templates are generic deployment descriptors so you can describe a whole family of servers with a single descriptor.
- **Server allocation**
You can reserve a specific number of server instances for allocation and exclusive use by particular clients.
- **Replication**
You can replicate objects across a number of servers; if one server is down, IceGrid transparently chooses a working replica in a different server on behalf of clients. You can also replicate the IceGrid registry to achieve fault tolerance.
- **Load balancing**
For replicated objects, IceGrid can dynamically load balance among the objects.



IceGrid
Copyright © 2005-2010 ZeroC, Inc.

16-33

Notes:

IceGrid supports a number of other features that are not covered in this course. Please consult the Ice manual for details.

17 Assignment 8

Using IceGrid

17 Assignment 8: Using IceGrid

17-1 Exercise Overview

Exercise Overview

In this exercise, you will:

- modify the file system application to work with IceGrid.

By the completion of this exercise, you will have gained experience in how to run IceGrid, deploy a server, and use indirect proxies in clients to bind indirectly to Ice objects.

Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.17-1

Notes:

In this exercise, you will modify the file system application to work with IceGrid.

17-1-1 Exercise Objectives

By the completion of this exercise, you will have gained experience in how to run IceGrid, deploy a server, and use indirect proxies in clients to bind indirectly to Ice objects.

17-2 Using IceGrid

Using IceGrid

- In this exercise, you will modify the application you developed in Assignment 6 to use IceGrid.

Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.17-2

Notes:

In this exercise, you will modify the application you developed in Assignment 6 to use IceGrid.

17-3 What You Need to Do

What You Need to Do

1. Run an IceGrid registry in a window.
2. Run an IceGrid node in a separate window.
3. Create a deployment descriptor for your server in `filesystem.xml`.
4. Run `icegridadmin` in a separate window.
5. The server in `Server.java` does not create an object adapter. Add the missing code to create the adapter.
6. Start the server using `icegridadmin`.
7. Verify that you can cleanly stop the server using `icegridadmin`.
8. The client requires configuration to bind indirect references. Place the missing configuration for the client into `config.client`.
9. Modify the client source code to specify an indirect reference for the root directory that matches the deployment of your server.
10. Run the client with protocol tracing and examine the messages that are exchanged between the client and the registry.
11. Run the IceGridGUI tool and use it to modify the server's deployment to advertise the root directory as a well-known object with the identity "RootDir".



Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.

17-3

Notes:

1. Run an IceGrid registry in a window. The registry won't start without appropriate configuration. Add the missing configuration entries to `config.registry` and start the registry with that configuration file. Your configuration file should enable all the registry tracing properties, so you can see what the registry is doing.
2. Run an IceGrid node in a separate window. Again, the node will not start without appropriate configuration. Place the missing configuration entries into `config.node` and start the node with that configuration file. Your configuration file should enable all node tracing properties, so you can see what the node is doing.
3. Create a deployment descriptor for your server in `filesystem.xml`. The server should be registered for automatic activation and exit if it is idle for more than 20 seconds. Allow the OS to assign a port to the server.
4. Run `icegridadmin` in a separate window. Place the configuration necessary for `icegridadmin` into `config.admin` and start the tool with that configuration file. Use `icegridadmin` to add the deployment descriptor you created to the registry.
5. The server in `Server.java` does not create an object adapter. Add the missing code to create the adapter.

6. Start the server using `icegridadmin`. If everything works, the server should start up and, after 20 seconds, exit again. If your server does not start, examine the trace from the node to work out what is going wrong. (Do not proceed to the next step until you can successfully start the server.)
7. Verify that you can cleanly stop the server using `icegridadmin`.
8. The client requires configuration to bind indirect references. Place the missing configuration for the client into `config.client`.
9. Modify the client source code to specify an indirect reference for the root directory that matches the deployment of your server. Run the client and verify that running the client activates the server and that the client can successfully list the contents of the root directory.
10. Run the client with protocol tracing and examine the messages that are exchanged between the client and the registry.
11. Run the IceGridGUI tool and use it to modify the server's deployment to advertise the root directory as a well-known object with the identity "RootDir". Modify the client source code to use this well-known identity and run the client to verify that it can bind to the server using a well-known proxy with this identity.

17-4 Registry Configuration

Registry Configuration

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.Data=registry
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifier
```

```
IceGrid.Registry.Trace.Locator=2
IceGrid.Registry.Trace.Adapter=3
IceGrid.Registry.Trace.Node=2
IceGrid.Registry.Trace.Server=1
IceGrid.Registry.Trace.Object=1
```



Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.

17-4

Notes:

```
IceGrid.Registry.Client.Endpoints=tcp -p 4061
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.Data=registry
IceGrid.Registry.AdminPermissionsVerifier=IceGrid/NullPermissionsVerifier
```

```
IceGrid.Registry.Trace.Locator=2
IceGrid.Registry.Trace.Adapter=3
IceGrid.Registry.Trace.Node=2
IceGrid.Registry.Trace.Server=1
IceGrid.Registry.Trace.Object=1
```

17-5 Node Configuration

Node Configuration

```
Ice.Default.Locator=IceGrid/Locator:tcp -p 4061
```

```
IceGrid.Node.Endpoints=tcp
```

```
IceGrid.Node.Name=Node
```

```
IceGrid.Node.Data=node
```

```
IceGrid.Node.Trace.Activator=3
```

```
IceGrid.Node.Trace.Adapter=3
```

```
IceGrid.Node.Trace.Server=3
```



Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.

17-5

Notes:

```
Ice.Default.Locator=IceGrid/Locator:tcp -p 4061
```

```
IceGrid.Node.Endpoints=tcp
```

```
IceGrid.Node.Name=Node
```

```
IceGrid.Node.Data=node
```

```
IceGrid.Node.Trace.Activator=3
```

```
IceGrid.Node.Trace.Adapter=3
```

```
IceGrid.Node.Trace.Server=3
```

17-6 Deployment Descriptor

Deployment Descriptor

```
<icegrid>
  <application name="filesystem">
    <node name="Node">
      <server id="fsserver" exe="java" activation="on-demand">
        <adapter name="Lab8" id="fsadapter" endpoints="tcp">
          </adapter>
        <property name="Ice.ServerIdleTime" value="20"/>
        <option>Server</option>
      </server>
    </node>
  </application>
</icegrid>
```



Notes:

17-7 Admin Configuration

Admin Configuration

- `Ice.Default.Locator=IceGrid/Locator:tcp -p 4061`

Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.17-7

Notes:

`Ice.Default.Locator=IceGrid/Locator:tcp -p 4061`

17-8 Server Source Modification

Server Source Modification

The server must call `createObjectAdapter` (instead of `createObjectAdapterWithEndpoints`) to create the adapter:

```
Ice.ObjectAdapter adapter =  
    communicator().createObjectAdapter("Lab8");
```



Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.

17-8

Notes:

The server must call `createObjectAdapter` (instead of `createObjectAdapterWithEndpoints`) to create the adapter:

```
Ice.ObjectAdapter adapter = communicator().createObjectAdapter("Lab8");
```

Client Configuration

- `Ice.Default.Locator=IceGrid/Locator:tcp -p 4061`



Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.

17-9

Notes:

`Ice.Default.Locator=IceGrid/Locator:tcp -p 4061`

17-9 Client Modification

Client Modification

- For step 9, the client needs to use the proxy:
`RootDir@fsadapter.`
- For step 11, the proxy is:
`RootDir.`

Assignment 8 Using IceGrid
Copyright © 2005-2010 ZeroC, Inc.17-10

Notes:

For step 9, the client needs to use the proxy `RootDir@fsadapter.`

For step 11, the proxy is `RootDir.`

18 The Ice Run Time in Detail

18 The Ice Run Time in Detail

18-1 Lesson Overview

Lesson Overview

- This lesson:
 - takes closer look at the Ice run time.
 - explains some advanced implementation techniques that allow you to take precise control of the performance–footprint trade-off for a server.
- By the completion of this chapter, you will know how to build realistic server applications that can scale to millions of objects.

The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.18-1

Notes:

This lesson takes a closer look at the Ice run time. In particular, it explains some advanced implementation techniques that allow you to take precise control of the performance–footprint tradeoff for a server.

18-1-1 Lesson Objectives

By the completion of this lesson, you will know how to build realistic server applications that can scale to millions of objects.

18-2 The Ice::Communicator Interface

The Ice::Communicator Interface

Ice::Communicator is the main handle to the Ice run time.

The communicator is associated with a number of resources:

- Client- and server-side thread pool
- Configuration properties
- Object factories to instantiate Slice classes
- A logger object to redirect warning and error messages
- A statistics objects to collect statistics on traffic volumes
- A default router (used by Glacier2)
- A default locator (used by IceGrid)
- A plug-in manager to load plug-ins (such as IceSSL)
- One or more object adapters

You can have more than one communicator in a process (for example, to use different configuration properties with each).



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-2

Notes:

Ice::Communicator is a local interface that represents the main handle to the Ice run time. Before you can do anything in your code with Ice, you must obtain a handle to a communicator.

A communicator is associated with a number of run-time resources:

- Client-side thread pool
The client-side thread pool is responsible for processing messages received on outgoing connections. Threads in this pool also dispatch callbacks for asynchronous method invocations, and handle incoming requests from bi-directional connections.
- Server-side thread pool
Threads in this pool accept incoming connections and handle requests from clients.
- Configuration properties
Various aspects of the Ice run time can be configured via properties. Each communicator has its own set of such configuration properties (see Chapter 8).
- Object factories

In order to instantiate classes that are derived from known base types, the communicator maintains a set of object factories that can instantiate the classes on behalf of the Ice run time.

- **Logger object**
A logger object implements the `Ice::Logger` interface and determines how log messages that are produced by the Ice run time are handled. (See the Ice manual for details.)
- **Statistics object**
A statistics object implements the `Ice::Stats` interface and is informed about the amount of traffic (bytes sent and received) that is handled by a communicator. (See the Ice manual for details.)
- **Default router**
A router implements the `Ice::Router` interface. Glacier2 is a router that implements the firewall functionality of Ice (see Chapter 14).
- **Default locator**
A locator is an object that resolves an object identity to a proxy. Locator objects are used to implement location services; IceGrid is an example of a service that implements a locator (see Chapter 16).
- **Plug-in manager**
Plug-ins are objects that add features to a communicator. For example, IceSSL is implemented as a plug-in. Each communicator has a plug-in manager that implements the `Ice::PluginManager` interface and provides access to the set of plug-ins for a communicator.
- **Object adapters**
Object adapters dispatch incoming requests and take care of passing each request to the correct servant.
- **Administrative facility**
Ice applications often require remote administration, such as when an IceGrid node needs to gracefully deactivate a running server. The Ice run time provides an extensible, centralized facility for exporting administrative functionality. This facility consists of an object adapter named `Ice.Admin`, an Ice object activated on this adapter, and configuration properties that enable the facility and specify its features.
- **Implicit context**
All proxy invocations support an optional trailing argument of type `Ice::Context` (which is a map of name-value strings). Applications do not typically use this parameter, but for those that do use it, having to pass the same context explicitly in every proxy invocation can be cumbersome. The communicator allows you to establish an implicit context that is sent with all invocations, provided that you do not supply an explicit context with the call.

You can use more than one communicator in a process. Each communicator maintains a separate set of its associated run-time resources, such as thread pools and network connections. In some circumstances it may be necessary to create more than one communicator, for example, in order to use different configuration properties for each. As a general rule, however, a single communicator is sufficient for most applications.

18-3 The Ice::Communicator Interface (cont. 1)

The Ice::Communicator Interface (1)

```
module Ice {
    local interface Communicator {
        string proxyToString(Object* obj);
        Object* stringToProxy(string str);

        ObjectAdapter createObjectAdapter(string name);
        ObjectAdapter createObjectAdapterWithEndpoints(
            string name,
            string endpoints);

        void shutdown();
        void waitForShutdown();
        void destroy();
        // ...
    };
    // ...
};
```



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-3

Notes:

The communicator offers a number of commonly-used operations:

- `proxyToString`
`stringToProxy`

These operations allow you to convert a proxy into its stringified representation and vice versa.

Instead of calling `proxyToString` on the communicator, you can also use the `ice_toString` operation on a proxy to stringify it. However, you can only stringify non-null proxies that way—to stringify a null proxy, you must use `proxyToString`. (The stringified representation of a null proxy is the empty string.)

- `createObjectAdapter`
`createObjectAdapterWithEndpoints`

These operations create a new object adapter. Each object adapter is associated with one or more transport endpoints. Typically, an object adapter has a single transport endpoint. However, an object adapter can also offer multiple endpoints. If so, these endpoints each lead to the same set of objects and represent alternative means of accessing these objects. This is useful, for example, if a server is behind a firewall but must offer access to its objects to both internal and external clients; by binding the adapter to both the internal and external interfaces, the objects implemented in the server can be accessed via either interface.

Whereas `createObjectAdapter` determines which endpoints to bind itself to from configuration information, `createObjectAdapterWithEndpoints` allows you to specify the transport endpoints for the new adapter. Typically, you should use `createObjectAdapter` in preference to `createObjectAdapterWithEndpoints`. Doing so keeps transport-specific information, such as host names and port numbers, out of the source code and allows you to reconfigure the application by changing a property.

- `shutdown`

This operation shuts down the server side of the Ice run time:

- Dispatched operations that are in progress at the time `shutdown` is called are allowed to complete normally. `shutdown` does not wait for these operations to complete; when `shutdown` returns, you know that no new incoming requests will be dispatched, but operations that were already in progress at the time you called `shutdown` may still be running. You can wait for still-executing operations to complete by calling `waitForShutdown`.
- Operation invocations that arrive after the server has called shutdown either fail with a `ConnectFailedException` or are transparently redirected to a new instance of the server (see Chapter 16).

Note that `shutdown` initiates deactivation of all object adapters associated with the communicator, so attempts to use an adapter once shutdown has completed raise an `ObjectAdapterDeactivatedException`.

- `waitForShutdown`

This operation suspends the calling thread until the communicator has shut down (that is, until no more operations are executing in the server). This allows you to wait until the server is idle before you destroy the communicator.

If you call `destroy` without calling `shutdown`, the call waits for all dispatched operations to complete before it returns (that is, the implementation of `destroy` implicitly calls `shutdown` followed by `waitForShutdown`). `shutdown` (and, therefore, `destroy`) deactivate all object adapters that are associated with the communicator.

One the client side, calling `destroy` while outgoing invocations are still pending causes those invocations to terminate immediately with a `CommunicatorDestroyedException`.

18-4 Creating a Communicator

Creating a Communicator

```
final class InitializationData implements Cloneable
{
    public java.lang.Object clone();
    public Properties properties;
    public Logger logger;
    public Stats stats;
    public ThreadNotification threadHook;
    public ClassLoader classLoader;
    public Dispatcher dispatcher;
}
static Communicator initialize();
static Communicator initialize(String[] args);
static Communicator initialize(StringSeqHolder ah);
static Communicator initialize(InitializationData id);
static Communicator initialize(String[] args,
                               InitializationData id);
static Communicator initialize(StringSeqHolder ah,
                               InitializationData id);
```



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-4

Notes:

The various initialization functions create a new communicator. Note that the methods are overloaded. The alternative versions accept a `StringSeqHolder` instead of an array and remove any Ice-specific options from the passed sequence so, once the initialization function returns, the sequence contains only application-specific options. This saves you having to write code that skips over Ice-related options when parsing the command line.

The `InitializationData` parameter allows you to control various aspects of the communicator, such as its property set. (If you want to prevent command-line options from overriding property settings, you can pass a dummy argument vector.)

18-5 Object Adapters

Object Adapters

Object adapters link the server-side run time to the server-side application code.

Each server has at least one object adapter.

Each object adapter provides one or more transport endpoints at which it listens for incoming requests.

Each object adapter provides an Active Servant Map to dispatch incoming requests.

Operations that manipulate the ASM:

`Object* add(Object servant, Identity Id)`

`Object* addWithUUID(Object servant)`

`Object remove(Identity Id)`

`Idempotent Object find(Identity Id)`



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-5

Notes:

As we saw in Chapter 1, object adapters provide the link between the Ice run time and the server-side application code. The main responsibilities of an object adapter are to provide one or more transport endpoints to which clients can send requests, and to maintain the mapping between object identities and servants, that is, to maintain the ASM.

You can add an entry to the ASM by calling `add` or `addWithUUID`. (The former requires you to provide an object identity for the servant, whereas the latter generates a unique object identity for each servant that is added.)

The `remove` operation removes the specified entry from the ASM, and the `find` operation returns the servant for a given identity (or null if no such entry exists).

18-6 Servant Locators

Servant Locators

By default, if the identity for an incoming request cannot be found in the ASM, the run time returns `ObjectNotExistException` to the client. You can register one or more servant locators with an object adapter. The job of a servant locator is to locate or create a servant for a request.

```
local interface ServantLocator
{
    Object locate(Current curr,
                 out LocalObject cookie);
    void finished(Current curr,
                 Object servant,
                 LocalObject cookie);

    void deactivate(string category);
};
```



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-6

Notes:

A servant locator is an object that you register with an object adapter. If an incoming request fails to find a servant in the ASM, instead of throwing `ObjectNotExistException`, the adapter calls the locator's `locate` operation. If `locate` returns a servant, the request is dispatched to the returned servant and, as soon as the request completes, the adapter calls `finished`. The servant that is returned by `locate` is *not* added to the ASM. This means that the association between the returned servant and the incoming request is valid only for the duration of a single request.

The `finished` operation allows you to perform clean-up work once the request completes. For example, inside `finished`, you can deallocate a resource that was allocated by `locate`.

The `cookie` out-parameter of `locate` is passed to the corresponding call to `finished`. This allows you to pass state between `locate` and `finished`.

If `locate` cannot procure a servant (typically, because the object identity does not correspond to an existing Ice object), it can return null. A null return value indicates to the object adapter that no servant could be found; in response, the object adapter marshals an `ObjectNotExistException` back to the client. In addition, if `locate` throws an exception, that exception is passed back to the client. (However, remember that exceptions other than `ObjectNotExistException`, `FacetNotExistException`, and `OperationNotExistException` appear as `UnknownLocalException` in the client.)

The Ice run time calls `deactivate` when a servant locator is no longer needed, that is, when you destroy the object adapter or destroy the adapter's communicator. `deactivate` allows you to perform final clean-up operations (such as closing a database connection).

18-7 Threading Guarantees for Servant Locators

Threading Guarantees for Servant Locator

Guarantees provided by the Ice run time:

- Every call to `locate` is balanced by a call to `finished`.
- `locate`, the servant operation, and `finished` are called by the same thread. (When using AMD, `finished` may be called by a different thread.)
- No call to `locate` or `finished` can arrive after `deactivate` is called, and `deactivate` is not called concurrently with `locate` or `finished`.

Note that:

- Multiple calls to `locate` can proceed concurrently.
- Multiple calls to `finished` can proceed concurrently.
- `locate` and `finished` can proceed concurrently.

Concurrency can involve the same object ID!



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-7

Notes:

The Ice run time guarantees that every call to `locate` is balanced by a corresponding call to `finished`. (Naturally, this is true only if `locate` indeed returns a servant; if `locate` returns null or throws an exception, `finished` is not called.)

`locate`, the operation, and `finished` are guaranteed to be called by the same thread.

This allows you to, for example, acquire a lock in `locate` and release that lock in `finished`. (If you are using Asynchronous Method Dispatch (AMD), this guarantee does not hold: `finished` may be called by a different thread, but `locate` and the operation are still guaranteed to be called by the same thread.)

`deactivate` is guaranteed to be the last call on a servant locator, and does not start until all outstanding calls to `finished` have completed.

Beyond that, no guarantees are provided. In particular, it is possible for multiple calls to `locate` and `finished` to proceed concurrently (for the same or for different object identities).

18-8 Servant Locator Registration

Servant Locator Registration

```
local interface ObjectAdapter {
    void addServantLocator(ServantLocator locator,
                          string category);

    ServantLocator findServantLocator(string category);

    // ...
};
```

Note that a servant locator is registered for a specific category.

- If the target identity of an incoming request has a matching category, the run time calls the corresponding servant locator.
- Otherwise, if you have servant locator with an empty category, the run time calls that servant locator (known as the *default locator*).



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-8

Notes:

Recall the definition of an object identity:

```
struct Identity {
    string name;
    string category;
};
```

As you can see, each object identity consists of two fields, the **name** and the **category**. A servant **locator** is registered for a specific category. For an incoming request to trigger a locator's **locate** operation, the category in the incoming request must match the servant locator's category.

However, you can also register a servant locator with the empty category. That locator is known as the *default servant locator*, and its **locate** operation is called for all requests that either have an empty category, or have a non-empty category for which no specific servant locator is registered.

18-9 Call Dispatch Rules

Call Dispatch Rules

1. Look for the identity in the ASM. If the ASM contains an entry, dispatch the request. Finished.
2. If the category of the request is non-empty, look for a matching servant locator.
 - If a matching locator is found, call its `locate` operation. If `locate` returns a servant, dispatch the request; otherwise, throw `ObjectNotExistException`. Finished.
 - If no matching locator is found, continue with Step 3.
3. Look for a default servant locator.
 - If a default servant locator is found, call its `locate` operation. If `locate` returns a servant, dispatch the request; otherwise, throw `ObjectNotExistException`. Finished.
 - If no default locator is found, continue with Step 4.
4. Raise `ObjectNotExistException` in the client.



Notes:

The above slide summarizes the rules that are used to dispatch calls. The run time first looks for an entry in the ASM and, if one exists, dispatches the call to the corresponding servant.

Otherwise, if the request has a non-empty category *and* a servant `locator` is registered for that category, that servant locator's `locate` operation is called and determines the outcome of the request.

Otherwise, if a default servant locator is registered and the request category is empty or the category is non-empty but no specific servant locator is registered for that category, the default locator's `locate` operation is called and determines the outcome of the request.

Otherwise, no servant could be found, either in the ASM or by a servant locator, so the run time raises `ObjectNotExistException` in the client.

18-10 Implementing Servant Locators

Implementing Servant Locators

Each servant locator must be derived from the `Ice::ServantLocator` base class:

```
public class MyServantLocator implements Ice.ServantLocator
{
    public Ice.Object
    locate(Ice.Current c, Ice.LocalObjectHolder cookie);

    public void
    finished(Ice.Current c,
            Ice.Object servant,
            Object cookie);

    public void
    deactivate(String category);
}
```



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-10

Notes:

To implement a servant locator, you must define a class that implements `Ice.ServantLocator` and supply implementations of the `locate`, `finished`, and `deactivate` methods. (Frequently, the `finished` and `deactivate` methods are empty.)

18-11 Implementing locate

Implementing locate

```
public Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    MyServantDetails d = null;
    try {
        d = DB_lookup(c.id.name);
    } catch (DB_error e)
        return null;
    }
    return new MyInterfaceI(d);
}
```

This implementation locates the state for a servant in a database.

Note that, for each request, a new servant is created.

- Depending on the relative costs of operations and initialization, this may be inefficient.
- Without interlocks, this can result in multiple servants for the same Ice object, if requests arrive concurrently.



Notes:

The above code shows a very simple implementation of a servant locator for an interface **MyInterface**.

We assume that the state for the servant is stored in a database. **DB_lookup** is a database lookup function to retrieve that state. Note that we use the name field of the object identity as the lookup key.¹ If **DB_lookup** cannot find a servant, **locate** returns null, which raises **ObjectNotExistException** in the client. Otherwise, the code instantiates a new servant of type **MyInterfaceI** (passing the servant state to the constructor) and returns the servant.

This implementation, in which a new servant is instantiated for each and every request, is probably too simplistic for a real application. The potential exists that concurrent invocations by clients for the same object identity will result in more than one instantiated servant in memory for the same Ice object. This may be OK (depending on exactly what the operations on the servant do), or it may spell disaster. If you want to avoid creating multiple servants for the same Ice object, you must keep a list of existing servants and do interlocked lookups into that list in **locate**.

¹ This is a common pattern when making objects persistent: the identity serves as the database key for the remainder of the servant state.

However, the implementation illustrates the basic principle: the job of `locate` is to come up with a servant for a request. How it does that is up to each application. (For example, `locate` may return an already-instantiated servant instead of creating a new one.)

18-12 Information Provided to locate

Information Provided to locate

`locate` is passed the `Ice::Current` object for the incoming request.

`Ice::Current` contains the identity for the incoming request, and the operation name.

Typically, this is all the information you need to locate the correct servant for a request.

Note that `locate` must usually instantiate a servant, but the type of the servant's interface is *not* part of the `Current` object.

If you have locators for servants with different interfaces, you must register a separate locator for each interface type.

The category can be any identifier you choose; you can use the type ID of the servant's interface, or any other suitable identifier.



Notes:

If you have more than one type of object, it is no longer implicit as to which type of servant should be instantiated by `locate`. This is the purpose of having category-specific servant locators: for each type of object, you register a different servant locator, each with its own category identifier.

By using corresponding category values when you create proxies, this ensures that each type of object is serviced by its own servant locator. For example:

```
public class DirectoryLocator implements Ice.ServantLocator
{
    // ...
}
public class FileLocator implements Ice.ServantLocator
{
    // ...
}
// ...

adapter.addServantLocator(new DirectoryLocator(), "d");
adapter.addServantLocator(new FileLocator(), "f");
```

In this example, we used the letters **d** and **f** as the category identifiers. However, you can use any identifier you like—one obvious choice is to use the type ID of the corresponding interface.

18-13 Lazy Initialization

Lazy Initialization

```
Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
{
    MyServantDetails d = null;
    try {
        d = DB_Lookup(c.id.name);
    } catch (DB_error e)
        return null;
    }
    myInterfaceI servant = new MyInterfaceI(d);
    try {
        c.adapter.add(servant, c.id);
    } catch (Ice.AlreadyRegisteredException ex)
        return c.adapter.find(c.id);
    }
    return servant;
}
```

The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc. 18-13

Notes:

For simple servers, a common strategy is to, on start-up, instantiate one servant for each Ice object, add each servant to the ASM, and then activate the object adapter. This strategy is perfectly viable, provided that the server has sufficient memory to keep a servant for each Ice object permanently in memory, and that the time required to initialize all the servants on start-up is acceptable.

One way to mitigate this issue is to use lazy initialization: instead of instantiating all servants upfront, you can instantiate them on an as-needed basis using a servant locator, as shown above. With this strategy, the ASM is initially empty, and servants are created on demand and added to the ASM as clients access the corresponding Ice objects.

The main advantages of this approach are:

- Initialization cost is spread out over many invocations instead of being incurred all at once during server start-up.
- Servants are instantiated only for those Ice objects that are actually accessed by clients.

In general, incremental initialization is beneficial if instantiating servants during start-up is too slow. The memory savings can be worthwhile as well but, as a rule, are realized only for comparatively short-lived servers: for long-running servers, chances are that, sooner or later, every Ice object will be accessed by some client or another; in that case, there are no memory savings because we end up with an instantiated servant for every Ice object regardless.

18-14 Creating Proxies

Creating Proxies

You can create a proxy for an Ice object without instantiating a servant for that object:

```
local interface ObjectAdapter {
    Object* createProxy(Identity id);
    // ...
};
```

This is more efficient than instantiating a servant and adding it to the ASM in order to obtain its proxy. `createProxy` is particularly useful for `list` operations that return a large number of proxies to clients. When used in combination with servant locators, this avoids having to instantiate a servant for each Ice object returned in a list.

The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.18-14

Notes:

For the previous technique to be effective, it must be possible to pass a proxy for an Ice object to clients without having to first instantiate a servant in order to obtain the proxy.

The `createProxy` operation on the object adapter allows you to do this. It creates a proxy for an Ice object given its identity, so there is no need to first instantiate a servant and to add it to the ASM.

`createProxy` is particularly useful for `list` operations that return large numbers of proxies. Instead of instantiating a servant for each Ice object in the returned list, you can create the proxies for the Ice objects to be returned with `createProxy` and rely on a servant locator to instantiate those servants only if a client actually uses them.

`createProxy` takes the endpoint information that is part of every proxy from the object adapter's configuration, that is, `createProxy` automatically creates a proxy with the correct binding information, regardless of whether the adapter uses direct or indirect binding (see Chapter 16).²

² Note that the adapter also provides operations to specifically create direct and indirect proxies. See the Ice manual for details.

18-15 Default Servants

Default Servants

A servant that implements many different Ice objects simultaneously is called a *default servant*.

Default servants are useful for servers that act as a front end to backend storage, such as servers that sit in front of a database and present database records as Ice objects.

Ice provides an API similar to servant locators that makes it easy to register your default servants.

Each operation implementation uses the object identity for the request to determine which servant state to operate on.

Default servants allow unlimited scalability with very small memory footprint.



Notes:

A common requirement is to make already existing data available as Ice objects to clients. Usually, the data is present in a database, or can be retrieved via a custom API from an already-existing application. In such cases, the Ice server merely acts as a front end to the existing data, as a facade.

A good way to implement such servers is to use *default servants*. A default servant is a regular servant that you register for a specific category with the object adapter, much like servant locators and using a very similar API. (In fact, another way to implement a default servant is using a servant locator that unconditionally returns the same single servant from its `locate` operation. Prior to Ice 3.4, this was the *only* way to implement a default servant, but the use case was so common that an explicit API was added.)

Default servants differ from servant locators in that there is no `locate` hook in which the application is given an opportunity to examine the `Current` object and select a servant to dispatch the current request. Rather, the object adapter dispatches *all* requests for a category to its registered default servant unless a servant is found in the ASM matching the identity of the current request. The implication here is that the association between an identity category and an interface type becomes critical: if you have several interface types, you need one default servant for each type.

Each operation implementation in a default servant uses the object identity that is delivered to it in the `Current` object for the request to determine the target object of the request, and to adjust its behavior accordingly.

Registering default servants is as easy as registering servant locators. Assuming that we have default servants for directories and files, we can register them with the object adapter as follows:

```
_adapter.addDefaultServant(new DirectoryI(), "d");  
_adapter.addDefaultServant(new FileI(), "f");
```

Again, for this example, we have used `d` and `f` as the category to distinguish directories and files, but we could have used other identifiers, such as `Filesystem::Directory` and `Filesystem::File` instead.

18-16 Default Servants (cont. 1)

Default Servants (1)

With a default servant, the implementation of each operation:

- uses the **Current** object to get the object identity
- uses the **name** member of the identity to locate the state of the Ice object (for example, by using it as the key of a database table). If no state can be found for the identity, the operation throws **ObjectNotExistException**.
- Implements the operation to operate on the retrieved state.

This makes the server completely stateless. Each operation retrieves the state, operates on it, and forgets the state again.



Notes:

Here is a pseudo-code implementation of a **write** operation in the default servant for a file:

```
void
write(String[] l, Ice.Current c)
{
    DBRecord file = null;

    // Locate the file using its identity.
    //
    try {
        file = DB_getRecord(c.id.name);
    } catch (DB_error ex)
        throw new Ice.ObjectNotExistException();
    }

    // Update the file.
    //
    updateRecord(file, l); // Helper function to update file
```

```
}
```

Exactly how the operation goes about implementing the correct functionality is irrelevant. The important point is that the object identity is used as a key to retrieve the servant state, so the same single operation can take on different personas depending on which Ice object is the target of the request.

Note that this also allows you to implement Ice objects as lumps of data; for example, the server could keep a map in memory that, for each identity, stores a structure with the servant state. Alternatively, the operation implementation could access a physical hardware device in order to implement the operation, or use a network protocol such as SNMP to access the state of a device in an internal network.

18-17 Default Servants (cont. 2)

Default Servants (2)

If you use default servants, you should override the `ice_ping` operation on the skeleton to do the right thing.

The inherited default implementation always succeeds. However, if you use a default servant, a client may ping an Ice object that truly does not exist.

```
void  
ice_ping(Ice.Current c)  
{  
    try {  
        DB_lookup(c.id.name);  
    } catch (DB_error ex)  
        throw new Ice.ObjectNotExistException();  
}
```

You need to override `ice_ping` only if clients actually use it (but it is good practice to do so).



Notes:

One thing to keep in mind when using default servants is that the default implementation of `ice_ping` that the servant inherits from its skeleton always succeeds. If you use the ASM, this is just what is needed but, for default servants, this makes it possible for a client to successfully call `ice_ping` on a destroyed object.

To ensure that `ice_ping` works as intended, you should override it. The implementation must check whether the identity for the request denotes an existing object and, if not, throw `ObjectNotExistException`.

18-18 Hybrid Approaches and Caching

Hybrid Approaches and Caching

You can combine the ASM and a default servant:

- Put performance-critical servants that are accessed frequently into the ASM.
The implementation of these servants should cache all servant state in memory to get good performance.
- Use a default servant for less frequently-accessed servants.
The implementation of these servants retrieves state on demand from back-end storage, to keep memory consumption low.

This approach is useful if the access patterns to servants are known in advance and static.



Notes:

Depending on the nature of your application, you may be able to steer a middle path that provides better performance while keeping memory requirements low: if your application has a number of frequently-accessed objects that are performance-critical, you can add servants for those objects to the ASM. If you store the state of these objects in data members inside the servants, you effectively have a cache of these objects.

The remaining, less frequently-accessed objects can be implemented with a default servant. For example, in our file system implementation, we could choose to instantiate directory servants permanently, but to have file objects implemented with a default servant. This provides efficient navigation through the directory tree and incurs slower performance only for the (presumably less frequent) file accesses.

This technique could be augmented with a cache of recently-accessed files, along similar lines to the buffer pool used by the UNIX kernel. The point is that you can combine use of the ASM with servant locators and default servants to precisely control the trade-offs among scalability, memory consumption, and performance to suit the needs of your application.

Note that this approach is useful only if you have advance knowledge of the access patterns to servants. If it is impossible to statically predict which servants will be most frequently accessed, it is better to use an evictor.

18-19 Evictors

Evictors

An *evictor* is a servant locator that instantiates servants up to some predefined maximum number of instances:

- If a request arrives for a servant that is not yet in memory, the servant locator instantiates a new servant and returns it, provided that the limit of servants is not exceeded.
- If a request arrives for a servant that is already in memory, the servant locator returns that servant.
- If a request arrives for a servant that is not in memory, and the number of servants is already at the limit, the servant locator destroys an existing servant and instantiates a new one.

The servant that is evicted is the least-recently-used servant.

Evictors allow control of the footprint–performance trade-off in a server.

By choosing the evictor size appropriately, you get good performance for the most frequently-used servants, with acceptable memory consumption.



Notes:

18-21 Evictors (cont. 2)

Evictors (2)

Same evictor after a client accesses servant 3:

The evictor has dequeued the entry for servant 3 and placed it at the head of the evictor queue, making servant 3 the most recently-used servant.

 The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc. 18-21

Notes:

18-22 Evictors (cont. 3)

Evictors (3)

Same evictor after a client accesses servant 6:

The least recently-used servant (servant 1) has been removed from the evictor and is destroyed once it no longer services a request.

 The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc. 18-22

Notes:

18-23 Evictor Implementation

Evictor Implementation

Implementation goals:

- Reusable, so it can be used for any type of servant.
- Non-intrusive to servant implementation. (Servant implementation should not know about evictor.)
- High performance for both locating a servant and evicting a servant.
- Easily configurable evictor size.

Basic implementation:

- Use a map to store identity–servant pairs for quick lookup.
- Use a queue to maintain LRU order. Queue entries point at map entries. Enqueuing, dequeuing, and maintaining LRU order can be performed in constant time.
- Implementation is inherited from a base class.



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-23

Notes:

Our evictor implementation is designed as an abstract base class. The idea is to have the base class do all the work of maintaining the evictor queue; the derived class needs to be involved only when a servant needs to be instantiated (and, possibly, when a servant is evicted). The basic design then looks as follows:

```
public abstract class EvictorBase implements Ice.ServantLocator
{
    public EvictorBase(int size);

    synchronized public final Ice.Object
    locate(Ice.Current c, Ice.LocalObjectHolder cookie);

    synchronized public final void
    finished(Ice.Current c, Ice.Object o, Object cookie);

    synchronized public final void
    deactivate(String category);

    public abstract Ice.Object
```

```
    add(Ice.Current c, Ice.LocalObjectHolder cookie);

    public abstract void
    evict(Ice.Object servant, Object cookie);

    // ...
}
```

Note that the **EvictorBase** class implements its own **locate**, **finished**, and **deactivate** methods. (These methods do the work of maintaining the evictor queue.)

The constructor for the class accepts the size of the evictor. (Of course, you could easily modify the implementation to retrieve the size from properties instead.)

The class also defines two abstract methods that you must implement in the derived class:

- **add**

The **EvictorBase** class calls this method when a client request arrives for which no servant is in memory. The implementation of **add** is therefore the same as an implementation of **locate**: the method simply instantiates and returns a servant (without adding that servant to the ASM).

Note that you can return a cookie from **add**. If you do, that same cookie is passed to **evict** when the corresponding servant is evicted. This allows you to pass additional information from **add** to **evict** for each servant.

- **evict**

This method is called when the evictor evicts a servant. The **cookie** parameter is the same object that was returned from the corresponding call to **add**.

Unless you have special cleanup requirements, this method is usually empty.

18-24 Evictor Implementation (cont. 1)

Evictor Implementation (1)

The private part of `EvictorBase`:

- stores the cookie that is returned from `add` (so it can be passed to `evict`) in a map,
- stores an iterator into the evictor queue that marks the position of the servant in the queue,
- stores a use count for each servant that is incremented when an operation is dispatched, and decremented when an operation completes.



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-24

Notes:

The private part of `EvictorBase` obviously must store the cookie that is returned from `add`, so it can pass the same cookie to the corresponding call to `evict`.

In addition, the map stores an iterator into the evictor queue for each entry. This is necessary so we can efficiently maintain the queue in LRU order: the iterator allows the entry for a servant in the queue to be located in constant time (instead of having to perform a linear search on the queue).

Finally, for each servant, the evictor maintains a use count that counts the number of active operation invocations in that servant. We need the use count to correctly deal with long-running operations.

Suppose a client invokes a long-running operation on an Ice object with identity *I*. In response, the evictor adds a servant for *I* to the evictor queue. While the original invocation is still executing, other clients invoke operations on various Ice objects, which leads to more servants for other object identities being added to the queue. As a result, the servant for identity *I* gradually migrates toward the tail of the queue. If enough client requests for other Ice objects arrive while the operation on object *I* is still executing, the servant for *I* could be evicted while it is still executing the original request.

By itself, this will not do any harm. However, if the servant is evicted and a client then invokes another request on object *I*, the evictor would have no idea that a servant for *I* is still around and would add a second servant for *I*. However, having two servants for the same Ice object in memory is likely to cause problems, especially if the servant's operation implementations write to a database.

The use count allows us to avoid this problem: we keep track of how many requests are currently executing inside each servant and, while a servant is busy, avoid evicting that servant. As a result, the queue size is not a hard upper limit: long-running operations can temporarily cause more servants than the limit to appear in the queue. However, as soon as excess servants become idle, they are evicted as usual.

Note that the `LinkedList` class that is used here is a special version that does not invalidate iterators during deletion. (You can find the implementation of this class in the Ice distribution in `demo/book/evictor`.)

```
public abstract class EvictorBase implements Ice.ServantLocator
{
    // ...

    private class EvictorEntry
    {
        Ice.Object servant;
        Object userCookie;
        java.util.Iterator<Ice.Identity> pos;
        int useCount;
    }

    private class EvictorCookie
    {
        public EvictorEntry entry;
    }

    private java.util.Map<Ice.Identity, EvictorEntry> _map =
        new java.util.HashMap<Ice.Identity, EvictorEntry>();
    private LinkedList<Ice.Identity> _queue =
        new LinkedList<Ice.Identity>(); // Special list
    private int _size;
}
```

The constructor implementation is trivial: it simply initializes the size of the evictor queue:

```
public
EvictorBase(int size)
{
    _size = size < 0 ? 1000 : size;
}
```

The meat of the implementation is in `locate`:

```
synchronized public final Ice.Object
locate(Ice.Current c, Ice.LocalObjectHolder cookie)
```

```
{
    //
    // Create a cookie.
    //
    EvictorCookie ec = new EvictorCookie();
    cookie.value = ec;

    //
    // Check if we already have a servant in the map.
    //
    ec.entry = _map.get(c.id);
    if(ec.entry != null)
    {
        //
        // Got an entry already, dequeue the entry from
        // its current position.
        //
        ec.entry.pos.remove();
    }
    else
    {
        //
        // We do not have entry. Ask the derived class to
        // instantiate a servant and add a new entry to the map}
        //
        ec.entry = new EvictorEntry();
        Ice.LocalObjectHolder cookieHolder =
            new Ice.LocalObjectHolder();
        ec.entry.servant = add(c, cookieHolder); // Down-call
        if(ec.entry.servant == null)
        {
            return null;
        }
        ec.entry.userCookie = cookieHolder.value;
        ec.entry.useCount = 0;
        _map.put(c.id, ec.entry);
    }

    //
    // Increment the use count of the servant and enqueue
    // the entry at the front, so we get LRU order.
    //
    ++(ec.entry.useCount);
    _queue.addFirst(c.id);
    ec.entry.pos = _queue.iterator();
}
```

```

        ec.entry.pos.next(); // Position the iterator on the element.

        return ec.entry.servant;
    }

```

Note that the method is synchronized to protect the evictor's data structures from concurrent access. The first step is to create the cookie that is returned from `locate` and will be passed by the Ice run time to the corresponding call to `finished`. The cookie contains a reference to an evictor entry, of type `EvictorEntry`. This is also the value type of our map entries, so we do not store two copies of the same information redundantly.

The next step is to look in the evictor map to see whether we already have an entry for this object identity. If so, we initialize the cookie's value with that entry and remove the entry from its current queue position.

Otherwise, we do not have an entry for this object identity yet, so we have to create one. The code creates a new evictor entry, and then calls `add` to get a new servant. This is a down-call to the concrete class that will be derived from `EvictorBase`. The implementation of `add` must attempt to locate the object state for the Ice object with the identity passed inside the `Current` object and either return a servant as usual, or return null or throw an exception to indicate failure. If `add` returns null, we return null to let the Ice run time know that no servant could be found for the current request. If `add` succeeds, we initialize the entry's use count to zero and insert the entry into the evictor map.

The last few lines of `locate` add the entry for the current request to the head of the evictor queue to maintain its LRU property, increment the use count of the entry, and finally return the servant to the Ice run time.

The implementation of `finished` is much simpler: it decrements the use count and calls the private method `evictServants` to get rid of any servants that need to be evicted:

```

synchronized public final void
finished(Ice.Current c, Ice.Object o, Object cookie)
{
    EvictorCookie ec = (EvictorCookie)cookie;
    //
    // Decrement use count and check if
    // there is something to evict.
    //
    --(ec.entry.useCount);
    evictServants();
}

```

`evictServants` scans the evictor queue for elements in excess of the evictor's size: any excess entries with a zero use count are evicted by calling the derived class's `evict` method:

```

private void evictServants()
{
    //
    // If the evictor queue has grown larger than the limit,
    // look at the excess elements to see whether any of them
    // can be evicted.

```

```
//
for(int i = _map.size() - _size; i > 0; --i)
{
    java.util.Iterator<Ice.Identity> p = _queue.riterator();
    Ice.Identity id = p.next();
    EvictorEntry e = _map.get(id);
    if(e.useCount == 0)
    {
        evict(e.servant, e.userCookie); // Down-call
        p.remove();
        _map.remove(id);
    }
}
}
```

Finally, the **deactivate** implementation cleans up by setting the evictor size to zero and then calling **evictServants**. This causes all servants on the queue to be evicted. (It is guaranteed that the use count of all the servants will be zero because the Ice run time does not call deactivate until no more requests are executing inside the object adapter.)

```
synchronized public final void
deactivate(String category)
{
    _size = 0;
    evictServants();
}
```

18-25 Using EvictorBase

Using EvictorBase

```
public class MyInterfaceEvictor extends EvictorBase
{
    public Ice.Object
    add(Ice.Current c, Ice.LocalObjectHolder cookie)
    {
        MyServantDetails d = null;
        try {
            d = DB_lookup(c.id.name);
        } catch (DB_error ex)
            return null;
        }
        return new MyInterface(d);
    }

    public void
    evict(Ice.Object servant, Object cookie)
    {
    }
}
```



The Ice Run Time in Detail
Copyright © 2005-2010 ZeroC, Inc.

18-25

Notes:

Creating a concrete evictor could not be simpler: we simply move the code of our original `locate` implementation in Section 18-11 into `add` instead. Using the evictor now becomes a matter of instantiating the class and registering it as a servant locator with the object adapter:

```
EvictorBase = new MyInterfaceEvictor();
```

```
adapter.addServantLocator(ev, "");
```

19 License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

- a. "**Collective Work**" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with one or more other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.
- b. "**Creative Commons Compatible License**" means a license that is listed at <http://creativecommons.org/compatiblelicenses> that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of derivatives of works made available under that license under this License or either a Creative Commons unported license or a Creative Commons jurisdiction license with the same License Elements as this License.
- c. "**Derivative Work**" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.
- d. "**License Elements**" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.
- e. "**Licensor**" means the individual, individuals, entity or entities that offers the Work under the terms of this License.
- f. "**Original Author**" means the individual, individuals, entity or entities who created the Work.

- g. "**Work**" means the copyrightable work of authorship offered under the terms of this License.
 - h. "**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
2. **Fair Use Rights.** Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.
3. **License Grant.** Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:
- a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;
 - b. to create and reproduce Derivative Works provided that any such Derivative Work, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";
 - c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;
 - d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.
 - e. For the avoidance of doubt, where the Work is a musical composition:
 - i. **Performance Royalties Under Blanket Licenses.** Licensor waives the exclusive right to collect, whether individually or, in the event that Licensor is a member of a performance rights society (e.g. ASCAP, BMI, SESAC), via that society, royalties for the public performance or public digital performance (e.g. webcast) of the Work.
 - ii. **Mechanical Rights and Statutory Royalties.** Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).
 - f. **Webcasting Rights and Statutory Royalties.** For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. **Restrictions.** The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

- a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of a recipient of the Work to exercise of the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. When You distribute, publicly display, publicly perform, or publicly digitally perform the Work, You may not impose any technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise of the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by Section 4(c), as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any credit as required by Section 4(c), as requested.

- b. You may distribute, publicly display, publicly perform, or publicly digitally perform a Derivative Work only under: (i) the terms of this License; (ii) a later version of this License with the same License Elements as this License; (iii) either the Creative Commons (Unported) license or a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g. Attribution-ShareAlike 3.0 (Unported)); (iv) a Creative Commons Compatible License. If you license the Derivative Work under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Derivative Work under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and with the following provisions: (I) You must include a copy of, or the Uniform Resource Identifier for, the Applicable License with every copy or phonorecord of each Derivative Work You distribute, publicly display, publicly perform, or publicly digitally perform; (II) You may not offer or impose any terms on the Derivative Works that restrict the terms of the Applicable License or the ability of a recipient of the Work to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties; and, (IV) when You distribute, publicly display, publicly perform, or publicly digitally perform the Work, You may not impose any technological measures on the Derivative Work that restrict the ability of a recipient of the Derivative Work from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Derivative Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Derivative Work itself to be made subject to the terms of the Applicable License.

- c. If You distribute, publicly display, publicly perform, or publicly digitally perform the Work (as defined in Section 1 above) or any Derivative Works (as defined in Section 1 above) or Collective Works (as defined in Section 1 above), You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, consistent with Section 3(b) in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear, if a credit for all contributing authors of the Derivative Work or Collective Work appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND ONLY TO THE EXTENT OF ANY RIGHTS HELD IN THE LICENSED WORK BY THE LICENSOR. THE LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MARKETABILITY, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

- a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works or Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
- b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

- a. Each time You distribute or publicly digitally perform the Work (as defined in Section 1 above) or a Collective Work (as defined in Section 1 above), the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
- b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.
- c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
- d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
- e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.