# Harlequin Common Lisp ORB™

# Developing Component Software with CORBA®

Version 1.0



#### Copyright and Trademarks

Harlequin Common Lisp ORB™: Developing Component Software with CORBA®

Version 1.0 January 1999

Part number: 3LBDT2K15LT

Copyright © 1999 by Harlequin Group plc.

Companies, names and data used in examples herein are fictitious unless otherwise noted.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Harlequin Group plc.

The information in this publication is provided for information only and is subject to change without notice. Harlequin Group plc and its affiliates assume no responsibility or liability for any loss or damage that may arise from the use of any information in this publication. The software described in this book is furnished under license and may only be used or copied in accordance with the terms of that license.

LispWorks is a registered trademark of Harlequin Group plc. Harlequin, Delivery, Transducer/PC, The Authoring Book, ClassWorks, and KnowledgeWorks are all trademarks of Harlequin Group plc.

Liquid Common Lisp is a trademark of The Harlequin Group plc.

Adobe and PostScript are registered trademarks of Adobe Systems Incorporated. Other brand or product names are the registered trademarks or trademarks of their respective holders.

The code for walker.lisp and compute-combination-points is excerpted with permission from PCL, Copyright © 1985, 1986, 1987, 1988 Xerox Corporation.

CLX and CLUE bear the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Texas Instruments Incorporated, P.O. Box 149149, MS 2151, Austin, Texas 78714-9149

Copyright © 1987, 1988, 1989, 1990, 1991 Texas Instruments Incorporated.

Permission is granted to any individual or institution to use, modify and distribute this software, provided that this complete copyright and permission notice is maintained, intact, in all copies and documentation. Texas Instruments Incorporated provides this software "as is" without express or implied warranty.

The XP Pretty Printer bears the following copyright notice, which applies to the parts of LispWorks derived therefrom:

Copyright © 1989 by the Massachusetts Institute of Technology, Cambridge, Massachusetts.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representation about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. M.I.T. disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall M.I.T. be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

CORBA and Object Management Group are registered trademarks, and OMG, ORB, Object Request Broker, IIOP, and OMG Interface Definition Language (IDL) are trademarks, of Object Management Group, Inc. OMG marks and logos are trademarks or registered trademarks, service marks and/or certification marks of Object Management Group, Inc. registered in the United States.

#### US Government Use

The LispWorks Software is a computer software program developed at private expense and is subject to the following Restricted Rights Legend: "Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in (i) FAR 52.227-14 Alt III or (ii) FAR 52.227-19, as applicable. Use by agencies of the Department of Defense (DOD) is subject to Harlequin's customary commercial license as contained in the accompanying license agreement, in accordance with DFAR 227.7202-1(a). For purposes of the FAR, the Software shall be deemed to be 'unpublished' and licensed with disclosure prohibitions, rights reserved under the copyright laws of the United States. Harlequin Incorporated, One Cambridge Center, Cambridge, Massachusetts 02142."

http://www.harlequin.com/

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
UK

telephone +44 1223 873 800 fax +44 1223 873 873

North America:

Harlequin Incorporated One Cambridge Center Cambridge, MA 02142 USA

telephone +1 617 374 2400 fax +1 617 252 6505

Asia Pacific:

Harlequin Australia Pty. Ltd. Level 12 12 Moore Street Canberra, ACT 2601 Australia

telephone +61 2 6206 5522 fax +61 2 6206 5525

# **Contents**

# Preface v 1 Common Lisp and CORBA 9 About CORBA 9 About the Harlequin Common Lisp ORB 10 Features of the Harlequin Common Lisp ORB 12 CORBA examples 12 2 **Quick Start Tutorial** 13 A CORBA-based Hello World 13 Complete source code for the Hello World example 20 3 **Setting up the Bank Example** About the bank example 23 Where to find the example code 24 Building the bank client and server 24 Running the server and client 25 Writing and Compiling IDL 27 4

Compiling IDL for a CORBA application 31

Mapping IDL to Common Lisp 32

Designing the IDL 27

#### 5 The Bank Client 35

The client 35
The client's perspective 35
Implementing the bank client's GUI 36
Defining the interfaces 37

#### 6 The Bank Server 49

The server 49
Implementing CORBA objects on the server 50
The server's perspective 52
Implementing the bank server 53

## 7 The Harlequin Common Lisp ORB 61

The CORBA modules 61
Parsing IDL into stubs and skeletons 61
Configuring a name service and an interface repository 62
Error handling in user code called by the server 63

# Appendix A Common Lisp IDL Binding 65

Introduction to IDL 65
How IDL is used 66
Mapping summary 66
Mapping in more detail 67
Mapping pseudo-objects to Lisp 89
The mapping of IDL into Common Lisp servants 91

#### Index 95

# **Preface**

#### **Product**

The Harlequin's Common Lisp Object Request Broker  $^{\text{\tiny TM}}$  (ORB  $^{\text{\tiny TM}}$ ) supports the Common Object Request Broker Architecture (CORBA  $^{\text{\tiny B}}$ ) defined by Object Management Group  $^{\text{\tiny B}}$ , Inc. (OMG  $^{\text{\tiny TM}}$ ).

Harlequin's Common Lisp ORB and supporting tools provide CORBA architecture functionality to Common Lisp programmers, combining standardized distributed system development with a state-of-the-art dynamic object-oriented language.

#### **Parts**

The CORBA components included in The Harlequin Common Lisp ORB are:

1. Four fasls that are placed in the library directory, and which may be required into the image:

corba-support The client side of the ORB runtime plus the IDL parser.

corba-orb The server side, the actual Harlequin Common Lisp

ORB, which loads corba-support if it is not present.

A place holder for a number of graphical tools. (On

Liquid Common Lisp this requires common-lispworks.)

corba-orbix-support

A number of patches to support OrbixWeb.

2. Some example Common Lisp code including the subfolders:

```
hello-world
bank
```

of the examples subfolder of the Harlequin installation folder.

3. This manual.

#### **Audience**

This manual is intended for use by application programmers who wish to build CORBA applications using Common Lisp. The guide assumes that the reader is familiar with both the Common Lisp programming language and with building distributed applications using CORBA.

### Standards compliance

The Harlequin Common Lisp ORB conforms to the CORBA 2.0 specification with some elements of CORBA 2.2, most notably the Portable Object Adapter (POA). Work is underway to complete CORBA 2.2 compliance.

## **Further reading**

Many resources exist for those who want to learn about CORBA and distributed software development. The OMG maintains a great starting point for beginners at:

```
http://www.omg.org/corba/beginners.html (This address may change.)
```

Some recommended highlights are:

CORBA Web pages like "A CORBA overview" at

```
http://www.infosys.tuwien.ac.at/Research/Corba/
OMG/arch2.htm#446864
(This address may change.)
```

Related books and magazines like

Instant CORBA by R. Orfali

Published by John Wiley & Sons, 1997

ISBN 0-471-18333-4

• And mailing lists like CORBA Development

corba-dev@randomwalk.com

This list discusses building CORBA-based systems. To subscribe, send email to corba-dev-request@randomwalk.com with subscribe corba-dev in the body of the message.

Preface

# 1

# Common Lisp and CORBA

#### 1.1 About CORBA

Object Management Group, Inc. describe their CORBA architecture as follows:

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

The ORB is the middleware that establishes the client-server relationships between objects. Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

In fielding typical client/server applications, developers use their own design or a recognized standard to define the protocol to be used between the devices. Protocol definition depends on the implementation language, network transport and a dozen other factors. ORBs simplify this process. With an ORB, the protocol is defined through the application interfaces via a single implementation language-independent specification, the IDL. And ORBs provide flexibility. They let programmers choose the most appropriate operating system, execution environment and even programming language to use for each component of a system under construction. More importantly, they allow the integration of existing components. In an ORB-based solution, developers simply model the legacy component using the same IDL they use for creating new objects, then write "wrapper" code that translates between the standardized bus and the legacy interfaces.

CORBA is a signal step on the road to object-oriented standardization and interoperability. With CORBA, users gain access to information transparently, without them having to know what software or hardware platform it resides on or where it is located on an enterprise's network. The communications heart of object-oriented systems, CORBA brings true interoperability to today's computing environment.

At the time of writing, the text above was available at:

http://www.omg.org/corba/whatiscorba.html

It has been reproduced with permission.

# 1.2 About the Harlequin Common Lisp ORB

Harlequin has written a CORBA-compliant, native-IIOP ORB in Common Lisp.

The Harlequin Common Lisp ORB will let you build and run distributed applications in Common Lisp, straight out of the box. When combined with a database interface, you are able to build three-tier, client-server applications completely in Common Lisp.

However, the *raison d'etre* of CORBA is interoperability. Hence, the choice of IIOP provides immediate interoperation with any other ORB you may be using. For example, given a Java ORB you could write GUI clients in AWT that communicate with servers written in Common Lisp. Conversely, given a C++ ORB you can build CAPI clients that talk to C++ servers.

The advantages of building an ORB in Common Lisp (apart from proving that Common Lisp can tackle another complex domain) are:

- ORB-vendor independence The Harlequin Common Lisp ORB can be married to any existing ORB infrastructure, or introduced first without affecting later ORB procurement decisions.
- "Batteries included" No need to purchase a separate ORB to get a full system. The Harlequin Common Lisp ORB provides "instant CORBA" to get distributed Common Lisp applications up and running without additional procurement or installation.
- Lower impedance mismatch No need to trampoline from Common Lisp to IIOP via another language binding or via a non-Common Lisp IIOP engine API.
- CORBA expertise A deeper understanding of CORBA inside Harlequin, which can be shared with our customers via technical support.
- CORBA customization Harlequin likes to offer a high level of customization support and consulting to both internal and external customers which it could not do with a third-party product written in a more primitive language. This also means faster fixes for basic bugs.
- 100% pure Common Lisp Providing users with enhanced debugging and interaction facilities available in a homogeneous implementation model.
- Inter-Language compatibility Harlequin is building kernel ORBs for Dylan and Standard ML as well. The Common Lisp ORB borrows from the same design pattern.

# 1.3 Features of the Harlequin Common Lisp ORB

The following features are supported:

- CORBA 2.0 with parts of CORBA 2.2 (notably the POA)
- Internet Inter-ORB Protocol (IIOP) 1.0 (GIOP 1.0)
- Portable Object Adapter (POA)
- Dynamic Invocation Interface (DII)
- Dynamic Skeleton Interface (DSI)
- Common Lisp Language Binding

# 1.4 CORBA examples

The Harlequin Common Lisp ORB includes two example applications to help you start writing client/server applications in CORBA. We study these applications later in this guide.

Hello World A client/server implementation of the standard Hello

World application.

Bank A three-tier client/server implementation of a banking

application.

This manual walks you through these examples, which are ready to build and run straight out of the box.

# **Quick Start Tutorial**

In this chapter, we develop a very basic CORBA application, designed to demonstrate some of the key concepts for using Common Lisp for distributed objects.

The chapter aims to show you the sort of coding involved in using CORBA with Common Lisp, and to get a client/server application up and running quickly. It is not concerned so much with explaining how things work. Subsequent chapters go into more detail, using a deeper example, and explaining the approach we have taken to implementing the CORBA architecture for Common Lisp.

In this example application, a client program asks a server program for a string and prints it to standard output. This chapter is going to take you through the basic steps needed to create the application.

# 2.1 A CORBA-based Hello World

This chapter's example is an implementation of the standard Hello World application, using Common Lisp and CORBA. In this version of Hello World, a client application asks a server application for a string. When the client receives the string, it prints it to the standard output, and then exits.

We will take these basic steps to create the application:

- 1. Define the interface.
  - We define the interface to the server using OMG's *Interface Definition* Language (IDL).
- 2. Implement the client.
- Implement the server.
- Build and test the application.

The complete Hello World application can be found in the examples/corba/hello-world folder. (For instance, in a default LispWorks 4.1 Windows NT installation, the location is c:\Program Files\Harlequin\ LispWorks\lib\4-1-0-0\examples\corba\hello-world.)

### 2.1.1 Defining the interface

We first need to define the interface of the Hello World server object. The client and server will communicate across this interface. The interface is defined using IDL in a single file that must have the extension .idl.

- 1. Create a file called hello-world.idl.
- 2. Enter the IDL declaration below into the hello-world, idl file.

```
module HelloWorld {
  interface world {
    string hello();
  };
};
```

This IDL declaration says that there are CORBA objects of a kind called world, and that there is an operation called hello on world objects that takes no arguments and returns a string. Servers implement world, and clients call hello on instances of world.

Now that we have written the IDL, we can run the IDL parser over it to produce *stub* and *skeleton* code for the client and server parts of the application.

## 2.1.2 Generating the stub and skeleton code from IDL

We need the IDL parser to parse the IDL to generate appropriate stubs and skeletons. We do this by including the IDL file in the defsystem that defines the code we are writing. For Hello World, the relevant defsystem is:

The defsystem utility has been extended to correctly handle a file of type idl-file. In this case, the fasl corresponding to the IDL file contains the compiled stubs and skeletons for the given IDL and is generated when we compile the system.

To create a defsystem file for the Hello World application:

- 1. Create a defsystem file called defsys.lisp.
- 2. Enter the following Lisp code into the defsys.lisp file:

3. Save and close the defsystem file.

When it comes time to run the application, stubs and skeletons will be generated.

## 2.1.3 Defining utilities for sharing an object reference

Now we will define some utilities for communicating an object reference from the server to the client by converting the object reference into a string using ORB-supplied functions and writing it to a shared file. The client can then read the string from the shared file and convert it back into an object reference. Note that a real application would probably use a higher level service such as a Name Service for passing object references between applications.

- 1. Create a file called shared.lisp.
- 2. Enter the following Common Lisp code into the shared.lisp file:

```
(in-package "CL-USER")
(defparameter *hello-world-ior-file*
 #+Harlequin-PC-Lisp "c:/temp/hello.ior"
 #-Harlequin-PC-Lisp "/tmp/hello.ior")
(defun object-to-file (orb object)
  (with-open-file (st *hello-world-ior-file* :direction :output
                                          :if-exists :supersede)
    (prin1 (op:object_to_string orb object) st)))
(defun file-to-object (orb)
  (with-open-file (st *hello-world-ior-file*)
    (op:string_to_object orb (read st))))
```

This code does the following:

- object-to-file opens the shared file and uses the op:object\_to\_string function to convert the object reference into a string, which is then written into the file
- file-to-object performs the inverse operation: it reads the string from the file and uses op:string\_to\_object to convert the string back into a client-side proxy object
- 3. Save and close the shared.lisp file.
- 4. Add shared.lisp to the defsystem by adding one line of code to the defsys.lisp file, which should then look like this:

```
(in-package "CL-USER")
(require "corba-orb")
(defsystem hello-world-corba-object ()
  :members (
            ("hello-world" :type :idl-file)
            "shared"
            ))
```

#### 2.1.4 Implementing the client

Now we will implement the client side of the Hello World application. We create a file hello-world-client.lisp and add it to the defsystem. (You can implement this as you wish, but here is one possible implementation.)

- 1. Create a file called hello-world-client.lisp.
- **2.** Enter the following Common Lisp code into the hello-world-client.lisp file:

This code does the following:

- · gets a world object from somewhere
- invokes op:hello on the object to get a string
- · writes out the string and a new line to the standard output stream

The elided details are not important at this stage, they involve getting an object reference from somewhere. In the full source at the end of this chapter (Section 2.2) you can see that a shared file is used to pass a stringified object reference.

- 3. Save and close the hello-world-client.lisp file.
- 4. Add hello-world-client to the defsystem by adding one line of code to the defsys.lisp file, which should then look like this:

```
(in-package "CL-USER")
(require "corba-orb")
```

```
(defsystem hello-world-corba-object ()
  :members (
            ("hello-world" :type :idl-file)
            "shared"
            "hello-world-client"
            ))
:rules ((:in-order-to :compile "all
         (:requires (:load :previous)))))
```

## 2.1.5 Implementing the server

Implementing the server is also easy. We create a file hello-world-server.lisp.

In the server the main function is less interesting because it is concerned with the administrative details of writing out a stringified form of the object reference into the shared file and initializing the server. The actual core of the application implementation is:

```
(defclass world-implementation (HelloWorld:world-servant) ())
(corba:define-method op:hello ((self world-implementation))
  (declare (ignore self))
  "Hello World!")
```

This subclasses a special generated class on the server side called a *servant*, and then implements a method on op:hello that actually returns the desired string.

- 1. Create a file called hello-world-server.lisp.
- 2. Enter the following code into hello-world-server.lisp:

```
(in-package "CL-USER")
(defclass world-implementation (HelloWorld:world-servant) ())
(corba:define-method op:hello ((self world-implementation))
 (declare (ignore self))
 "Hello World!")
```

3. Add hello-world-server to the defsystem by adding one line of code to the defsys.lisp file, which should then look like this:

#### 2.1.6 Building and testing the application

To build and test this distributed Hello World application, you must copy the rest of the source code from Section 2.2 on page 20 into the respective files. The code can also be found in the <code>corba/hello-world</code> subdirectory of the standard examples directory.

After supplementing your files with the complete source code, perform the following steps in the Listener to run the example:

**1.** Load the defsystem file by entering:

```
(load "defsys")
(compile-system "HELLO-WORLD-CORBA-OBJECT" :load t)
```

You may need to adjust the (load "defsys") command for your platform and/or provide a more precise location on your machine. For example, for a default LispWorks 4.1 Windows NT installation, you would specify:

```
(load "C:\\Program Files\\Harlequin\\LispWorks\\lib\\
 4-1-0-0\\examples\\corba\\hello-world\\defsys")
```

Now, you can run the application to test that it works.

2. If you are using LispWorks on a UNIX platform and not running with multiprocessing enabled, then call:

```
(mp:initialize-multiprocessing)
```

You need to run the server first so that it is waiting and ready to receive calls from the client.

**3.** Enter the command:

```
(cl-user::server-startup)
```

**4.** You can then run the client using:

```
(cl-user::run-client)
```

Note that you do not have to be running the client and the server in the same Lisp image (although you can if desired). In the simple example we have just implemented, they must be running on the same machine (to allow the object reference to be shared using a single file), but we have true location transparency in the way the client can be written with no regard for the location of the server process.

# 2.2 Complete source code for the Hello World example

The complete source code for the Hello World application is included here for your convenience. It can also be found in the corba/hello-world subdirectory of the standard examples directory.

#### 2.2.1 The complete interface source code

The complete code for the Hello World interface (the hello-world.idl file), written in IDL, is:

```
module HelloWorld {
  interface world {
    string hello();
  };
};
```

#### 2.2.2 The complete defsystem source code

The complete code for the Hello World defsystem (the defsys.lisp file) is:

# 2.2.3 The complete source code for the file transfer of the IOR

The complete code for the Interoperable Object Reference (IOR) file transfer (the shared.lisp file) is:

```
(defun file-to-object (orb)
  (with-open-file (st *hello-world-ior-file*)
    (op:string_to_object orb (read st))))
```

# 2.2.4 The complete Hello World client source code

The complete code for the Hello World client (the hello-world-client.lisp file) is:

```
(in-package "CL-USER")
(defun run-client ()
 (let ((orb (op:orb_init nil "Harlequin Common Lisp Orb")))
    (let ((world (op:narrow 'HelloWorld:world (file-to-object
      (format t "~S~%" (op:hello world)))))
```

#### 2.2.5 The complete Hello World server source code

The complete code for the Hello World server (the hello-world-server.lisp file) is:

```
(in-package "CL-USER")
(defclass world-implementation (HelloWorld:world-servant) ())
(corba:define-method op:hello ((self world-implementation))
 (declare (ignore self))
  "Hello World!")
(defun server-startup ()
 (let* ((orb (op:orb_init nil "Harlequin Common Lisp Orb"))
         (poa (op:resolve_initial_references orb "RootPOA"))
         (impl (make-instance 'world-implementation))
         (world (op:narrow 'HelloWorld:world
                           (op:servant_to_reference poa impl))))
    (object-to-file orb world)
    (let ((manager (op:the_poamanager poa)))
      (op:activate manager))))
```

# Setting up the Bank Example

Chapters 3–6 guide you through the development of a more complex client-server application using the Harlequin Common Lisp ORB. This example application illustrates how to implement and use CORBA objects in Common Lisp.

# 3.1 About the bank example

The example described in this tutorial is a simple simulation of a bank. The architecture of the bank is composed of three components:

- A database that provides persistent storage for accounts managed by the bank. (This is simulated in the following code, but is easily replaced by a real database backend.)
- A CORBA server that represents the bank and provides an object-oriented interface to its accounts.
- A CORBA client that provides a graphical user interface to the bank.

The server provides a single CORBA object that represents the bank. This object manages a collection of CORBA objects that represent customer accounts. The bank has operations for opening and closing accounts, and for retrieving existing accounts from the database. In turn, accounts support operations for querying and updating their balance.

The client initially contacts the server by obtaining a reference to the bank object from the Harlequin Common Lisp ORB. It then presents the user with a graphical interface to the bank.

In response to user requests, the interface invokes operations on the bank, obtaining further references to accounts created on the server. The client manages separate graphical objects for the bank and each of the accounts that are active in the server.

The user interface is implemented using the Harlequin CAPI library. Note that this application is a typical example of a three-tier architecture comprising a database access layer, a business logic layer, and a user interface layer.

# 3.2 Where to find the example code

The bank example code developed in this tutorial can be found in the toplevel Harlequin installation folder, under

```
examples/corba/bank
```

This folder has several subfolders.

- examples/corba/bank, which contains the IDL that defines the CORBA interface to the server (the file bank.idl)
- examples/corba/bank/client, which contains the implementation of the client
- examples/corba/bank/server, which contains the implementation of the server

# 3.3 Building the bank client and server

We can now build the client and server applications for the demo. The client and server implementations are available in the respective examples subdirectories.

In your Common Lisp environment,

**1.** Build the client by running:

```
(load "examples/corba/bank/client/defsys")
(compile-system "bank-client" :load t)
```

You may need to adjust the (load "defsys") command for your platform and/or provide a more precise location on your machine. For example, for a default LispWorks 4.1 Windows NT installation, you would specify:

```
(load "C:\\Program Files\\Harlequin\\LispWorks\\lib\\
4-1-0-0\\examples\\corba\\bank\\client\\defsys")
```

**2.** Build the server by running:

```
(load "examples/corba/bank/server/defsys")
(compile-system "bank-server" :load t)
```

The defsystem automatically invokes the IDL compiler on the file bank.idl to generate the source code for the skeletons and stubs, which are compiled and stored in the bank fasl file. Following the standard defsystem rules, the IDL will only be recompiled if the fasl file is out of date.

# 3.4 Running the server and client

We can now run the bank demo for the first time. To run the server:

1. In the Listener, run:

```
(load "examples/corba/bank/server/defsys")
(compile-system "bank-server" :load t)
(cl-user::bank-server)
```

After performing some initialization, the application presents an information dialog to indicate that the server is ready. This dialog has a single **Stop server** button to shut down the server.

After the server's dialog has appeared:

**2.** Run the following commands in a separate invocation of Common Lisp (in another image):

```
(load "examples/corba/bank/client/defsys")
(compile-system "bank-client" :load t)
(cl-user::bank-client)
```

A single Corba Bank window appears, presenting a GUI to the bank. You can now interact with the bank using the Action menu to create new accounts, deposit amounts, and so on.

Once you have finished interacting with the bank, close the Corba Bank window to exit the client application. Then click the Stop server button in the server's dialog to exit the server.

# Writing and Compiling IDL

# 4.1 Designing the IDL

The first step in developing a CORBA application is to define the interfaces to its distributed application objects. We can define these interfaces using the CORBA Interface Definition Language (IDL).

Essentially, the IDL specification of an interface lists the names and types of operations that:

- · any CORBA object, satisfying that interface, must support
- any CORBA client, targeting such an object, may request

Our application manages three types of CORBA object, representing accounts, checking accounts, and banks. We declare the interfaces to all three objects within the same CORBA module, BankingDemo:

```
module BankingDemo {
  interface account {
    // details follow
  };
  interface checkingAccount : account {
    // details follow
  };
```

```
interface bank {
    // details follow
 };
};
```

The following subsections describe the IDL declarations for each of the three interfaces. You can find the complete IDL description for the bank demo in the examples folder, under

examples/corba/bank/bank.idl

#### 4.1.1 IDL for the account interface

We begin with the IDL definition of the interface to an account object.

```
// in module BankingDemo
interface account {
 readonly attribute string name;
 readonly attribute long balance;
 void credit (in unsigned long amount);
  exception refusal {string reason;};
 void debit (in long amount)
    raises (refusal);
};
```

The name of an account is recorded in its name attribute. The state of an account is recorded in its balance attribute. To keep things simple, we use CORBA long values to represent monetary amounts.

To prevent clients from directly altering the account's name or balance, these attributes are declared as readonly attributes. The operations credit and debit are provided to allow updates to an account's balance attribute.

The operation credit adds a non-negative amount to the current account balance.

Next comes an exception declaration:

```
exception refusal {string reason;};
```

This declares a named exception, refusal, that the debit operation uses to signal errors. The refusal exception is declared to contain a reason field that documents the reason for failure in the form of a string.

The operation debit subtracts a given amount from the current balance, provided doing so does not make the account balance negative. Qualifying debit by the phrase

```
raises (refusal)
```

declares that invoking this operation may raise the exception refusal. Although a CORBA operation may raise any CORBA system exception, its declaration must specify any additional user-defined CORBA exceptions that it might raise.

This completes the IDL declaration of the account interface.

#### 4.1.2 IDL for the checkingAccount interface

The bank application also manages another sort of account called a checking account. While an ordinary account must maintain a positive balance, a checkingAccount may be overdrawn up to an agreed limit.

We use IDL's notion of interface inheritance to capture the intuition that a checking account is a special form of account:

```
// in module BankingDemo
interface checkingAccount : account {
  readonly attribute long limit;
};
```

The declaration checkingAccount: account specifies that the interface checkingAccount inherits all the operations and attributes declared in the account interface. The body of the definition states that a checkingAccount also supports the additional limit attribute.

The fact that checkingAccount inherits some operations from account does not imply that the methods implementing those operations need to be inherited too. We will exploit this flexibility to provide a specialized debit method for checkingAccounts.

#### 4.1.3 IDL for the bank interface

We can now design the interface of a bank object. The intention is that a bank associates customer names with accounts, with each name identifying at most one account. A client is able to open accounts for new customers and to retrieve both accounts and checking accounts for existing customers from the persistent store. If the client attempts to open a second account under the same name, the bank should refuse the request by raising an exception. Similarly, if the client attempts to retrieve an account for an unknown customer, the bank should reject the request by raising an exception.

The IDL definition of the bank interface captures some of these requirements:

```
// in module BankingDemo
interface bank {
 readonly attribute string name;
 exception duplicateAccount{};
 account openAccount (in string name)
   raises (duplicateAccount);
 checkingAccount openCheckingAccount(in string name,
                                      in long limit)
   raises (duplicateAccount);
 exception nonExistentAccount{};
 account retrieveAccount(in string name)
   raises (nonExistentAccount);
 void closeAccount (in account account);
};
```

The name of a bank is recorded in its name attribute.

The operation openAccount is declared to take a CORBA string and return an account. Because account is defined as an interface, and not a type, this means that the operation will return a reference to an account object. This illustrates an important distinction between ordinary values and objects in CORBA: while members of basic and constructed types are passed by value, objects are passed by reference.

The qualification raises (duplicateAccount) specifies that openAccount can raise the user-defined exception duplicateAccount, instead of returning an account. (The exception duplicateAccount has no fields.)

The operation openCheckingAccount is similar to openAccount, but takes an additional argument, limit, which represents the account's overdraft limit.

The operation retrieveAccount looks up the account (or checking account), if any, associated with a customer name, returning an object reference of interface account. The operation may raise the exception nonExistentAccount to indicate that there is no account under the supplied name.

The last operation, closeAccount, closes an account by deleting it from the bank's records.

Because checkingAccount inherits from account, a checkingAccount object may be used wherever an account object is expected, whether as the actual argument, or the result, of an operation. For instance, this means that we can use closeAccount to close checking accounts as well as accounts; and to use retrieveAccount to fetch checking accounts as well as accounts.

The complete IDL definition for the bank can be found in file bank.idl.

# 4.2 Compiling IDL for a CORBA application

The Harlequin Common Lisp ORB product includes an IDL compiler that it uses to check and compile IDL files into Common Lisp fasl files. When the IDL file is processed by the Harlequin preprocessor, it maps the IDL into Common Lisp. Appendix A, "Common Lisp IDL Binding" provides a specification for a standard mapping from CORBA IDL to Common Lisp.

According to this specification:

- CORBA types are mapped to Common Lisp types and classes
- CORBA interfaces are mapped to Common Lisp classes
- CORBA interface inheritance is mapped to Common Lisp class inheritance
- CORBA attributes are mapped to Common Lisp getter and setter functions

- CORBA operations are mapped to Common Lisp generic functions
- CORBA exceptions are mapped to Common Lisp conditions

By including the IDL file in the defsystem, the application can access these mappings.

# 4.3 Mapping IDL to Common Lisp

To provide a better understanding of the IDL to Common Lisp mapping, we can take a look at the result of applying the mapping to the file bank.idl.

Parsing the IDL defines a package BANKINGDEMO.

As an example of the mapping scheme, the following subsections examine the Common Lisp counterparts of some of the more representative IDL declarations from the file bank, idl.

## 4.3.1 Mapping for basic types

The IDL types string, long, and unsigned long are mapped to the Common Lisp types corba:string, corba:long, and corba:ulong, which are typedefs for the types string, integer, and integer.

# 4.3.2 Mapping for interfaces

The IDL interfaces account, checkingAccount, and bank map to the Common Lisp classes BankingDemo: account, BankingDemo: checkingAccount, and BankingDemo: bank.

Notice how IDL interface inheritance (checkingAccount: account) maps naturally onto Common Lisp class inheritance: the class BankingDemo: checkingAccount is defined as a subclass of BankingDemo: account.

### 4.3.3 Mapping for attributes

The read-only balance attribute of an IDL account gives rise to the Common Lisp generic functions:

op:balance

If we had omitted the readonly keyword from the definition of the balance attribute, the mapping would have introduced an additional generic setter function:

(setf op:balance)

# 4.3.4 Mapping for operations

The IDL operation credit is mapped to the Common Lisp generic function:

op:credit

In IDL, the credit operation is defined within the account interface, declaring it to be an operation on account objects. The Common Lisp language binding adopts the convention that an operation's target object should be passed as the first argument of the corresponding Common Lisp generic function. Thus the first parameter of the generic function op:credit is an object of type BankingDemo:account.

The operation's in and inout arguments become the remaining parameters of the corresponding Common Lisp generic function. In this case, the credit operation specifies a single in parameter, in unsigned long amount, that determines the second and only other parameter, amount, of the op:credit generic function.

The operation's result type, and any other parameters declared as out or inout, become results of the corresponding Common Lisp generic function. In this case, because the result type of credit is void, and the operation has no out or inout parameters, op:credit has an empty result list.

## 4.3.5 Mapping for exceptions

The IDL exception refusal maps onto the Common Lisp class
BankingDemo:account/refusal. Its member, reason string; maps onto a
slot reason of type string.

Note that BankingDemo:account/refusal is a subclass of CORBA:userexception and, by inheritance, of Common Lisp condition. This means that CORBA user exceptions can be raised on the server, and handled in the client, using the standard Common Lisp condition mechanism.

4 Writing and Compiling IDL

# The Bank Client

## 5.1 The client

In this section, we design and implement a CORBA client. Our client presents a graphical user interface to a bank object and its operations. We implement the user interface using CAPI.

Because the primary motivation for this tutorial is to illustrate the use of CORBA, we focus less on the design of the graphical interface, and more on the method for interacting with CORBA objects.

# 5.2 The client's perspective

From the client's perspective, the IDL definition of a bank's interface fully determines its functionality. This means that we need only rely on the information in the IDL to interact with a bank object. In particular, knowing the IDL description, we can implement the client before an implementation of a bank object is available.

The bank fasl file, produced by the IDL compiler

- Specifies the protocol for interacting with CORBA objects that satisfy the interfaces in the IDL file, bank.idl, and
- Provides the client-side implementation of this protocol

This fasl should be used by any application that wants to act as a client with respect to some CORBA object matching an interface in the bank.idl file. The bank fasl file defines classes BankingDemo:account,

BankingDemo: checkingAccount, and BankingDemo: bank.

The class BankingDemo:checkingAccount is defined to inherit from BankingDemo:account, matching the inheritance relationship in the IDL. Instances of these classes act as proxies for CORBA objects on the server.

The bank fasl also defines a *stub* method, specialized on the appropriate proxy class, for each protocol function stemming from an IDL attribute or operation. When the client applies the generic function to a particular target proxy, the stub method communicates with the ORB to invoke the corresponding operation on the actual target object in the server. If the request succeeds, the stub method returns the result to the client. If the request fails, raising a CORBA user or system exception, the stub method raises the corresponding Common Lisp condition of the appropriate class. This condition can then be handled by the client code using standard Common Lisp constructs.

# 5.3 Implementing the bank client's GUI

To keep things simple, we organize the structure of the user interface to closely match the IDL description of the bank. Each CORBA object is presented in its own interface. We define one subclass of capi:interface for each CORBA interface.

The definition of these subclasses is derived from the declaration of their corresponding CORBA interfaces. In particular, we use display panes to represent IDL attributes, and buttons or menu items to invoke IDL operations. Each interface contains a slot that contains the CORBA object it represents.

Clicking on a button of the frame triggers a callback that invokes the corresponding operation on the CORBA object associated with that frame. The user is notified of any CORBA user-exceptions that these operations raise. The CORBA-specific code resides, to a large extent, in these callbacks.

The source code for the client's GUI is in the file client/interfaces.lisp.

#### 5.4 Defining the interfaces

**Note:** This section assumes some basic familiarity with the CAPI library. See the *CAPI Reference Manual* for details.

In this section, we define three CAPI interface classes account-interface, checkingAccount-interface, and bank-interface. These classes are used to present graphical interfaces to CORBA objects with the IDL interfaces account, checkingAccount, and bank.

We begin by defining the interface class account-interface:

```
(capi:define-interface account-interface ()
 ((account-ref :initarg :account-ref)
  (account-name :initarg :account-name :accessor account-name)
  (bank-interface :initarg :owner))
 (:panes
 (balance-field capi:display-pane
                  :title (:initarg :account-name)
                  :min-width '(:character 10)
                  :max-width nil)
  (button-panel capi:push-button-panel
                 :callbacks '(credit debit)
                 :items '("Credit" "Debit")
                 :callback-type :interface))
 (:layouts
  (account-layout capi:column-layout '(balance-field
                                        button-panel)))
 (:default-initargs :auto-menus nil :max-width t))
```

This is how we use an instance of class account-interface. We store the name of the customer owning this account in the title of the display pane (using initarg :title).

The account-ref slot stores a CORBA object reference (of class BankingDemo:account) to the corresponding CORBA account object on the server. The bank-interface slot stores a pointer to the bank interface for this object.

The pane balance-field reports the state of the CORBA object's balance attribute as a readonly text field. We delegate the initialization of this field value to an initialize-instance after method specialized on account-interface. The value needs to be updated after each invocation of a CORBA debit or credit operation.

The button panel button-panel defines buttons to activate callbacks debit-callback and credit-callback. These callbacks prompt the user for amounts and then invoke the corresponding CORBA operations debit and credit on the object reference stored in the account-ref field. We will implement these callbacks in a moment.

The buttons are laid out in a column layout account-layout. Mirroring the fact that the IDL interface checkingAccount inherits from account, we define the Common Lisp frame class checking-account-interface as a subclass of account-interface:

```
(capi:define-interface checking-account-interface
                             (account-interface) ()
  (:panes
   (limit-field capi:display-pane
                :min-width '(:character 10)
                :max-width nil))
  (:layouts
  (checking-account-layout capi:column-layout
                            '(account-layout limit-field))))
```

The pane limit-field reports the state of the CORBA object's limit attribute as a readonly text field. Again, we can delegate the initialization of this field's value to an initialize instance after method specialized on checking-account-interface.

The layout checking-account-layout simply lays out the inherited layout account-layout, containing the account's balance, together with the additional limit-field.

The definition of bank-interface class follows the same pattern:

```
(capi:define-interface bank-interface ()
  ((bank-ref :initarg :bank-ref))
  (:menu-bar open-actions)
  (:menus
   (open-actions
    "Action"
    (("Open Account" :callback 'open-account-callback)
     ("Open Checking Account" :callback
                               'open-checking-account-callback)
     ("Retrieve Account" :callback 'retrieve-account-callback)
     ("Close Account" :callback 'close-account-callback))
    :callback-type :interface))
  (:layouts
   (accounts-area capi:row-layout ()
                  :accessor accounts-area
                  :horizontal-scroll t))
 (:default-initargs :auto-menus nil :best-width 400))
```

The accounts-area layout keeps track of the account-interfaces created by the bank interface as the result of invoking operations on the CORBA bank object. This list is maintained to prevent the user from obtaining more than one interface to the same account. We need to update it whenever an account interface is exited.

The interface menu items Open Account, Open Checking Account, Retrieve Account, and Close Account activate callbacks open-Account-callback, open-Checking-Account-callback, retrieve-Account-callback, and close-Account-callback.

These callbacks prompt the user for appropriate arguments and then invoke the corresponding CORBA operations openAccount, openCheckingAccount, retrieveAccount, and closeAccount on the object reference stored in the bank-ref slot. We will see the implementation of these callbacks in a moment.

#### 5.4.1 Initializing and exiting account frames

Each time we make a new account interface we want to ensure two things:

- The account interface is registered in the bank interface that spawned it
- Its balance pane displays the correct value

An easy way to do this is to add an initialize-instance after method specialized on account-interface. (In Common Lisp, each call to make an instance of a given class is automatically followed by a call to initialize that instance; you are free to specialize the initialize-instance generic function on particular classes.)

```
(defmethod initialize-instance :after ((self account-interface)
  (with-slots (account-ref balance-field) self
    (when account-ref
      (setf (capi:display-pane-text balance-field)
            (format nil "~A" (op:balance account-ref))))))
```

Here, we encounter our first example of invoking a CORBA operation on a CORBA object. The Common Lisp variable account-ref, of class BankingDemo: account, contains a proxy for a CORBA account object on the server. The application (op:balance account-ref) invokes a stub method specialized on the proxy's class. The stub method forwards the request across the ORB to the actual object on the server.

The request is executed on the object in the server and the result passed back across the ORB to the stub method, which returns the value to the client as a corba:long. This value is then used to set the initial value of the balance field.

We can initialize the interface for a checking account in a similar way:

```
(defmethod initialize-instance :after (
     (self checking-account-interface) &key)
 (with-slots (account-ref limit-field) self
    (when account-ref
      (setf (capi:display-pane-text limit-field)
            (format nil "~A" (op:limit account-ref))))))
```

Inheritance ensures that the method on account-interface is called which registers the interface and sets up its balance field; a call to the op:limit stub determines the initial value of its limit field.

For convenience, we define a generic function make-account-frame that makes the correct class of frame for a given account object reference:

```
(defmethod make-account-frame ((self BankingDemo:account)
                                &key bank-interface title)
 (push-new-item bank-interface
                 title
                 (make-instance 'account-interface
                                :account-ref self
                                :account-name title
                                :owner bank-interface)))
(defmethod make-account-frame ((self BankingDemo:checkingAccount)
                                &key bank-interface title)
 (push-new-item bank-interface
                 title
                 (make-instance 'checking-account-interface
                                :account-ref self
                                :account-name title
                                :owner bank-interface)))
```

These methods simply dispatch on the class of the object reference to make an account-interface or checkingAccount-interface as appropriate.

#### 5.4.2 Defining the callbacks

Defining the callbacks attached to each button is straightforward. Recall that in CAPI, because we stated that the button callback type was :interface, the argument passed to a callback is the interface whose activation triggered that callback.

The credit-callback is activated by the Credit button of some account interface:

The callback is passed the account interface. It then extracts the CORBA object reference stored in the frame's account-ref slot and prompts the user for an amount. The function capi:prompt-for-integer queries the user for an integer and returns nil if the user cancels the dialog. If the amount is valid, the callback invokes the stub method op:credit on the CORBA object reference with the specified absolute value of the amount (recall that the credit opera-

tion expects an unsigned long as its argument). Finally, it updates the balance field of the frame with the current value of the object's balance attribute, obtained by invoking the stub method op:balance.

The definition of debit-callback is very similar to the definition of credit-callback:

```
(defun debit (self)
 (with-slots (balance-field account-ref) self
   (let ((amount (capi:prompt-for-integer "Amount?" :min 0)))
      (when amount
        (handler-case
            (progn
              (op:debit account-ref amount)
              (setf (capi:display-pane-text balance-field)
                    (format nil "~A" (op:balance account-ref))))
          (BankingDemo:account/refusal
           (capi:display-message "Debit returned refusal with
                                 string: <~A>"
                                 (op:reason xx))))))))
```

The only difference is that debit-callback must deal with the additional possibility that the debit operation, when invoked on the target object, may fail, raising the IDL exception refusal. If the object raises this exception, the op:debit stub method signals it as a Common Lisp condition of class BankingDemo:account/refusal.

The exception can then be caught and handled in any of the standard Common Lisp ways. Here, we simply place the invocation in the body of a handler-case statement with an appropriate exception clause to handle the condition.

The open-account-callback is activated by the openAccount-button of some bank frame:

The callback extracts the CORBA object reference stored in the interface's bank-ref slot. The function capi:prompt-for-string queries the user for the new customer's name returning a string (or nil if the user cancels the dialog). If the dialog has not been cancelled, the callback invokes the stub method op:openAccount on the target object reference bank, passing the argument name. If successful, the invocation returns an object reference, of class BankingDemo:account, to an IDL account object, which is then used to make and start a new account-interface, via a call to make-account-frame.

Recall that the IDL operation openAccount may fail, raising the IDL user exception duplicateAccount. As in the definition of debit-callback, we cater for this eventuality by placing the invocation in the body of a handler-case statement and install a handler on the corresponding Common Lisp condition of class BankingDemo:bank/duplicateAccount. This handler simply informs the user of the exception using the CAPI function display-message to create and display a simple alert dialog box.

The definition of open-checking-account-callback is similar to the definition of openAccount-callback but prompts the user for an additional integer to pass as the overdraft limit of the new checking account:

```
(defun open-checking-account-callback (self)
 (with-slots (bank-ref) self
   (let ((name (capi:prompt-for-string "Name?")))
      (when name
        (let ((limit (capi:prompt-for-integer "Limit?")))
          (when limit
            (handler-case
                (let ((account-ref
                       (op:opencheckingaccount bank-ref
                                              name limit)))
                  (make-account-frame account-ref
                   :bank-interface self :title name))
              (Bankingdemo:Bank/DuplicateAccount
               (capi:display-message "Cannot create another
                                    account for ~A" name)))))))))
```

While openAccount and openCheckingAccount create accounts for new customers, the retrieveAccount operation is simply meant to look up the account of an existing customer:

```
(defun retrieve-account-callback (self)
  (with-slots (bank-ref) self
    (let ((name (capi:prompt-for-string "Name?")))
      (when name
        (if (find-named-frame self name)
            (capi:display-message "Already viewing it...")
          (handler-case
              (let ((account-ref
                     (op:retrieveaccount bank-ref name)))
                (when (op:Is_a account-ref (op:id
                   Bankingdemo:_Tc_Checkingaccount))
                  (setf account-ref
                        (op:narrow 'Bankingdemo:Checkingaccount
                                    account-ref)))
                (make-account-frame account-ref
                  :bank-interface self :title name))
            (Bankingdemo:Bank/NonExistentAccount
             (capi:display-message "No account exists for
                                   name ~A" name))))))))
```

This callback incorporates a test that prevents the user from being presented with more than one interface to the same account. It invokes the stub method op:retrieveAccount only if the account under that name is not already on display. Because of IDL inheritance, the server implementing the IDL

retrieveAccount operation may return any object reference whose interface inherits from the IDL account interface.

In particular, the server may return an IDL checkingAccount as a special instance of an IDL account. In Common Lisp terms, this means that the stub method <code>op:retrieveAccount</code> may return an object reference of class <code>BankingDemo:checkingAccount</code> as a special instance of <code>BankingDemo:account</code>. The call to <code>make-account-frame</code> dispatches on the actual, or most derived, class of the resulting object reference, making an <code>account-interface</code> or <code>checking-account-interface</code> as appropriate.

The definition of the close-account-callback is straightforward:

The function prompt-with-list presents a dialog asking the user to select a name from the list of available account frames (indexed by their account-name), returning nil if the user decides to cancel the dialog. Given a valid selection, the callback invokes the stub method op:closeAccount on the target object reference, bank-ref, passing the name of the selected account. Finally, the account interface is removed from the bank interface.

#### 5.4.3 Initializing the ORB and obtaining the first object reference

A client can only communicate with a CORBA object if it possesses a reference to that object. This raises the question of how the client obtains its initial object reference. The fact that some IDL operation may return an object reference is of no help here: without a reference to specify as its target, there is no way to invoke this operation.

In more detail, before a client can enter the CORBA environment, it must first:

Be initialized into the ORB

- Get a reference to the ORB pseudo-object for use in future ORB operations
- Get an initial reference to an actual object on the server

CORBA provides a standard set of operations, specified in *pseudo IDL* (PIDL), to initialize applications and obtain the appropriate object references.

Operations providing access to the ORB reside in the CORBA module. (Like an IDL interface declaration, a (P)IDL module declaration defines a new namespace for the body of declarations it encloses. What it does not do is define a new type of CORBA object.) Operations providing access to an Object Adapter, Interface Repository, Naming Service, and other Object Services reside in the ORB interface defined within the CORBA module.

To provide some flavor of PIDL, here is a fragment of the PIDL specification of CORBA that we rely on in our implementation of the bank client.

```
module CORBA {
  interface Object {
   boolean is_a (in string logical_type_id);
  };
  interface ORB {
    string object_to_string (in Object obj);
    Object string to object (in string str);
  };
  typedef string ORBid;
  typedef sequence <string> arg_list;
  ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

The Object interface is implicitly inherited by all IDL interfaces, much as every Common Lisp class inherits from the class standard-object.

The is\_a operation provides a test for inheritance (the logical\_type\_id is a string representation of an interface identifier). The operation returns true if the object is an instance of that interface, including if that interface is an ancestor of the most derived interface of that object.

The ORB operations object\_to\_string and string\_to\_object provide an invertible mapping from object references to their representations as strings. Notice that the CORBA operation <code>ORB\_init</code> is defined outside the scope of any interface, providing a means of bootstrapping into the CORBA world. Calling <code>ORB\_init</code> initializes the ORB, returning an ORB pseudo-object that can be used as the target for further ORB operations.

Like most other language bindings, the Common Lisp binding adopts the pseudo-objects approach in which these CORBA and ORB operations are accessed by applying the binding's normal IDL mapping rules to the PIDL specification.

In this tutorial, we use a very simple technique to obtain the initial object reference. The client assumes that the server has published a reference to its implementation of the bank object, encoded as a string, in a shared file. After starting up, the client reads the file, decodes the string as an object reference, and then uses this reference as the target of further operations.

Here is the remaining Common Lisp code that completes the implementation of the client:

The defparameter \*bank-ior-file\* is the name of the shared file used to pass the reference of the bank object from the server to the client.

The method file-as-string reads a file's contents.

The top-level let statement first initializes the Harlequin Common Lisp ORB by calling the Common Lisp generic function op:ORB\_init corresponding to the PIDL ORB\_init operation. The first argument to this call is an empty list. Passing an empty sequence instructs the op:ORB\_init function to ignore this argument and use the application's command line arguments (if any) instead. The value of the second argument, "Harlequin Common Lisp ORB", merely identifies the ORB to use.

Invoking op:string\_to\_object on this ORB, passing the string read from the shared file, reconstitutes the string as an unspecific object reference of class CORBA:Object. Calling the op:narrow method on this object reference narrows

(that is, coerces) it to a more specific object reference of class BankingDemo:bank. (The op:narrow method employs an implicit call to the object's is\_a operation to check that the desired coercion is safe.)

Finally, the resulting object reference bank-ref, of class BankingDemo:bank, is used to make and start a new bank interface, displaying the initial GUI to the user. The implementation of the client is now complete.

## The Bank Server

#### 6.1 The server

In this chapter, we use Common Lisp to design and implement a CORBA server using the Harlequin Common Lisp ORB.

Our server presents an object-oriented interface to a bank object and its accounts. Because we want the bank's account records to persist beyond the lifetime of the server, we would store the account records in a database. This database could be manipulated by the server using an SQL interface, such as that currently available with the LispWorks product.

Since the primary motivation for this tutorial is to illustrate the use of CORBA, we simply simulate the database using a hash table. It would be fairly easy to replace this implementation with code that manipulates a real database.

The hash table simply uses a structure instance for each row:

(defstruct database-row name balance limit)

In the case of an account that does not allow an overdraft, limit will be nil.

#### 6.2 Implementing CORBA objects on the server

A CORBA server has to provide an implementation object, called a servant, for each of the proxy objects manipulated by the client. Our server needs to implement the initial bank servant, and then create new servants for each of the account objects created in response to openAccount, openCheckingAccount and retrieveAccount requests.

Each of these servants needs to be registered in the CORBA environment and assigned an object reference, so that the ORB can direct incoming requests to the appropriate servant.

In CORBA, the primary means for an object implementation to access ORB services such as object reference generation is via an *object adapter*. The object adapter is responsible for the following functions:

- Generation and interpretation of object references
- Registration of servants
- Mapping object references to the corresponding servants
- IDL method invocations, mediated by skeleton methods
- Servant activation and deactivation

The Harlequin Common Lisp ORB provides an implementation of the Portable Object Adapter (POA). This object adapter forms part of the CORBA standard and, like the ORB, has an interface that is specified in pseudo IDL (PIDL). The Common Lisp interface to the POA conforms closely to the interface obtained by applying the Common Lisp mapping rules to the POA's PIDL specification.

A POA object manages the implementation of a collection of objects, associating object references with specific servants. While the ORB is an abstraction visible to both the client and server, POA objects are visible only to the server.

User-supplied object implementations are registered with a POA and assigned object references. When a client issues a request to perform an operation on such an object reference, the ORB and POA cooperate to determine the servant on which the operation should be invoked, and perform the invocation as an upcall through a skeleton method.

The POA allows several ways of using servants although it does not deal with the issue of starting the server process. Once started, however, there can be a servant started and ended for a single method call, a separate servant for each object, or a shared servant for all instances of a particular object type. It allows for groups of objects to be associated by means of being registered with different instances of the POA object and allows implementations to specify their own activation techniques. If the implementation is not active when an invocation is performed, the POA can start one.

Unfortunately, the flexibility afforded by the POA means that its interface is complex and somewhat difficult to use. The example in this tutorial makes only elementary use of the POA.

Here is the PIDL specification of the facilities of the POA that the bank tutorial uses:

```
module PortableServer {
  native Servant;
  interface POAManager {
    exception AdapterInactive{};
    void activate() raises (...);
    ...
};
  interface POA {
    exception WrongAdapter {};
    readonly attribute POAManager the_POAManager;
    Object servant_to_reference(in Servant p_servant)
        raises (...);
    Servant reference_to_servant(in Object reference)
        raises (WrongAdapter, ...);
    ...
};
...
};
```

The POA-related interfaces are defined in a module separate from the CORBA module, called the PortableServer module. It declares several interfaces, of which only the POA and POAManager are shown here.

The PortableServer module specifies the type Servant. Values of type Servant represent language-specific implementations of CORBA interfaces. Since this type can only be determined by the programming language in question, it is merely declared as a native type in the PIDL.

In the Common Lisp mapping, the servant type maps to the abstract class PortableServer: Servantbase. User defined Common Lisp classes that are meant to implement CORBA objects and be registered with a POA must inherit from this class.

Each POA object has an associated POAManager object. A POA manager encapsulates the processing state of the POA it is associated with. Using operations on the POA manager, an application can cause requests for a POA to be queued or discarded, and can cause the POA to be deactivated.

A POA manager has two main processing states, holding and active, that determine the capabilities of the associated POA and the handling of ORB requests received by that POA. Both the POA manager and its associated POA are initially in the holding state.

When a POA is in the holding state, it simply queues requests received from the ORB without dispatching them to their implementation objects. In the active state, the POA receives and processes requests. Invoking the POA Manager's activate operation causes it, and its associated POAs, to enter the active state.

A POA object provides two useful operations that map between object references and servants. The servant\_to\_reference operation has two behaviors. If the given servant is not already active in the POA, then the POA generates a new object reference for that servant, records the association in the POA, and returns the reference. If the servant is already active in the POA, then the operation merely returns its associated object reference.

The reference\_to\_servant operation returns the servant associated with a given object reference in the POA. If the object reference was not created by this POA, the operation raises the wrongAdapter exception.

#### 6.3 The server's perspective

The bank fasl file corresponding to the bank.idl file (produced when compiling the defsystem) defines a servant class for each of the protocol classes corresponding to an IDL interface. Each of these classes inherits from the abstract class PortableServer: ServantBase, allowing instances of these classes to be registered with a POA.

The user provides an implementation of a servant class by defining a subclass of that class, called an *implementation class*, and defining methods, specialized on this implementation class, for each of the protocol functions corresponding to an IDL attribute or operation.

The bank fasl file defines a concrete skeleton method, specialized on the appropriate servant class, for each protocol function stemming from an IDL attribute or operation. When the POA receives a request from a client through the ORB it looks up the servant targeted by that request, and invokes the corresponding skeleton method on that servant. The skeleton method performs an upcall to the method that implements the protocol function for the implementation class of the servant. If the upcall succeeds, the skeleton method sends the result to the client. If the method raises a Common Lisp condition corresponding to a CORBA user or system exception, the skeleton method sends the CORBA exception back to client.

#### 6.4 Implementing the bank server

#### 6.4.1 Implementing the servant classes

The bank fasl file defines three abstract servant classes

BankingDemo:account-servant,

BankingDemo: checkingAccount-servant, and

BankingDemo:bank-servant corresponding to the IDL interfaces account,

checkingAccount, and bank. The class

BankingDemo: checkingAccount-servant is defined to inherit from BankingDemo: account-servant, matching the inheritance relationship in the IDL.

Note that each class inherits from the abstract class PortableServer: ServantBase, allowing instances of the class to be registered with a POA.

In our implementation of the bank server, these servant classes are implemented by the subclasses

- bank-implementation,
- account-implementation, and

checkingAccount-implementation

The bank-implementation class implements BankingDemo:bank-servant by representing a bank as a connection to a database:

```
(defclass bank-implementation (BankingDemo:Bank-servant)
  ((connection :initarg :connection)
   (poa :initarg :poa)
   (account-impls :initform nil)))
```

We have included the poa slot to record the POA in which the bank servant is active, so that servants representing accounts at the bank can be registered in the same POA. A slot op:name corresponding to the attribute name defined in the IDL is inherited from the Bank-servant, as are definitions of accessor functions for this slot.

The account-implementation class implements BankingDemo:account-servant:

```
(defclass account-implementation (BankingDemo:Account-servant)
  ((bank:initarg:bank)))
```

An instance of this class represents an account. The bank slot provides a connection to the database that holds the account's record. Slots op:name and op:balance, corresponding to attributes defined in the IDL, are inherited from account-servant. The name slot identifies the record in the database.

Finally, the checkingAccount-implementation class implements BankingDemo: checkingAccount-servant simply by inheriting from account-implementation:

```
(defclass checkingaccount-implementation
  (Bankingdemo:Checkingaccount-servant account-implementation)
())
```

A slot op:limit, corresponding to the attribute limit defined in the IDL, is inherited from checkingaccount-servant.

#### 6.4.2 Implementing the servant methods

The next step in implementing the server is to define methods, specialized on the implementation classes, for each of the protocol functions corresponding to an IDL attribute or operation. Implementing a protocol function boils down to defining a concrete method for that function that specializes on the implementation class of its target object. Recall that the target object of a protocol function is the first parameter to that function.

We can now present the implementations of the protocol functions. The op:name method corresponding to the name attribute is automatically generated by the IDL compiler to reference a slot op:name in the class that takes the initary :name to initialize it. The same rules apply to op:balance.

The op:credit method on an Account increments the record's balance field by executing a database update statement:

The op:debit method on an Account executes a database update statement that decrements the record's balance field, provided the balance exceeds the desired amount:

The op:limit method is automatically generated, as it is also an attribute.

Because we defined checkingAccount-servant to inherit from account-servant, there is no need to re-implement the op:credit method for this implementation class. However, we do want to define a specialized op:debit method on checkingAccount, to reflect that a checking account can be overdrawn up to its limit:

```
(corba:define-method op:debit (
                    (self checkingAccount-implementation) amount)
 (with-slots (op:name bank op:balance) self
    (with-slots (connection) bank
      (let ((old-balance (lookup-row-value op:name
                          connection :balance))
            (limit (lookup-row-value op:name connection :limit)))
        (if (< (+ old-balance limit) amount)</pre>
            (error 'BankingDemo:Account/Refusal
                  :reason (format nil "Can't debit ~A because the
                                    balance is ~A (limit is ~A)."
                                   amount old-balance limit))
          (update-database-row op:name connection
                               :balance (setf op:balance (-
                                  old-balance amount)))))))
```

The BankingDemo bank op:name method returns the value of the bank's op:name slot and is automatically generated.

The op:openAccount method on Bank illustrates the raising of CORBA user exceptions:

```
(corba:define-method op:openAccount ((self bank-implementation)
                                      name)
 (with-slots (connection poa account-impls) self
    (when (find-database-row name connection)
      (error 'Bankingdemo:Bank/Duplicateaccount))
    (create-database-row name connection)
    (update-database-row name connection :balance 0)
    (let ((new-account (make-instance 'account-implementation
                                      :name name
                                      :bank self
                                      :balance 0)))
      (push new-account account-impls)
      (op:narrow 'BankingDemo:Account
                 (op:servant_to_reference poa new-account)))))
```

If the (find-database-row name connection) test succeeds, the call to (error 'Bankingdemo: Bank/Duplicateaccount) raises a Common Lisp condition. (We omit the definition of find-database-row, which can be found in the source.) Recall that the condition class

BankingDemo:bank/duplicateAccount corresponds to the IDL

duplicateAccount exception. The POA that invoked this method in response
to a client's request will catch the condition and send the duplicateAccount
exception back to the client. If there is no existing account for the supplied
name, the op:openAccount method creates a new record in the database.

Finally, the method makes a new servant of class account-implementation, registers it with the bank's POA with a call to op:servant\_to\_reference, and narrows the resulting object reference to the more specific class

BankingDemo:account, the class of object references to account objects.

The op:openCheckingAccount method is similar, except that it initializes the op:limit field of the new account record with the desired overdraft limit and registers a new servant of class checkingAccount-implementation, returning an object reference of class BankingDemo:checkingAccount:

```
(corba:define-method op:openCheckingAccount (
                           (self bank-implementation) name limit)
 (with-slots (connection poa account-impls) self
   (when (find-database-row name connection)
     (error 'Bankingdemo:Bank/Duplicateaccount))
   (create-database-row name connection)
   (update-database-row name connection :balance 0 :limit limit)
   (let ((new-account (make-instance
                         'checkingaccount-implementation
                                      :name name
                                      :bank self
                                      :balance 0
                                      :limit limit)))
      (push new-account account-impls)
      (op:narrow 'Bankingdemo:Checkingaccount
                 (op:servant_to_reference poa new-account)))))
```

The op:retrieveAccount method uses the name parameter to find a database row of the given name. If the query returns nil, indicating that there is no record with that name, the method raises the CORBA user exception nonExistentAccount by signalling the corresponding Common Lisp error.

Otherwise, the method uses the value of the op:limit field to distinguish whether the account is an account or a current account, creating a new servant of the appropriate class:

```
(corba:define-method op:retrieveAccount ((self
                                       bank-implementation) name)
 (with-slots (connection poa account-impls) self
   (unless (find-database-row name connection)
      (error 'Bankingdemo:Bank/NonExistentAccount))
   (let ((limit (lookup-row-value name connection :limit))
          (balance (lookup-row-value name connection :balance)))
     (if (not limit)
          (let ((account (make-instance 'account-implementation
                        :name name :bank self :balance balance)))
            (push account account-impls)
            (op:narrow 'BankingDemo:Account
                       (op:servant_to_reference
                        account)))
       (let ((account (make-instance
                         'checkingaccount-implementation
                                      :name name
                                      :bank self
                                      :balance balance
                                      :limit limit)))
          (push account account-impls)
          (op:narrow 'Bankingdemo:Checkingaccount
                     (op:servant_to_reference
                      account)))))))
```

Finally, the closeAccount removes the record of an account from the database by executing delete-database-row call:

```
(corba:define-method op:closeaccount
    ((self bank-implementation) account)
  (with-slots (connection poa account-impls) self
    (let ((servant (op:reference_to_servant poa account)))
      (op:deactivate_object poa (op:reference_to_id poa account))
      (removef account-impls servant)
      (with-slots (op:name) servant
         (delete-database-row op:name connection))))
```

Note that we need to de-reference the object reference account that is passed in as the parameter of the op:closeAccount operation, using a call to the op:reference\_to\_servant operation of the POA.

Here, we make implicit use of our knowledge that, in our application, the server only encounters object references registered with its local POA. This assumption is not true in general.

## 6.4.3 Obtaining the initial POA object and registering the first object reference

To complete the implementation of the server we need to write the code that enters it into the CORBA environment. In detail, we need to:

- Initialize the server's ORB
- Get a reference to the ORB pseudo-object for use in future ORB operations
- Get a reference to the root POA pseudo-object for use in future POA operations
- Make a bank servant and register it with the POA
- Make the object reference of the bank servant available to the client
- Activate the POA to start processing incoming requests

To do this, we need to make use of some additional operations specified in the CORBA module:

```
module CORBA {
    ...
    interface ORB {
        ...
        typedef string ObjectId;
        exception InvalidName {};
        Object resolve_initial_references (in ObjectId identifier)
            raises (InvalidName);
        void shutdown( in boolean wait_for_completion );
    }
}
```

The CORBA standard specifies the ORB operation

resolve\_initial\_references. This operation provides a portable method for applications to obtain initial references to a small set of standard objects (objects other than the initial ORB). These objects are identified by a mnemonic name, using a string known as an <code>objectid</code>. For instance, the <code>objectID</code> for an initial POA object is <code>RootPOA</code>. (References to a select few other objects, such as the <code>InterfaceRepository</code> and <code>NamingService</code>, can also be obtained in this manner.)

The ORB operation resolve\_initial\_references returns the object associated with an objectid, raising the exception invalidname for an unrecognized ObjectID.

Meanwhile, the shutdown operation instructs the ORB, and its object adapters, to shut down. If the wait\_for\_completion parameter is TRUE, the operation blocks until all pending ORB processing has completed, otherwise it simply shuts down the ORB immediately.

```
(defun bank-server ()
  (let* ((orb (op:orb_init nil "Harlequin Common Lisp ORB"))
         (rootPOA (op:resolve initial references orb "RootPOA")))
    (let ((bank-impl (make-instance 'bank-implementation
                                :connection (connect-to-database)
                                   :poa rootPOA)))
      (let ((bank-ref (op:servant_to_reference rootPOA
               bank-impl)))
        (object-to-file orb bank-ref)
        (capi:display (make-instance 'server-controller
                                     :bank-poa rootPOA
                                     :bank-ref bank-ref)))
      (op:activate (op:the_poamanager rootPOA)))))
```

The top-level function first initializes the Harlequin Common Lisp ORB by calling the Common Lisp generic function op:orB\_init, just as we initialized the ORB in the client.

The call returns an ORB pseudo-object. Invoking op:resolve\_initial\_references on this ORB, passing the ObjectID RootPOA, returns a POA object of class PortableServer: POA. This is the CORBA standard method for obtaining the initial POA object. Note that root POA is initially in the holding state.

Next, we connect to the database and use the connection to make a bank servant. We register the servant with the POA, ROOLPOA, and publish the resulting object reference, encoded as a string, in the shared file.

We then obtain the POA Manager for the POA using the POA operation op:the\_POAManager. The call to op:activate moves the POA out of the holding state, into the active state, ready to receive and process incoming requests.

This completes the description of our implementation of the server.

### 7

# The Harlequin Common Lisp ORB

#### 7.1 The CORBA modules

The Common Lisp CORBA product is made up of the following four modules, which can be loaded into the image using require.

corba-support The client side of the ORB runtime plus the IDL parser.

corba-orb The server side, the actual Harlequin Lisp ORB, which

loads corba-support if it is not present.

A place holder for a number of graphical tools. (On

Liquid Common Lisp this requires common-lispworks.)

corba-orbix-support

A number of patches to support OrbixWeb.

#### 7.2 Parsing IDL into stubs and skeletons

The interface to the IDL parser is currently via defsystem. Once the corba-support module has been loaded, defsystem will have been extended with the following extra types:

```
:idl-file
:idl-client-definition
:idl-server-definition
```

The :idl-file type instructs the IDL parser to generate code for both the client-side stubs and the server-side skeletons. The :idl-client-definition type instructs the IDL parser to generate code only for the client-side stubs. The :idl-server-definition type instructs the IDL parser to generate code only for the server-side skeletons.

For example, a typical use would be something like:

```
(defsystem server-side ()
  :members (
            ("grid" :type :idl-file)
            "grid-impl"
  :rules
  ((:in-order-to :compile :all
    (:requires (:load :previous)))))
(defsystem client-side ()
  :members (
            ("grid" :type :idl-client-definition)))
```

#### 7.3 Configuring a name service and an interface repository

The Harlequin Common Lisp Orb is not supplied with an interface repository or a naming service, but functions are supplied that allow the user to configure the ORB to present these services in response to calls of op:list initial services.

The function corba:set-pluggable-module-details can be used for this purpose. For example,

```
(corba:set-pluggable-module-details "InterfaceRepository"
       :ior-string "IOR:00000000000002149444c...")
(corba:set-pluggable-module-details "NameService"
       :ior-string "IOR:00000000000002149444c...")
```

would direct the ORB to use the given IOR-described CORBA object as an interface repository or name service, respectively. Values set in this manner are persistent. The code

```
(corba:get-pluggable-module-details "InterfaceRepository"
    :ior-string)
```

returns a list of two values. The first is the object reference of the given service if it is contactable (nil if it is not). The second is the value that has been set using <code>corba:set-pluggable-module-details</code>.

Type information corresponding to IDL parsed by the IDL compiler is not added to the interface repository until the information is needed in response to a call to get interface from a client.

#### 7.4 Error handling in user code called by the server

Error handlers are wrapped around user code called by the POA in response to incoming CORBA requests. By default, an error is handled and a CORBA exception is returned to the client.

If the variable corba:\*orb-status-output\* is set to a non-nil value, a back-trace is printed to the given stream designator.

If the variable corba: \*poa-break-on-error-in-server\* is set to a non-nil value, the debugger will be entered. When the user opts to continue, a CORBA exception will be returned to the client. Note that while debugging is going on, a synchronous call from a client will be hanging, waiting for a response.

7 The Harlequin Common Lisp ORB

# Appendix A

# Common Lisp IDL Binding

Version 1.0 (The RFP for the IDL Common Lisp mapping was agreed by the ORBOS and PTC committees at the OMG meeting in Washington on 99/01/14.)

This chapter briefly reviews some concepts of IDL and defines the notion of a language mapping. A summary of the IDL/Common Lisp mapping is presented.

#### A.1 Introduction to IDL

IDL, or Interface Definition Language, is a language defined by the Object Management Group. The key data type in IDL is the interface, which describes the behavior of objects that implement that interface. The IDL definition for an interface describes all of the operations to which an object that implements that interface can respond. For each such operation, it describes the allowed types of the parameters to the operation and the allowed type of the value returned by the operation.

IDL allows types other than interfaces to be expressed. For example, primitive types such as boolean, several signed and unsigned integer types, and some floating point types may be defined.

Constructed types analogous to the C struct or Pascal record type may be defined, and some simple type aliasing is possible in a way analogous to the C typedef construct. Arrays and sequences may also be defined.

#### A.2 How IDL is used

IDL is typically used in the following manner. A server process wishes to make some of its functionality available for invocation by clients. These clients may not be in the same process, on the same machine, or even written in the same language.

The server publishes the IDL definitions that define the interfaces of the objects that it implements. A client can use those definitions to invoke operations on objects that reside within the server process.

The syntax used by the client to invoke a method on an object defined in IDL, and the relationship between the data types specified in IDL and the native datatypes of the language in which the client is implemented is defined by the *mapping* of IDL into that language.

This document describes a mapping from IDL into Common Lisp.

#### A.3 Mapping summary

The main points of the mapping from IDL to Common Lisp are as follows:

- Primitive data types are mapped to corresponding primitive data types in Lisp.
- Struct and union are mapped to classes. Each member of the struct or union can be accessed using a regular syntax.
- Arrays map to arrays.
- Sequences can map either to lists or to vectors; that is, sequences map to sequences.
- Exceptions are mapped to conditions.
- Interfaces are mapped to classes, and interfaces that inherit map to classes that inherit.

- Operations on interfaces map to methods on a generic function. This
  generic function discriminates only on its first argument, which is then
  interpreted as the receiver of the operation.
- The module in which an IDL entity is declared is mapped to the package name of the corresponding symbol. The name of the symbol is formed from the rest of the scope of the module.

#### A.4 Mapping in more detail

This section describes the mapping of IDL into the Lisp language. In most cases examples of the mapping are provided. It should be noted that the examples are code *fragments* that try to illustrate only the language construct being described.

#### A.4.1 Mapping concepts

By an IDL entity we mean an element defined in some IDL file. For example, consider the code fragment:

```
module A {
  interface B {
    void op1(in long bar);
  };
}
```

The IDL entities are the module named A, the interface named B, the operation named op1, the formal parameter named bar, and the primitive data types void and long.

Our mapping will associate a corresponding Lisp entity to each IDL entity declared in a an IDL specification. The Lisp entity corresponding to a given IDL entity will be said to be *generated* from the IDL entity.

If the IDL entity has a name, then the corresponding Lisp entity will also have a name. Whereas IDL entities are named by strings (in other words, identifiers), Lisp entities are named by symbols.

This chapter specifies, for each IDL construct, the Lisp entity, and the name of that entity, that is generated by the mapping.

#### A.4.2 Semantics of type mapping

The statement that an IDL type I is mapped to a Lisp type L indicates that if V is a Lisp value whose corresponding IDL type is I, then the consequences are not specified if the value of V is not a member of the type L. For example, if V is passed as a parameter to an IDL operation or if V is returned from an IDL operation, then a conforming implementation may reasonably perform any of the following actions if V is not of the type L.

- If *V* may be coerced to *L*, then *V* may be replaced by the result of coercing *V* to the type *L*.
- If V cannot be coerced to L, then an error may be signalled. If the error
  occurs during marshalling or unmarshalling, corba:marshal should be
  signaled.

#### A.4.3 Mapping for basic types

The following table shows the basic mapping. The first column contains the IDL name of the IDL type to be mapped. Each IDL type denotes a set of IDL abstract values.

The set of values denoted by an entry in the first column is mapped, under the mapping described in this document, to a set of Lisp values. That set of Lisp values is described in two ways:

IDL Type	Name of Lisp Type	Lisp Type Specifier
boolean	corba:boolean	boolean
char	corba:char	character
octet	corba:octet	(unsigned-byte 8)
string	corba:string	string
short	corba:short	(signed-byte 16)
unsigned short	corba:ushort	(unsigned-byte 16)
long	corba:long	(signed-byte 32)

Table A.1

IDL Type	Name of Lisp Type	Lisp Type Specifier
unsigned long	corba:ulong	(unsigned-byte 32)
float	corba:float	see text
double	corba:double	see text

Table A.1

For example:

```
(typep -3 'corba:short)
> T
(typep "A string" 'corba:string)
> T
```

#### A.4.3.1 boolean

The IDL boolean constants TRUE and FALSE are mapped to the corresponding Lisp boolean literals T and nil. The type specifier corba:boolean specifies this type.

#### A.4.3.2 char

IDL char maps to the Lisp type character. The type specifier corba:char specifies this type.

For example:

```
(typep #\x corba:char)
> T
(typep "x" 'corba:char)
> nil
```

#### A.4.3.3 octet

The IDL type octet, an 8-bit quantity, is mapped as an unsigned quantity to the type corba:octet. The type specifier corba:octet denotes the set of integers between 0 and 255, inclusive. This set can also be denoted by the type specifier (unsigned-byte 8).

For example:

```
(typep 255 'corba:octet)
> T
(typep -1 'corba:octet)
> nil
```

#### **A.4.3.4** string

The IDL string, both bounded and unbounded variants, are mapped to string. Range checking for characters in the string as well as bounds checking of the string shall be done at marshal time. The type specifier corba:string denotes the set of Lisp strings.

For example:

```
(typep "A string" 'corba:string)
> T
(typep nil 'corba:string)
> nil
```

#### A.4.3.5 Integer types

The integer types each map to the Lisp integer type. Each IDL integer type has a corresponding type specifier that denotes the range of integers to which it corresponds.

The names of the type specifiers are corba:long, corba:short, corba:ulong, and corba:ushort.

#### A.4.3.6 Floating point types

The floating point types float and double map to Lisp types named corba:float and corba:double, respectively. These types must be subtypes of the type real. They must allow representation of all numbers specified by the corresponding CORBA types.

#### A.4.4 Introduction to named types

We now discuss the mapping of types that are named. We begin with a discussion of terminological issues.

#### A.4.4.1 IDL naming terminology

Notation for naming can be confusing, so some care is needed. Our specification is not formally rigorous, but we have tried to illustrate enough points with examples so that situations likely to arise in practice can be handled.

#### **IDL Naming Terminology**

By "the IDL name of an IDL entity", we mean the string that is the simple name of that entity. An IDL entity can be declared at the top-level or nested inside some other IDL entity. We say that the outer IDL entity encloses the inner one. We will sometimes elide the quotation marks in describing the names of IDL (and other entities) when no confusion is likely to result.

Here is an IDL example:

```
module A{
  interface B{
    struct c {long foo;};};}
```

The name of the struct is the string c. The name of the interface is the string B. The name of the module is the string A. The name of the struct member is the string foo. The innermost enclosing IDL entity of the struct is the interface named B. The innermost enclosing module of the struct is the module named A.

#### **Lisp Naming Terminology**

The name of a symbol is a string used to identify the symbol. Packages are collections of symbols. A symbol has a home package, which also has a name. A package can be named by a symbol or a string. We sometimes loosely say "the package x" when we mean "the package named by x".

A package may have nicknames, and we will consider that the nicknames of a package name the package. Unless otherwise stated, we will assume that distinct package names refer to distinct packages.

The notation for symbols consists of three concatenated parts: the name of the home package of the symbol, followed by the character ":", followed by the name of the symbol. Case is not significant when this notation is used. Thus, all symbols generated by this mapping are external symbols of their home package.

A symbol can name a function, a package, a class, a type, a slot, or a variable. These namespaces are disjoint. All alphabetic characters in the names of symbols used in this document are upper-case unless otherwise stated.

Thus, the names notated here are implicitly converted to uppercase when they name a symbol. For example, when we write the symbol named hello-goodbye or the symbol hello-goodbye, we actually mean the symbol whose name is the string Hello-goodbye.

#### A.4.5 Distinguished packages

This document will refer to two kinds of packages:

- The first kind are those packages defined explicitly by this specification (this Appendix).
- The second kind consists of those packages created as a result of compiling user IDL code.

The first kind of package consists of these three distinct packages: the root package, the corba package, and the operation package.

The names of these packages are described below. The name of the root package is the string "OMG.ORG/ROOT". The name of the corba package is "OMG.ORG/CORBA". The name of the operation package is the string "OMG.ORG/OPERATION".

The precise semantics of these three packages is described below. Informally, the root package is the package in which Common Lisp names corresponding to IDL definitions not contained in a top-level module are interned. The corba package is the package in which Common Lisp names corresponding to IDL definitions and pseudo-IDL definitions in the CORBA module are interned. The operation package is the package into which names of Common Lisp functions corresponding to IDL operations are interned.

In addition, this specification makes use of the standard Common Lisp packages named KEYWORD and COMMON-LISP.

## A.4.5.1 Nicknames for distinguished packages

An implementation is expected to support the addition of nicknames for a package via the standard Common Lisp nicknames facility. An ORB should support the following default nicknames:

- For the package omg.org/corba, the default nickname shall be corba.
- For the package omg.org/operation, the default nickname shall be op.

This document will use these nicknames without comment.

# A.4.6 Scoped names and scoped symbols

Many of the Common Lisp entities we consider will be named according to the scoped naming convention described in this section. In particular, the following entities will be mapped according to this naming convention:

- interface
- union
- enum
- struct
- exception
- const
- typedef

A scoped symbol will be associated with the IDL entity, and it is this scoped symbol that names the Lisp value generated by the given IDL entity.

#### A.4.6.1 Definitions

For any named IDL entity I there is a Lisp symbol S called the scoped symbol of I. The scoping separator is the string "/".

If *I* is a top-level module, then the name of *S* is the name of *I*.

If I is a module nested within another module J, then the name of S is the concatenation of the name of the scoped symbol of J, the scoping separator, and the name of I. The home package of the scoped symbol of a module is :keyword.

Suppose I is a named IDL entity that is not a module. The name of the scoping symbol S of I is determined as follows. If the declaration of I is enclosed inside another IDL entity J that is not a module, then the name of S is the concatenation of the name of the scoping symbol for J, the scoping separator, and the name of I. Otherwise the name of S is the name of I.

If I is enclosed in a module M, then the home package of S is named by the scoped symbol for M. Otherwise the home package for S is the root package.

# A.4.6.2 Examples of scoping symbols

First we consider a simple example:

```
module a { interface foo {};}
```

The scoped symbol of the module is :a. Thus, the home package of this symbol is :keyword and the name of the symbol is the string A. The scoped symbol of the interface is the symbol a:foo. Thus, the name of the symbol is the string foo, and the home package of the symbol is the package whose name is the string A.

```
module a {
  interface outer {
    struct inner {
       in long member;
    };
    };
}
```

Here the scoped symbol for the module is :a, the scoped symbol for the interface is a:outer, and the scoped symbol for struct is a:outer/inner.

```
module a{
  module b{
    interface c{
      struct d{
        long foo;
      };
    };
};
```

The scoped symbol for the struct is a/b:c/d. The scoped symbol for the struct member is a/b:c/d/foo.

# A.4.7 The package\_prefix pragma

A package\_prefix pragma has the form:

```
#pragma package_prefix string
```

where string is an IDL string literal. For example, #pragma package\_prefix COM.HARLEQUIN.

A package\_prefix pragma affects the mapping of all top-level modules whose definition textually follows that pragma in the IDL file. The name of the scoping symbol for such a top-level module is the concatenation of the given package\_prefix with the name of the module.

```
#pragma package_prefix COM/HARLEQUIN
module a{
  module b{
   interface c{};
  };
};
```

The scoped symbol for the interface is COM/HARLEQUIN/A/B:C.

# A.4.8 Mapping for interface

An IDL interface is mapped to a Lisp class. The name of this class is the scoped symbol for the interface. The direct superclasses of a generated Lisp class are determined as follows.

If the given IDL interface has no declared base interfaces, the generated class has the single direct superclass named <code>corba:object</code>. Otherwise, the generated Lisp class has direct superclasses that are the generated classes corresponding to the declared base interfaces of the given interface. The Lisp value <code>nil</code> can be passed wherever an object reference is expected.

An IDL interface is also mapped into server-side classes. The server classes are described in Section A.6 on page 91.

For example, in IDL:

```
module example{
    interface foo {};
    interface bar {};
    interface fum : foo,bar {};
}

And in generated Lisp:

    (defclass example:foo(corba:object)())
    (defclass example:bar(corba:object)())
    (defclass example:fum (example:foo example:bar)())
```

## A.4.9 Mapping for operation

This section discusses only how the user is to invoke mapped operations, not how the user is to implement them. The implementation of operations is discussed in Section A.6 on page 91.

An IDL operation is mapped to a Lisp function named by the symbol whose print-name is given by the name of the operation interned in the operation package.

We will assume that all operation names have been appropriately imported into the current package in the examples.

Thus, when an example is given in which there is a reference to the symbol naming the mapped function corresponding to an IDL operation, the package of that symbol will be assumed to be the operation package. Common Lisp provides a number of facilities for the implementation of this functionality and for handling name conflicts.

# A.4.9.1 Parameter passing modes

The function defined by the IDL operation expects actual arguments corresponding to each formal argument that is declared in or inout, in the order in which they are declared in the IDL definition of the operation.

#### A.4.9.2 Return values

The function defined by the IDL operation returns multiple values. The first value returned is that value corresponding to the declared return value, unless the declared return value is void. Following the value corresponding to the declared return value, if any, the succeeding returned values correspond to the parameters that were declared out and inout, in the order in which those parameters were declared in the IDL declaration.

Note that this implies that generated functions corresponding to operations declared void, which have neither out nor inout formal parameters, return zero values.

# **A.4.9.3** oneway

Operations declared oneway are mapped according to the above rules.

# A.4.9.4 Efficiency optimization: Using macros instead of functions

A conforming implementation may map an operation to a macro whose name and invocation syntax are consistent with the above mapping. For the sake of terminological simplicity, however, this document will continue to refer to mapped operations as "functions".

## A.4.9.5 exception

An invocation of a function corresponding to a given IDL operation may result in the certain conditions being signalled, including the conditions generated by the exceptions declared in the raises clause of the operation, if any. Such conditions are signalled in the dynamic environment of the caller.

An invocation of a function may also result in the signalling of conditions corresponding to system exceptions.

#### A.4.9.6 context

If the operation is specified to take a context (using the IDL context clause), the generated operation takes an extra optional parameter corresponding to a context object generated using the normal IDL context manipulation operations.

For example, in IDL:

```
module example {
       interface face {
         long sample_method (in long arg);
         void voidmethod();
         void voidmethod2(out short arg);
         string method3 (out short arg1, inout string arg2, in boolean
     arg3);
       };
In generated Lisp:
     (defpackage :example)
     (defclass example:face (corba:object)())
And in use:
     ; Suppose x is bound to a value of class example: face.
     (sample_method x 3)
     > 24
     (voidmethod x)
     > ; No values returned
     (voidmethod2 x)
     > 905; This is the value corresponding to the out arg
     (method3 x "Argument corresponding to arg2" T)
     > "The values returned" -23 "New arg2 value"
     ; The Lisp construct multiple-value-bind can also be used
     ; to recover these values.
     (multiple-value-bind (result arg1 arg2)
        (method3 x "Argument corresponding to arg2" T)
     (list result arg1 arg2))
     > ("The values returned" -23 "New arg2 value")
```

# A.4.10 Mapping for attribute

An attribute is mapped using a naming convention similar to that for operation.

# A.4.10.1 readonly attribute

An attribute that is declared with the readonly modifier is mapped to a method whose name is the name of the given attribute and whose home package is the operation package.

This method is specialized on the class corresponding to the IDL interface in which the attribute is defined.

#### A.4.10.2 normal attribute

Attributes that are not declared readonly are mapped to a pair of methods that follow the convention used for default slot accessors generated by defclass. Specifically, a reader-method is defined whose name follows the convention for readonly attributes. A writer is defined whose name is (setf name) where name is the name of the defined reader-method.

For example, in IDL:

```
module example{
  interface attributes {
    attribute string attr1;
    readonly attribute long attr2;};}
```

And in use:

```
;; Assume x is bound to an object of class example:attributes
(attr2 x)
> 40001
(attr1 x)
> "Sample"
(setf (attr1 x) "New value")
> "New value"
(attr1 x)
> "New value"
```

# A.4.11 Mapping of module

An IDL module is mapped to a Lisp package whose name is the name of the scoped symbol for that module.

For example, in IDL:

```
interface outer_interface {};
module example {
  interface inner_interface {};
  module nested_inner_example {...
   interface nested_inner_interface{};
   module doubly_nested_inner_example{...};
  };
}
```

And in generated Lisp:

```
(defpackage :example)
(defpackage :example/nested_inner_example)
(defpackage
   :example/nested_inner_example/doubly_nested_inner_example)
(defclass omg.root:outer_interface...)
(defclass example:inner_interface ...)
(defclass example/nested_inner_example:nested_inner_interface...)
```

# A.4.12 Mapping for enum

An IDL enum is mapped to a Lisp type whose name is the corresponding scoped symbol.

Each member of the enum is mapped to a symbol with the same name as that member whose home package is the keyword package.

For example, in IDL:

And in use:

```
(typep :goodbye 'example:foo)
> T
(typep :not-a-member 'example:foo)
> nil
```

# A.4.13 Mapping for struct

An IDL struct is mapped to a Lisp type whose name is the corresponding scoped symbol. Each member of the struct is mapped to an initialization keyword, a reader, and a writer. The initialization keyword is a symbol whose name is the name of the member and whose package is the keyword package.

The reader is named by a symbol that follows the conventions for attribute accessors. In the case of a reader, its package is the operation package, and its name is the name of the member. The writer is formed by using setf on the generalized place named by the reader.

The type corba:struct is supertype of all such generated types.

An IDL struct has a corresponding constructor whose name is the same as the name of the mapped Lisp type. This constructor takes keyword arguments whose package is the keyword package and whose name equals the name of the corresponding member.

For example, in IDL:

```
module structmodule{
    struct struct_type {
        long field1;
        string field2;
    };
};

In generated Lisp:
    (defpackage :structmodule)
    (defstruct structmodule:struct_type ...)

And a usage example:
    (setq struct (structmodule:struct_type :field1 100000
        :field2 "The value of field2"))
```

```
(op:field1 struct)
> 100000
(setf (op:field1 struct) -500)
> -500
(op:field1 struct)
> -500
```

# A.4.14 Mapping for union

An IDL union is mapped to a Lisp type named by the corresponding scoped symbol. This type is a subtype of corba:union.

The value of the discriminator can be accessed using the accessor function named union-discriminator whose home package is the operation package and using an initialization argument named :union-discriminator.

The value can be accessed using the accessor function named union-value in the operation package with initialization argument :union-value.

An IDL union has a corresponding constructor whose name is the same as the name of the type. This constructor takes two constructors whose names are :union-value and :union-discriminator.

#### A.4.14.1 Member accessors

Each union member has an associated constructor and accessor.

The symbol-name of the name of the constructor corresponding to a particular member is the concatenation of the name of the union constructor to the scoping separator to the name of the member. The home package of the name of the constructor corresponding to a particular member is the home package of the name of the union constructor. A constructor corresponding to a member takes a single argument, the value of the union. The discriminator is set to the value of the first case label corresponding to that member.

It is an error if a member reader is invoked on a union whose discriminator value is not legal for that member. The member writer sets the discriminator value to the first case label corresponding to that member.

The default member is treated as if it were a member named default whose case labels include all legal case labels that are not case labels of other members in the union.

For example, in IDL:

```
module example {
       enum enum_type {first,second,third,fourth,fifth};
       union union_type switch (enum_type) {
         case first: long win;
         case second: short place;
         case third:
         case fourth: octet show;
         default: boolean other;
       };
     };
In generated Lisp:
     (defpackage :example)
     (defstruct (example:union_type ...))
And in use:
     (setq union
       (example:union_type
          :union-discriminator :first
          :union-value -100000))
     (op:union-value union)
     > -100000
     (op:union-discriminator union)
     > :FIRST
     (setq same-union (example:union_type/win -100000))
     (op:union-discriminator same-union)
     > :FIRST
     (setf (op:show same-union) 3)
     (op:union-discriminator same-union)
     > :THIRD
     (op:show same-union)
     > 3
```

```
(setf (op:default same-union) nil)
> nil
(op:union-discriminator same-union)
> :FIFTH
```

# A.4.15 Mapping for const

An IDL const is mapped to a Lisp constant whose name is the scoped symbol corresponding to that const and whose value is the mapped version of the corresponding value.

For example, in IDL:

```
module example {
    const long constant = 321;
};

And in generated Lisp:
    (defpackage :example)
    (defconstant example:constant 321)
```

# A.4.16 Mapping for array

An IDL array is mapped to a Lisp array of the same rank. The element type of the mapped array must be a supertype of the Lisp type into which the element type of the IDL array is mapped.

Multidimensional IDL arrays are mapped to multidimensional Lisp arrays of the same dimensions.

For example, in IDL:

```
module example {
  typedef short array1[2];
  interface array_interface{
    array1 op();
  }
}
```

In generated Lisp:

```
(defpackage :example)
(deftype example:array1 () '(array t (2)))
```

```
;; mapping for the interface...
  (defclass example:array_interface...)
And in use:
    (setq a2 (op x)); Get an array
    (aref a2 1); Access an element
    > 3; Just an example, could be any value that is a short
```

# A.4.17 Mapping for sequence

An IDL sequence is mapped to a Lisp sequence. Bounds checking shall be done on bounded sequences when they are marshaled as parameters to IDL operations. An implementation is free to specify the type of the mapped list more specifically.

Suppose *foo* is an IDL data type and let L be the corresponding Lisp type. This means that anywhere a parameter of type sequence<foo> is expected, either a vector (all of whose elements are of type L) or a list (all of whose elements are of type L) may be passed.

Conversely, when such a sequence is returned from an operation invocation, the Harlequin Common Lisp ORB will always return a value of type vector.

For example, in IDL:

```
; Let x be an object of type example:seq
(param_is_valid x '(-2 3))
> T
(param_is_valid x #(-200 33))
>T
```

# A.4.18 Mapping for exception

Each IDL exception is mapped to a Lisp condition whose name is the scoped symbol for that exception. User exceptions inherit from a condition named corba:userexception. And exception is a subclass of serious-condition.

System exceptions inherit from a condition named corba: systemexception.

Both corba:userexception and corba:systemexception inherit from the condition corba:exception.

## A.4.19 User exception

The reader functions and initialization arguments for a condition generated by an IDL exception follow the convention for the mapping of IDL structs. For example:

```
module example {
  exception ex1 { string reason; };
};

; generated Lisp
(defpackage :example)
(define-condition example:ex1 (corba:userexception)
  ((reason :initarg :reason ...)))

; Usage example
(error (example:ex1 :reason "Example of condition"))
```

# A.4.20 System exception

The standard IDL system exceptions are mapped to Lisp conditions that are subclasses of corba:systemexception. Such generated conditions have reader-functions and initargs consistent with the IDL definition of these exceptions.

# A.4.21 Mapping for typedef

IDL typedef is mapped to a Lisp type whose name is the scoped symbol corresponding to that typedef. This name of this type denotes the set of Lisp values that correspond to the Lisp type that is generated by the mapping of the IDL type to which the typedef corresponds.

However, it is not required to perform recursive checking of the contents of constructed types like array, sequence, and struct.

For example, in IDL:

```
module example{
    typedef unsigned long foo;
    typedef string bar;
};

In generated Lisp:
    (defpackage :example)
    (deftype example:foo () 'corba:ulong)
    (deftype example:bar() 'corba:string)

And in use:
    (typep -3 'example:foo)
    > nil
    (typep 6000 'example:bar)
    > nil
    (typep "hello" 'example:bar)
    >T
```

# A.4.22 Mapping for "any"

The IDL type any represents an IDL entity with an associated typecode and value. It is mapped to the type corba: any, which encompasses all Lisp values with a corresponding typecode.

#### A.4.23 Constructors

The constructor corba: any takes two keyword arguments named any-value and any-typecode. If any-typecode is specified, then any-value must be specified. If any-value and any-typecode are each specified, then any-value must be a member of the type denoted by any-typecode.

An any may also be created with the invocation:

```
(corba:any :any-typecode val :any-value type)
```

# A.4.24 The deduced typecode

The actual typecode of a Lisp value *v* is defined as follows:

- If the value is an integer, then corba:\_tc\_long.
- If the value is a typecode, then corba:\_tc\_typecode.
- If the value is of type corba:object, then the typecode of the corba object reference.
- If the value is a single-float, then corba:\_tc\_float.
- If the value is a double-float, then corba:\_tc\_double.
- If the value is a symbol of type nil or t, then corba:\_tc\_boolean.
- If the value is a character, then corba: tc char.
- If the value is an any, then corba:\_tc\_any.
- If the value is a structure, exception, or union, then the relevant type from the IDL.
- If the value is a string, then corba:\_tc\_string (an unbounded string).

# A.4.25 Mapping overview

The detailed mapping guidelines for specific types is designed to conform to a small set of uniform principles.

# A.4.25.1 Rule 1: How names of types are formed

If an IDL identifier I names a type at the top level of some module named M, then the corresponding Lisp type is named  $\mathtt{M:I}$ , that is, the symbol in package M whose name is the string "I".

Nested types are separated by the character "/". Thus, if there is another type J defined within the scope of the type named by I, the corresponding Lisp symbol is M:I/J. This retains consistency with the way in which repository IDs are formed.

## A.4.25.2 Rule 2: How names of operations are formed

The rule for operation package mapping is simpler: all symbols that correspond to Lisp functions that correspond to IDL operations are interned in a single package. This package can be denoted by "op". Thus, op:foo denotes the operation named foo.

## A.4.25.3 Rule 3: Lisp functions corresponding to IDL types

IDL defines many kinds of types: unions, structs, interfaces, and exceptions. We can think of each of these types, informally, as denoting entities with named slots. For example, the named slots of a struct, union, or exception are its members; the named slots of an interface are its attributes.

For each IDL type, there is an associated constructor function that creates a value of that type and there are accessors for each member.

#### The constructor Function

The constructor function corresponding to a type is identical to the (fully scoped) name of the type. It takes keyword initialization arguments whose names are the names of the named members of that type; these initialize the given members.

# **Accessing the Members**

Each named slot defines two functions: a reader and a writer. The reader has the same name as the named slot. The writer uses the standard (setf name) convention familiar to Lisp users. Of course, the home package of the reader is, as for all such function names, the package op.

**Note:** In applying Rule 3, remember that not all of the associated functions make sense for all of the types. For example, there is obviously no constructor function defined for an interface, nor are there writer functions defined for attributes declared readonly.

# A.5 Mapping pseudo-objects to Lisp

*Pseudo-objects* are constructs whose definition is usually specified in IDL, but whose mapping is language specified. A pseudo-object is not (usually) a regular CORBA object.

A pseudo-object differs from a regular CORBA object in the following ways:

- It is not represented in the Interface Repository.
- It may not be passed as a parameter to an operation expecting a CORBA object.
- It may not be returned as a CORBA Object.
- It may not be stored in an any.
- If it is represented as a class, it may not be safely subclassed by user code.

We have chosen the option allowed in the IDL specification to define status as void and have eliminated it for the convenience of Lisp programmers.

Each of the standard IDL pseudo-objects is mapped according to the translation rules just defined.

# A.5.1 Narrowing

In line with the other language binding, we define an operation for narrowing an object reference. The code:

```
(op:narrow class-symbol object-refence)
```

attempts to narrow the given object reference into an object of the named client-side class.

For example, to narrow an client-side value stored in account-ref, into an object of type Bankingdemo: Checkingaccount:

# A.5.2 Typecodes for parsed IDL

Parameters holding the typecode value are generated for all parsed IDL types. If the IDL parser generates a Lisp type of name A:B, then the typecode of the given type is available in the parameter A:\_TC\_B.

For example, the IDL code

# A.6 The mapping of IDL into Common Lisp servants

This section discusses how implementations create and register objects with the ORB runtime.

# A.6.1 Mapping of native types

Specifically, the native type PortableServer::Servant is mapped to the Lisp class named PortableServer::ServantBase. The native type PortableServer::ServantLocator::Cookie can take any lisp value.

# A.6.2 Implementation objects

An interface corresponding to a class named by a Lisp symbol *s* with package *p* and name *n* may be implemented by extending the class named by the symbol whose package is *p* and whose name is the concatenation of *n* to the string "-servant".

If the interface has no base interfaces, then the associated skeleton class has as direct superclasses the class corresponding to the class named portableserver: ServantBase.

Otherwise, if the interface has base interfaces named *A*, *B*, *C*, and so forth, then its associated servant class has as direct superclasses the class corresponding to the servant classes corresponding to *A*, *B*, *C*, and so forth.

Attributes in an interface generate slots of the corresponding name in the oppackage, together with server-side accessor methods.

## A.6.3 Defining methods

The only portable way to implement an operation on a servant class is to use the corba:define-method macro. The syntax of corba:define-method is intended to follow the syntax of the Lisp defmethod macro as closely as possible.

The syntax of corba:define-method is as follows:

This corba:define-method macro is used to implement an operation on an interface. operation-name is a symbol whose name is the name either of an operation or of an attribute declared in an IDL interface implemented by the class named by the symbol receiver-class.

The number of parameter-specifiers listed in the normal-lambda-list must equal the combined number of in and inout parameters declared in the signature of the operation denoted by the function-name, or 0 if the operation is an attribute. If the function-name is a list whose car is setf, the corresponding operation-name should name an attribute that is not readonly.

If function-name denotes an operation, then the effect of corba:define-method is to inform the ORB that requests for the operation on instances of the class receiver-class should return the value or values returned by the body forms of the define-method macro, executed in a new lexical environment in which each parameter-specifier is bound to the actual parameters and in which the context-identifer is bound to the value of the corresponding context.

The operation of corba:define-method in the case in which function-name names an attribute is analogous. The behavior of auxiliary specifiers and of dispatch is the same as their corresponding action under defmethod. Attribute accessors will be generated automatically and inherited by subclasses of the servant classes; the methods can be overridden by user definitions.

Note that the syntax of corba:define-method is a strict subset of that of defmethod: every legal corba:define-method invocation is also a legal defmethod invocation. The main difference between them is that corba:define-method only allows specialization on the first argument. An implementation is free to extend the syntax of corba:define-method, for example, to allow type-checking, interlocking, or multiple dispatch.

## A.6.3.1 Examples

#### A Named Grid Example

The first example shows how one might encapsulate a *named grid*, which is a grid of strings.

This is the IDL of the interface to a named grid of strings:

```
module example{
  interface named_grid{
    readonly attribute string name;
    string get_value ( in unsigned short row,
        in unsigned short column);
    void set_value ( in unsigned short row,
        in unsigned short column,
        in string value);
  }
}
```

The IDL compiler might generate a class corresponding to the example::named\_grid interface using code something like this:

```
(defpackage :example)
(defclass example:named_grid(corba:object)())
```

# **Servant Class Examples**

In order to implement the IDL interface, the user would extend the class example:named\_grid-servant.

The attribute in the IDL will cause the class to have a slot op:name with the appropriate accessors specializing on the class.

# A.6.3.2 Implementation of the IDL operations

The corba:define-method macro is used to define the methods that implement each of the operations defined in the IDL interface. These implementations do not perform any of the argument or range checking that a production system would, of course, perform.

The implementation is free to define other methods on the class, including print-object methods and auxiliary methods for initialize-instance.

# Index

A	interface repository 62
account frames	name service 62
initializing and exiting 39	const 84
account interface 28	constructors 87
any 87	context 78
applications	CORBA
building and testing 19	examples 12
array 84	modules 61
attribute 79	overview 9
attributes	_
mapping for 32	D
readonly 79	debugging 63
v	deduced typecode 88
В	defining
bank client 35	callbacks 41
Bank example	interfaces 14, 37
about 23	methods 92
location of source code 24	utilities for sharing an object
bank interface 30	reference 15
bank server 49	definitions 73
implementing 53	defsystem
basic types 68	Hello World source code 21
boolean 69	designing IDL 27
building and testing applications 19	distinguished packages 72, 73
building the bank client and server 24	<u>_</u>
0	E
C	efficiency optimization
callbacks	using macros instead of functions 77
defining 41	$\mathtt{enum}$ $80$
char 69	error handling
checkingAccount interface 29	user code called by the server 63
Common Lisp	example of a named grid 93
mapping IDL to 32	examples of scoping symbols 74
compiling IDL for CORBA applications 31	exception 77
configuring	exception $86$

exceptions mapping for 33 exiting account frames 39  F first object reference obtaining initial POA object and registering 59 floating point types 70	the ORB 45 integer types 70 interface 75 interface repository configuring 62 interfaces defining 14, 37 mapping for 32 IOR file 15, 21
G generating stub and skeleton code from IDL 15 GUI implementing bank client's 36	L Lisp functions corresponding to IDL types 89 location of Bank example code 24 of Hello World example code 20
H Harlequin Common Lisp ORB about 10 features 12 Hello World 13–20 complete source code 20 how IDL is used 66  I IDL account interface 28 bank interface 30 checkingAccount interface 29 designing 27 generating stub and skeleton code 15 how it is used 66 introduction 65 parsing into stubs and skeletons 61 terminology 71 IDL operations implementation 94 IDL types corresponding Lisp functions 89 implementation objects 91 implementation 94	mapping any 87 array 84 attribute 79 attributes 32 basic types 32, 68 const 84 details 67 enum 80 exception 86 exceptions 33 how names of operations are formed 89 how names of types are formed 88 IDL into Common Lisp servants 91 IDL to Common Lisp 32 interface 75 interfaces 32 Lisp functions corresponding to IDL types 89 module 80 native types 91 operation 76 operations 33
implementing bank client GUI 36 bank server 53 CORBA objects on the server 50 hello world client 17 hello world server 18 servant classes 53 servant methods 54 initializing account frames 39	overview 88 pseudo-objects to Lisp 89 sequence 85 struct 81 summary 66 typedef 87 union 82 member accessors 82 methods defining 92

00	aconing armhola
module 80	scoping symbols
modules	examples 74
CORBA 61	semantics of type mapping 68
NI	sequence 85
N	servant classes
name service	implementing 53
configuring 62	servant methods
named types 70	implementing 54
names	servants
scoped 73	IDL into Common Lisp 91
narrowing an object reference 90	server
native types	bank 49
mapping 91	implementing CORPA objects 50
nicknames for distinguished packages 73	implementing CORBA objects 50
	running 25
0	sharing an object reference 15 skeleton code
object reference	
defining utilities for sharing 15	generating from IDL 15
narrowing 90	skeletons
obtaining first 45	parsing IDL into 61 source code
octet 69	Bank example 24
oneway 77	
operation 76	Hello World example 20
operations 33	string 70 struct 81
ORB	stub code
initializing 45	generating from IDL 15
	stubs
P	parsing IDL into 61
package_prefix pragma 75	symbols
packages	examples of scoping 74
distinguished 72	scoped 73
parameter passing modes 77	system exception 86
parsing IDL into stubs and skeletons 61	-J
PIDL 50	Т
POA object	_
obtaining initial 59	terminology IDL 71
Portable Object Adapter (POA) 50	
pseudo IDL (PIDL) 50	type mapping semantics 68
pseudo-objects	typecode
mapping 89	deduced 88
	typecodes for parsed IDL 90
R	typedef 87
readonly attributes 79	types
registering	basic types 32
first object reference 59	integer 70
return values 77	named types 70
running server and client 25	native types 91
c	II
S	U
scoped names and scoped symbols 73	union $82$

user exceptions 86 utilities for sharing an object reference 15