Harlequin Dylan

Developing Component Software with CORBA®

Version 2.0 Beta

harlequin

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

Product

The Harlequin Dylan Enterprise Edition supports the Common Object Request Broker Architecture (CORBA®) defined by Object Management Group®, Inc. (OMG™).

The Harlequin Dylan Object Request Broker™ (ORB™) and supporting tools provide CORBA architecture functionality to Dylan programmers, combining standardized distributed system development with a state-of-the-art dynamic object-functional language.

Parts

The CORBA components included in the Harlequin Dylan Enterprise Edition are:

1. An IDL (Interface Definition Language™) compiler

   This comes in two forms:

   (**i**) some DLLs linked into the Harlequin Dylan development environment, and console Dylan compiler, to allow IDL files to be included in Dylan projects. This integrates CORBA into the Harlequin Dylan project manager and build system.

   (**ii**) a standalone IDL compiler called:

```
console-scepter.exe
```

in the **`Bin`** subfolder of the top-level Harlequin Dylan installation folder.

2. An ORB runtime library

   This is a single DLL called:

   d*x*orb.dll

   (for some value of ***x***) in the **`Redistributable`** subfolder of the top-level Harlequin Dylan installation folder.

3. Some example CORBA projects including:

   `hello-world`

   `bank`

   `chat`

   in the **`Examples`** subfolder to the top-level Harlequin Dylan installation folder.

4. The manual *Developing Component Software with CORBA*

Audience

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

Standards compliance

The Harlequin Dylan ORB version 2.0 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. See:

`<URL:http://www.omg.org/corba/c2indx.htm>`

This address may change.

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:

`<URL:http://www.omg.org/corba/beginners.html>`

This address may change.

Some recommended highlights are:

- CORBA web pages like "A CORBA overview" at:

  `<URL:http://www.infosys.tuwien.ac.at/Research/Corba/OMG/arch2.htm #446864>`

- 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`

  List discusses building CORBA based systems. To subscribe, send e-mail to **`corba-dev-request@randomwalk.com`** with **`subscribe corba-dev`** in the body of the message.

1

About Harlequin Dylan 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:

```
<URL:http://www.omg.org/corba/whatiscorba.html>
```

It has been reproduced with permission.)

1.2 About the Harlequin Dylan ORB

The Harlequin Dylan ORB is a CORBA-compliant ORB written in Dylan, with a native implementation of the Internet Inter-ORB Protocol (IIOP$^{\text{TM}}$).

The Harlequin Dylan ORB lets you build and run distributed applications in Dylan, straight out of the box. When combined with the Dylan interface to ODBC, you can build 3-tier client-server applications completely in Dylan.

However, the fundamental purpose of the CORBA architecture is interoperability. The ORB's IIOP implementation provides immediate interoperation with any other ORB you may be using. For example, given a Java ORB you could write GUI clients in Swing or AWT that communicate to servers written in Dylan. Conversely, given a C++ ORB you can build DUIM (Dylan User Interface Manager) clients that talk to C++ servers.

Apart from proving that Dylan could tackle another complex domain, the advantages of building an ORB in Dylan were:

- ORB-vendor independence – The Harlequin Dylan 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 Dylan ORB provides "instant CORBA" to get distributed Dylan applications up and running without additional procurement or installation.
- Lower impedance mismatch – No need to trampoline from Dylan to IIOP via another language binding or via a non-Dylan IIOP engine API.
- Expertise – A deeper understanding of the CORBA architecture inside Harlequin which can be shared with our customers via technical support.
- Customization – Harlequin likes to offer a high level of customization support and consultancy 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 Dylan – Providing users with enhanced debugging and interaction facilities available in a homogeneous implementation model.
- Inter-language compatibility – Harlequin has also built the Harlequin Common Lisp ORB. The Dylan ORB borrows from the same design pattern.

1.3 Features of Harlequin Dylan CORBA

The following features are supported in the Harlequin Dylan version 2.0 Enterprise Edition:

- 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)
- Dylan Language Binding

1.4 CORBA examples

The Harlequin Dylan Enterprise Edition includes example applications to help you start writing client/server applications in CORBA.

Hello World	A client/server implementation of the standard Hello World application.
Bank	A three-tier client/server implementation of a banking application.
Chat	An implementation of a chat program that allows users to communicate across a network.

We study the Hello World and Bank applications later in this book. The Hello World example is ready to build and run, straight out of the box. Before you can use the Bank example, you might need to install some (free) Microsoft ODBC drivers, depending on what you have installed on your computer already. The documentation for the Bank example provides instructions.

2 Quick Start Tutorial

2.1 About this chapter

In this chapter, we develop a simple example application that illustrates the basics of writing CORBA-based applications using Harlequin Dylan.

The aim of this chapter is to show the sort of coding involved in using CORBA with Harlequin Dylan, and 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 Dylan.

2.2 A CORBA-based Hello World

This chapter's example is an implementation of the standard Hello World application, using Dylan 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.

These are the basic steps we will take to create this application:

1. Create the projects.

   We use Harlequin Dylan's New Project wizard to create client and server CORBA projects.

2. Define the interface.

   We define the interface to the server using OMG's *Interface Definition Language* (IDL).

3. Implement the client.

4. Implement the server.

5. Build and test the application.

2.3 Creating the projects

In this section we create projects for the client and server parts of our CORBA-based Hello World application.

To create these projects we use the New Project wizard. For CORBA projects, the wizard can create either a client project, a server project, or both at once. Since we are writing both the client and the server, we are going to ask the wizard to create both our projects at once.

1. Select **File > New...** from the Harlequin Dylan main window, or click the New Project button ( ) on the toolbar.

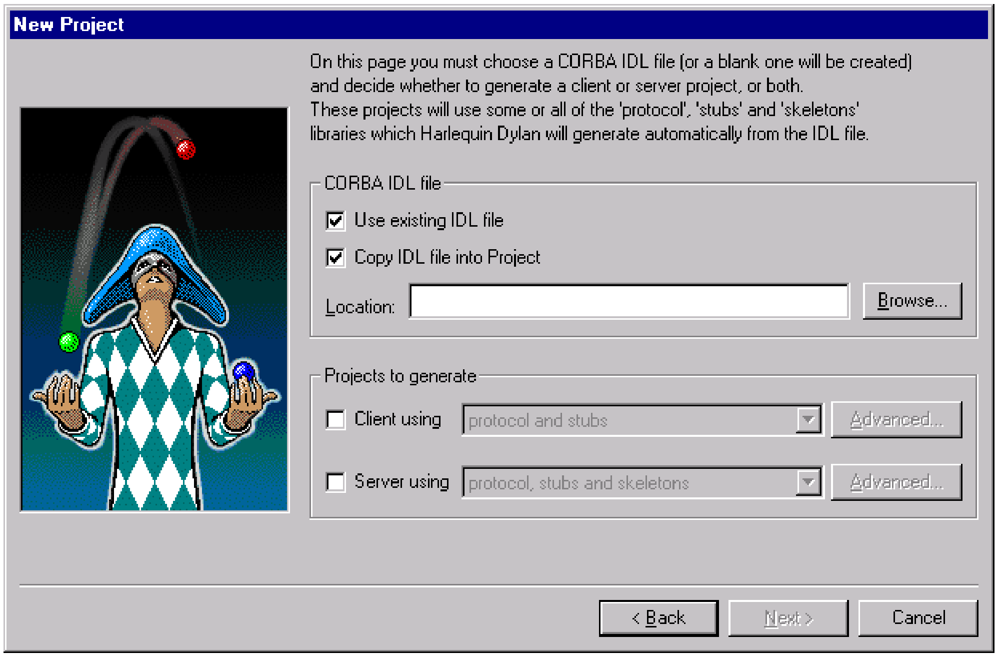
   The New Project wizard appears.

We are going to choose "CORBA Client and / or Server" as the project type. In the wizard, a CORBA project is a project defined relative to an IDL file. The IDL file is a file containing a declaration of the CORBA interfaces that the server implements and the client accesses, written in OMG's Interface Definition Language.

2. Select "CORBA Client and / or Server" in the Project type section.

3. Click **Next**.

   The wizard moves to a second page.



On this page you must choose a CORBA IDL file (or a blank one will be created) and decide whether to generate a client or server project, or both.
These projects will use some or all of the 'protocol', 'stubs' and 'skeletons' libraries which Harlequin Dylan will generate automatically from the IDL file.

- [✓] Use existing IDL file
- [✓] Copy IDL file into Project

Location:

- [ ] Client using: protocol and stubs
- [ ] Server using: protocol, stubs and skeletons

Figure 2.1 The CORBA project page in the New Project wizard.

The second page, shown in Figure 2.1, has several options for building CORBA projects. This is where we can specify whether to create a client project, a server project, or both. We can also choose an IDL file to base projects upon, or let the wizard create an empty one that we can edit after the projects themselves have been created.

If you choose a CORBA IDL file on which to base the project, by default the file is copied into the project folder that you choose on the next page (or is left there if it is already in that folder). You can explicitly choose to leave an existing IDL file where it is and the generated projects will refer to it.

For the sake of this example, however, we will let the wizard generate a blank IDL file for us, and then we will write the IDL by hand.

4. Uncheck the “Use existing IDL file” check box.
5. Check the “Client” box and the “Server” box.

Next to the “Client” and “Server” check boxes are options for the kind of supporting code we want to generate. Like other IDL compilers, Harlequin Dylan’s can create CORBA client *stub* and server *skeleton* code from IDL declarations. This stub and skeleton code, which is generated in the form of separate libraries, takes care of the details of communication between CORBA clients and servers.

The IDL compiler (which we call Scepter) also generates a *protocol* library that represents the spirit of the IDL declarations in Dylan, using generic function declarations, class declarations, and so on.

The protocol library is the public interface to the service provided by the servers and used by the clients, and is used by the stub and skeleton code libraries. On the server side, the protocol defines what you must implement by extending the skeleton implementation in the normal way, by adding methods, subclassing, and so on. On the client side, you use the protocol to invoke operations on the CORBA objects you have implemented in the server.

We will see more about protocol, stub, and skeleton libraries in Chapter 4, “Writing and Compiling IDL”.

6. Make sure the “Client using” setting is “protocol and stubs” and the “Server using” setting is “protocol, stubs, and skeletons”.

   The wizard will set up the client and server projects so that, when they are built, the IDL compiler will be invoked automatically to read the IDL and create stub, skeleton, and protocol projects.

7. Click **Next**.

   The wizard moves to a third page. This page is the wizard’s standard page for entering the project name and location.

On this page we enter the name used to create the client and server projects. Because the wizard is going to create two projects, it uses the name we enter as the stem of each project name, and adds a suffix to identify the project as a client (**`-client`**) or server (**`-server`**).

8. Type **`my-hello-world`** into the “Name” field.

   From this name, the wizard generates projects called My-Hello-World-Client and My-Hello-World-Server.

9. Click **Advanced…**

   The Advanced Project Settings dialog appears.

10. Change the “Windows Subsystem” setting to “Windows Console” and click **OK**.

    This step is necessary because the application is going to run in console mode, that is, without windowing.

11. Click **Next**.

    The wizard moves to a fourth page. This page is the wizard’s standard page for specifying how you want to choose libraries for your project.

12. Select the “Minimal” option.

The choice of “Minimal” makes the Hello World projects use only two libraries: the Harlequin-Dylan library, which provides the Dylan language and basic extensions, and the Dylan-ORB library, which provides the Harlequin Dylan ORB implementation.

13. Click **Next**.

    The wizard moves to a fifth and final page. This page is the wizard’s standard page for entering source file header keywords.

14. Make any header keyword changes you want.

15. Click **Finish**.

The client and server projects are now fully defined. After you click **Finish**, the wizard creates the appropriate project files on disk. Then two project windows appear: one for the client project, My-Hello-World-Client, and one for the server project, My-Hello-World-Server.

2.4 Defining the interface

In this section we declare the IDL interface that the Hello World client and server will communicate across. This is the usual first step in developing a CORBA-based application in Dylan.

We declare the interface in a single file that must have the extension **`.idl`**. Recall that when creating the client and server projects, we unchecked the “Use existing IDL file” option on the second wizard page. Instead of letting the client and server projects refer to an existing IDL file, the wizard created a dummy file into which we can write our IDL interface.

Both the client and server projects point to the dummy IDL file via a file called **`my-hello-world.spec`**. Spec files, or Harlequin Dylan Tool Specification files, contain special information for building projects. The file **`my-hello-world.spec`** is part of each project’s list of sources, but it is a different file in each project.

1. Double-click on **`my-hello-world.spec`** in the Sources page of either project window.

   The spec file opens in an editor window.

The client project’s spec file looks like this:

```
Origin: OMG-IDL
Idl-File: ..\my-hello-world.idl
Stubs: yes
```

The server project’s spec file looks like this:

```
Origin: OMG-IDL
Idl-File: ..\my-hello-world.idl
Skeletons: yes
```

Both spec files contain an **`Idl-File:`** keyword statement saying that the file **`my-hello-world.idl`**, which is in the parent folder of both My-Hello-World-Client and My-Hello-World-Server, should be compiled along with each project’s Dylan sources. We can ignore the other details for the moment.

2. Choose **File > Open...** in either project window, and navigate to the project’s parent folder containing **`my-hello-world.idl`**.

   Hint: clicking the General tab in either the client or server project window shows where the project resides on your machine.

3. Select **`my-hello-world.idl`** and click **Open**.

   The dummy IDL file opens in an editor window.

The dummy IDL file initially contains only a comment and no IDL declarations. We must write these ourselves. For Hello World, the IDL is simply:

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

This IDL declaration says 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`**.

4. Enter the IDL declaration above into the **`my-hello-world.idl`** file.

5. Save **`my-hello-world.idl`** with **File > Save**.

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

2.4.1 Generating stub, skeleton, protocol code from IDL

There are two ways of generating the stub, skeleton, and protocol code for a CORBA application. We can either run the IDL compiler on the command-line, or we can run it within the Harlequin Dylan development environment. The second option is more convenient given that we are already using the environment.

1. Go to the project window for the My-Hello-World-Client project.

Simply building the project in the normal way is enough to invoke the IDL compiler. This convenience is thanks to the spec file in the project Sources list: when the build gets to the spec file, the build system looks in it to see what to do with it. The spec file in My-Hello-World-Client states with the **`Origin:`** and **`Idl-File:`** keywords that there is a OMG IDL file to compile, and with the **`Stubs:`** keyword that a project containing client stub code should be generated.

If we build My-Hello-World-Client now, the IDL compiler will read the IDL file and create the stubs code. Meanwhile the rest of the Dylan code in the project is also compiled.

2. Select **Project > Build** from the project window's menu.

The spec file in My-Hello-World-Server is similar, but asks for skeletons instead of stubs.

3. Select **Project > Build** in the My-Hello-World-Server project window.

After the build, each project's Sources page is updated to include new subprojects. The subprojects are as follows.

In the client project:

My-Hello-World-Stubs	The stubs project contains implementation glue code that enables a client application to call a CORBA object in a server application over a network, using a Dylan protocol generated from the IDL.

In the server project:

My-Hello-World-Skeletons	The skeletons project contains implementation glue code that enables server applications implementing CORBA objects to be called over a network by the Dylan protocol functions generated from the IDL.

In addition, a protocol subproject appears in both the stubs and the skeletons projects:

My-Hello-World-Protocol	The protocol project contains open classes and generic functions, representing the spirit of the IDL declarations in Dylan. It is used in clients to call operations on CORBA objects in a server, and in a server as the basis of the implementation of those objects.

2.4.2 A browsing detour

The next step would be to implement the client and server. But it is instructive to take a brief detour and look at what is inside the new projects that we gained by compiling our small IDL file.

1. Go to the My-Hello-World-Client project window and select the Sources page.
2. Expand the My-Hello-World-Stubs subproject.

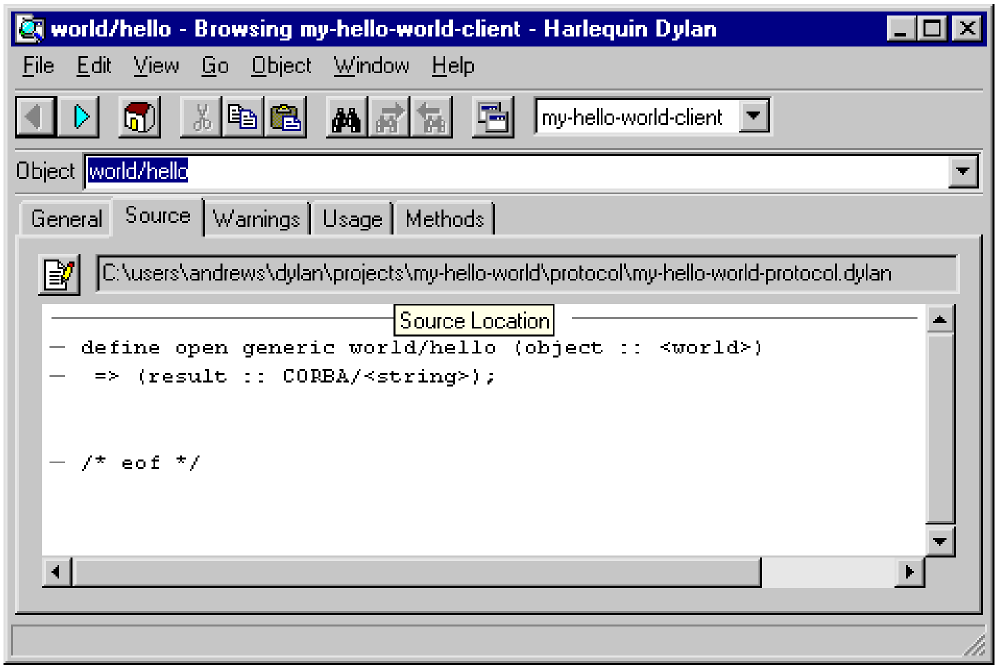
As expected, the My-Hello-World-Stubs project contains the protocol project, My-Hello-World-Protocol.

3. Expand the My-Hello-World-Protocol subproject.

The **`my-hello-world-protocol.dylan`** file contains the automatically generated Dylan protocol representing our IDL for Hello World.

4. Expand the **`my-hello-world-protocol.dylan`** file.
5. Double-click on the generic function for **`world/hello`**.

   The browser appears.



C:\users\andrews\dylan\projects\my-hello-world\protocol\my-hello-world-protocol.dylan

```
define open generic world/hello (object :: <world>)
 => (result :: CORBA/<string>);


/* eof */
```

Figure 2.2 A Dylan generic function mapped from an IDL declaration.

Notice how the Dylan code has been mapped from the IDL. An IDL operation whose full name was **`world::hello`** has been mapped to a Dylan open generic function whose name is **`world/hello`**. The mapping rules are part of what is called an IDL *binding* for Dylan. There is a draft standard for the Dylan IDL binding in Appendix A, “An IDL Binding for Dylan”.

6. Select the browser's Methods page.
7. Double-click on the only method listed.

The browser now shows the source of the method that takes care of bundling up arguments into a request and invoking the ORB in order to send it to the server. This method was generated automatically from the IDL, and "just works" — you do not have to worry about it, or other such methods that the compiler generates.

Note: Since the compiler will regenerate these files if the IDL is changed, you should not edit them by hand.

2.5 Implementing the client

In this section we implement the client side of the Hello World application.

The basic requirements for a client are to initialize an ORB, then obtain references to the CORBA objects that it wants to invoke operations upon. The client can then proceed with execution in the normal way; as necessary, it can invoke the operations on the CORBA objects to which it has references.

2.5.1 Initializing the ORB

The first thing our client needs to do is to initialize the Harlequin Dylan ORB. CORBA provides a standard operation for this, called **`orb_init`**. This is just one of many operations that ORBs must offer. The name is an IDL-style name (in fact it is a *pseudo-IDL* or *PIDL* name) that is mapped to Dylan just like the IDL we wrote for Hello World earlier. In the Harlequin Dylan ORB this operation is invoked as **`corba/orb-init`**.

An ORB initialization call looks like this:

```
let orb = corba/orb-init(make(corba/<arg-list>),
                              "Harlequin Dylan ORB")
```

Ignore the arguments for the moment — they are explained in Chapter 5. The call returns an object of class **`corba/<orb>`**. We need to keep our ORB reference around, so we bind this value to the name **`orb`**.

2.5.2 Obtaining an object reference

The next thing our client needs to do is to get a reference to a CORBA object implementing Hello World. Recall from "Defining the interface" on page 9 that our IDL defined CORBA objects of a kind called **world**:

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

The client needs a reference to a **world** object. An object reference is just CORBA's way of allowing CORBA objects to be identified and communicated with. Once our client has a reference to a **world** object it can call the **hello** operation and get the string it needs to print out.

There is more to say about object references and how they are obtained. For now, all we need to know is one way that clients can obtain object references is by reading them from files, where servers have placed them in a string form — a so-called *stringified object reference*. The client can get the string and convert it into a Dylan object that represents object references. To do this, it uses a .Dylan ORB utility called **corba/orb/file-to-object**.

Thus, given a shared file **hello.ior**, located as follows:

```
c:\temp\hello.ior
```

Our client can get a reference as follows:

```
  let world = as(<world>,
                 corba/orb/file-to-object(orb,
                                          "c:\\temp\\hello.ior")
```

The corba**/orb/file-to-object** method could have been implemented as follows using the standard ORB utility **corba/orb/string-to-object**, but **corba/orb/file-to-object** is provided as a convenience.

```
define method corba/orb/file-to-object (orb :: corba/<orb>,
                                        file :: <string>)
    => (object)
  with-open-file(stream = file, direction: #"input")
    corba/orb/string-to-object(orb, as(<string>,
                                       stream-contents(stream)));
  end;
end method;
```

The **`let world`** … expression takes the name of the shared file, and passes it (and the ORB value we obtained earlier) to the helper method **`file-to-object`**. The **`file-to-object`** method calls **`corba/orb/string-to-object`** to obtain the reference and make a Dylan object representing it. Then the returned Dylan object is coerced into an instance of the class **`<world>`**. That class is one of the protocol library classes generated from the IDL, and was intended to represent **`world`** objects in Dylan. (In fact the object is coerced to be an instance of an internal concrete subclass of **`<world>`** — **`<world>`** is just the public protocol class.)

The extra backslashes in the file name string serve as escape characters.

2.5.3 Invoking an operation

Our client has now initialized the ORB and obtained the **`world`** object reference it required. It is ready to invoke the **`hello`** operation and receive the string that it needs to print out.

Because all the initialization work is complete, this step is trivial. We simply call the hello operation directly, and let the underlying stub code deal with the business of sending the request to the server and receiving the response.

The IDL for hello was translated into the Dylan protocol library, My-Hello-World-Protocol, as **`world/hello`**. Though in IDL the operation did not have any arguments, when calling it from the Dylan client we do need to pass the reference of the CORBA object we are invoking the operation upon, since ORBs need to know which object, of perhaps many the client has references to, is actually being called.

The call is therefore simply:

```
world/hello(world)
```

The implementation of the client is therefore complete after the following call:

```
format-out("%s\n", world/hello(world));
```

This code gets the string and then prints it to the standard output.

2.5.4 Complete code for the client

The following code is the complete implementation of the client. Enter it into the file **`my-hello-world-client.dylan`** in the My-Hello-World-Client project.

```
Module: my-hello-world-client.dylan
Synopsis: Distributed Hello World
Author: Me Myself I

define constant $hello-world-ior-file = "c:\\temp\\hello.ior";

define method main () => ()
  let orb = corba/orb-init(make(corba/<arg-list>),
                           "Harlequin Dylan ORB");
  let world = as(<world>,
                 corba/orb/file-to-object(orb,
                                          $hello-world-ior-file));
  format-out("%s\n", world/hello(world));
end method main;

begin
  main();
end;

// eof
```

2.6 Implementing the server

In this section we implement the server side of the Hello World application.

There are two parts to implementing a server. We need to implement the CORBA objects that the server offers, and we also need to perform ORB initialization and other administrative tasks that get the server ready to service potential clients.

2.6.1 A note on terminology

CORBA objects are what CORBA servers implement and provide to CORBA clients. In CORBA, an object is basically a collection of code that supports an interface declared as an IDL **`interface`**.

Because server applications are typically not much more than the infrastructural wrapper around the code that implements the CORBA object, it is often convenient to talk of a server as if it *is* the CORBA object, or a collection of CORBA objects. Here and elsewhere, we occasionally use the term “CORBA object” where “server” might also be applicable.

2.6.2 Implementing the server’s CORBA objects

According to the IDL we wrote, the server is supposed to provide **`world`** objects, which have an operation called **`hello`**. How do we implement **`world`** and **`hello`** in the server?

An IDL interface, such as **`world`**, is represented in Dylan using a class. Operations in an interface are represented as methods on the class representing the interface.

The server skeletons library we generated by compiling our IDL file contains the underlying CORBA framework for **`world`** objects. This framework is represented by the **`<world-servant>`** Dylan class.

A *servant* is the CORBA way of connecting a request to an object implementation. In this case the mapping will be one-to-one: a servant will be the object implementation. However, many objects could be implemented by one servant, or conversely, many servants could implement a single object. These more complicated applications of servants need not concern us now.

The **`<world-servant>`** class is a subclass of **`<world>`** (again from the protocol library) and another class that provides CORBA functionality. By extending **`<world-servant>`** we provide a Dylan implementation of **`world`** and inherit the CORBA functionality that our objects will need in order to communicate with their clients:

```
define class <world-implementation> (<world-servant>)
end class;
```

As noted in “Invoking an operation” on page 16, the **`hello`** operation name was mapped into Dylan as the **`world/hello`** generic function, defined on **`<world>`**. To implement **`hello`**, we write a **`world/hello`** method on **`<world-implementation>`**:

```
define method world/hello (world :: <world-implementation>)
    => (hello :: <string>)
  "Hello World!"
end method;
```

This method returns the string that will be passed back to the client. With this, the Dylan implementation of the **world** object type we described in IDL is complete.

2.6.3 ORB and object initialization

The other part of writing the server is to perform the initializations necessary for the server to ready itself for CORBA-based interactions with a client.

Some of this coding is similar to what we did on the client side. The first thing we do is initialize, and get a reference to, the Harlequin Dylan ORB. This step is exactly the same as on the client side:

```
let orb = corba/orb-init(make(corba/<arg-list>),
                              "Harlequin Dylan ORB")
```

Next, we can make an instance of the object (servant) class:

```
let impl = make(<world-implementation>);
```

To make this object instance accessible to clients using CORBA, it has to be visible in the CORBA world, not just in the Dylan server application that instantiated it. To do this we use an *object adapter* — part of an ORB that deals with activating CORBA objects and connecting them to the outside world.

The Harlequin Dylan ORB supports CORBA's *portable object adapter* (POA) standard. To start the business of activating our **world** object, we get a reference to the POA:

```
let poa = corba/orb/resolve-initial-references(orb, "RootPOA");
```

This is another call to a standard CORBA operation. The operation **resolve_initial_references** takes an ORB reference and a list of string names for CORBA services, and returns object references for them.

The name used in the call above is the Dylan translation of the pseudo-IDL name **CORBA::ORB::resolve_initial_references**. The call resolves the name **"RootPOA"**, which is the standard name for the basic POA service, into an object reference.

The next step is to get the POA to create a reference for our **`world`** object instance that can be given to a client. There are several ways to get object references from a POA; this is one of them:

```
let world = portableserver/poa/servant-to-reference(poa, impl);
```

Next, we want to publish the reference where the client can find it. Recall that the client looks for the reference in **`c:\temp\hello.ior`**, expecting to find a string there to translate back into a reference with the ORB utility **`corba/orb/file-to-object`**. The ORB also offers the opposite operation, available in Dylan as **`corba/orb/object-to-file`**, Thus the next piece of code is:

```
corba/orb/object-to-file(orb, "c:\\temp\\hello.ior", world);
```

The **`corba/orb/object-to-file`** method could have been defined using the standard ORB operation **`corba/orb/object-to-string`** as follows, but **`corba/orb/object-to-file`** is provided by the Dylan ORB as a convenience.

```
define method corba/orb/object-to-file (orb :: corba/<orb>,
                                        file :: <string>,
                                        object :: corba/<object>)
  with-open-file(stream = file, direction: #"output")
    write(stream, corba/orb/object-to-string(orb, object));
  end;
end method;
```

Next, we have to allow the POA manager managing this POA to start processing requests. (A POA manager is an object that allows us to control the operation of a POA.) We use these standard POA calls:

```
let manager = portableserver/poa/the-poamanager(poa);
portableserver/poamanager/activate(manager);
```

Finally, on the servant side we need to set the ORB running to receive client requests:

```
corba/orb/run(orb);
```

2.6.4 Complete code for the server

The following code is the complete implementation of the server. Enter it into the file **`my-hello-world-server.dylan`** in the My-Hello-World-Server project.

```
Module: my-hello-world-server.dylan
Synopsis: Distributed Hello World
Author: Me Myself I

define constant $hello-world-ior-file = "c:\\temp\\hello.ior";

define class <world-implementation> (<world-servant>)
end class;

define method world/hello (world :: <world-implementation>)
    => (hello :: <string>)
  "Hello World!"
end method;

define method main () => ()
  let orb = corba/orb-init(make(corba/<arg-list>),
                           "Harlequin Dylan ORB");
  let poa = corba/orb/resolve-initial-references(orb, "RootPOA");
  let impl = make(<world-implementation>);
  let world = portableserver/poa/servant-to-reference(poa, impl);
  corba/orb/object-to-file(orb, $hello-world-ior-file, world);
  let manager = portableserver/poa/the-poamanager(poa);
  portableserver/poamanager/activate(manager);
  corba/orb/run(orb);
end method main;

begin
  main();
end;
```

2.7 Building and testing the application

In this section we build and test the completed CORBA-based Hello World application.

Once you have replaced the dummy code in **`my-hello-world-client.dylan`** with the code in section 2.5.4, and the dummy code in **`my-hello-world-server.dylan`** with the code in section 2.6.4, the client and server applications are ready to build again.

1. Choose **Project > Build** in the My-Hello-World-Client project.
2. Choose **Project > Build** the My-Hello-World-Server project.

Now, you can run the application to test that it works. You need to run the server first so that it is waiting to receive calls from the client by the time the client makes its call.

3. Choose **Project > Start** in the My-Hello-World-Server project.

   Wait until an MS-DOS console window appears. The server is now running.

4. Choose **Project > Start** in the My-Hello-World-Client project.

The client pops up another MS-DOS console window, into which it prints **`Hello World!`** before the development environment traps its exit and presents a confirmer.

Try quitting the client by clicking OK, and then running it repeatedly. Each time it gets a string from the same server, which keeps running all the while.

An interesting exercise is to set breakpoints in the client and server code to trap the request and open debugger windows as the request passes from client to server.

This concludes the first example.

3

Setting up the Bank Example

3.1 About the bank example

Chapters 4, 5, and 6 of this guide present a deeper CORBA development example than the Hello World example we saw in chapter 2. The new example is a simple implementation of a bank. The architecture of the bank has three components:

- A database that provides persistent storage for the accounts managed by the bank.
- 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.

This application is a typical example of a three-tier application architecture comprising a database access layer, a business logic layer, and a user interface layer.

Accounts are stored as records in a Microsoft Access™ relational database. The database is manipulated by the server using the Harlequin Dylan SQL-ODBC library.

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 Dylan 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 windows for the bank and each of the accounts that are active in the server.

We will use the Harlequin Dylan DUIM library to implement the client's user interface.

3.2 Where to find the example code

The bank example code is available in the Harlequin Dylan Examples menu, under CORBA. There is a Bank Client project and a Bank Server project.

The same code is also available in the top-level Harlequin Dylan installation folder, under

```
Examples\corba\bank
```

This folder has several subfolders.

- **`Examples\corba\bank\bank`** contains the file **`bank.idl`**. This is the IDL file declaring the CORBA interface to the server.
- **`Examples\corba\bank\bank-client`** contains the implementation of the client as project **`bank-client.hdp`**.
- **`Examples\corba\bank\bank-server`** contains the implementation of the server as project **`bank-server.hdp`**. This folder also contains a ready made Microsoft Access database file **`bankDB.mdb`**. The application uses this to record bank account details.

3.3 ODBC requirements

In order to run the example, you need to have ODBC version 3.0 (or higher) and an ODBC driver for Microsoft Access installed on the machine hosting the server application. You do not need a copy of Microsoft Access.

Both ODBC 3.*x* and the Microsoft Access driver are available free for download from the Microsoft Universal Data Access web site:

`<URL:http://www.microsoft.com/data/>`

From the Downloads section, download the Microsoft Data Access Components, version 1.5c or higher, for your Windows platform. You do not need to download the full 6.4MB file — **`mdacfull.exe`**, the MDAC 1.5c redistribution setup file, is only 3.4MB and contains everything you need.

This file installs ODBC 3.*x* and the Microsoft Access Driver, amongst other things.

This information may change in the future.

3.4 Registering the database with ODBC

So that the bank server can access the **`bank.mdb`** database using ODBC, we need to register the database as an ODBC data source. This installation step also tells ODBC which driver to use when connecting to the database.

To register the database in this way, ODBC drivers must be installed on the machine that will host the server.

3.4.1 Registering the database on Windows 95 and 98

To register the **`bank.mdb`** database with ODBC on Windows 95 and Windows 98:

1. Install the Microsoft Data Access Components if you have not done so already.

   See “ODBC requirements” on page 25 for details of the version you need and where to find it.

2. Now follow the instructions for registering the database with ODBC on Windows NT 4.

   See Section 3.4.2, below.

3.4.2 Registering the database on Windows NT 4

To register the **bank.mdb** database with ODBC on Windows NT 4:

1. From the Windows **Start** menu, choose **Start > Settings > Control Panel**.
2. Double-click the **ODBC** or **32bit ODBC** icon.

   The ODBC Data Source Administrator appears.
3. Select the **User DSN** page.
4. Click **Add...**.
5. Select **Microsoft Access Driver** from the list of available drivers.
6. Click **Finish**.

   The ODBC Microsoft Access Setup dialog appears.
7. In the ODBC Microsoft Access Setup dialog, enter **bankDB** in the Data Source Name field.
8. Click **Select...**.

   The Select Database dialog appears.

We can now specify the database file name. The file is stored under the top-level Harlequin Dylan installation folder, in the subfolder:

```
Examples\corba\bank\bank-server
```

9. In the Select Database dialog, browse until you reach the folder above.
10. Select **bankDB.mbd** from the list of available files, then click **OK**.
11. Click **OK** again to close the ODBC Microsoft Access Setup dialog.
12. Click **OK** to close the ODBC Data Source Administrator.

We also need to ensure that the **bank.mdb** file is writable.

1. Right-click on the file in a Windows Explorer window.
2. Select **Properties** from the shortcut menu.

3. Clear the Read-only attribute if it is checked, and click **OK**.

The ODBC setup work for the demo is now complete. We can move on to building and running the demo itself.

3.5 Building the Bank client and server

We can now build the client and server applications for the demo.

1. Start Harlequin Dylan.

The client and server projects are available in the Examples dialog. Either choose Open Example Project in the initial dialog as Harlequin Dylan starts up, or choose **Help > Open Example Project…** from the main window.

2. In the Examples dialog, open the Bank-Client project.

3. In the Bank-Client project window, choose **Project > Build**.

The IDL compiler, Scepter, is invoked automatically during the build process. It compiles the file **`bank.idl`** to generate the source code for the protocol and stubs libraries.

4. Bring up the Examples dialog again, and open the Bank-Server project.

5. In the Bank-Server project window, choose **Project > Build**.

3.6 Running the server and client

We can now run the bank demo for the first time.

In the Bank-Server project window, choose **Application > Start** to run the server executable. The server is represented by an administration window with a raw table of the database contents, a log window for seeing requests, and a couple of menu items.

Once you have finished interacting with the bank, click the close button in the top right-hand corner of this bank server window to exit. Do not do this yet.

Once the server’s dialog has appeared, go to the Bank-Client project window and choose **Application > Start** to run the executable for the client. A window presenting a GUI to the bank should appear.You can now interact with the bank to create new accounts, deposit amounts, and so on.

Once you have finished interacting with the bank, click the close button in the top right-hand corner of the bank window to exit the client application. Then do the same for the server window.

4

Writing and Compiling IDL

4.1 Writing IDL for a CORBA application

The first step in developing a CORBA application is to define the interfaces to its distributed application objects. You define these interfaces using OMG's *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 application, targeting such an object, may request.

Our bank server 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 Harlequin Dylan Examples folder, under

```
Examples\corba\bank\bank\bank.idl
```

4.1.1 IDL for the account interface

This is 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);
};
```

An **account** object's **name** attribute is used to store the name of the account holder. The state of an account is recorded in the **balance** attribute. To keep things simple, we use CORBA **long** values to represent the monetary amounts that we use to store account balances and to modify them with credits and debits.

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 is 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 using a **`string`**.

The operation **`debit`** subtracts a given amount from the current account balance, as long as this does not make the account balance negative. Qualifying **`debit`** by the phrase **`raises (refusal)`** declares that invoking this operation might raise the exception **`refusal`**. This phrase is necessary because 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

Our application manages a second sort of bank 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 just 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 **`checkingAccount`**s.

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** object 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**, and returns an object reference of interface **account**. The operation can 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 can be used wherever an **account** object is expected, whether as the actual argument, or the result, of an operation. For instance, we can use **closeAccount** to close **checkingAccount** objects as well as **account** objects, and we can use **retrieveAccount** to fetch **checkingAccount** objects as well as **account** objects.

4.2 Compiling IDL for a CORBA application

Harlequin Dylan includes an IDL compiler, Scepter, that it uses to check and compile IDL files into Dylan libraries. These libraries provide essential infrastructure for CORBA-based applications in Dylan.

The libraries are built according to the specification in Appendix A, "An IDL Binding for Dylan". That document is a draft specification for a standard mapping from CORBA IDL to the Dylan language. Briefly, the specification states that:

- CORBA types are mapped to Dylan types and classes
- CORBA interfaces are mapped to Dylan classes
- CORBA interface inheritance is mapped to Dylan class inheritance
- CORBA attributes are mapped to Dylan getter and setter functions
- CORBA operations are mapped to Dylan generic functions

- CORBA exceptions are mapped to Dylan conditions

IDL declarations are mapped to Dylan according to these rules. The resulting libraries provide a Dylan protocol equivalent to the IDL.

4.2.1 Libraries created by compiling IDL

The IDL compiler can produce *skeleton*, *stub*, and *protocol* libraries from an IDL file. This, again, is as specified in Appendix A, “An IDL Binding for Dylan”. The purpose of these libraries is to make writing CORBA applications easier, by providing a pre-built interface to CORBA operations.

The skeletons or *server skeletons* library contains code for use by a CORBA server application, while the stubs or *client stubs* library contains code for use by a CORBA client application. In both cases, the code hides the details of CORBA communication from the application, allowing you to invoke operations in other CORBA objects without having to worry about where those objects are running. The stubs and skeletons act as proxies for the real, remote operations.

The protocol library is a Dylan representation of the interface described in the IDL file. The Dylan representation is mapped from IDL according to the Dylan IDL binding, with open classes and open generic functions representing IDL interfaces and operations. As we saw in Chapter 2, “Quick Start Tutorial”, the protocol provides the basis for implementing clients and servers. The skeletons and the stubs library both use the protocol library and re-export the names from it.

Note: Typically a server project uses the client stubs library in addition to the skeletons and protocol library. This allows the server to make callbacks to the client.

4.2.2 IDL files in Dylan projects

IDL files can be treated as part of Dylan projects, allowing the IDL interface to be compiled at the same time as the project’s Dylan sources. This is in fact the simplest way to manage a Dylan project that uses CORBA facilities.

However, you do not include IDL files directly in the project. Rather, each IDL file that a project depends on is represented in the project by a corresponding Harlequin Dylan Tool Specification (spec) file.

The spec file indicates the path to the IDL file, and states which of the skeletons, stubs, and protocol libraries the project requires. The Harlequin Dylan development environment uses the spec file to invoke the IDL compiler as part of the normal build process for the project.

The Bank-Client and Bank-Server projects each contain a spec file for the **`bank.idl`** file. The Bank-Client project's spec file, **`idl.spec`**, requests that stubs and protocol libraries be generated. The Bank-Server project's spec file requests that skeletons, stubs, and protocol libraries be generated. (This second spec file is also called **`idl.spec`** but is a different file to the one in Bank-Client.)

After building both the bank client and the bank server, there will be three new subfolders:

`Examples\corba\bank\bank\stubs`

Contains the project **`bank-stubs.hdp`** that defines the Bank-Stubs library. This library is used by the implementation of the bank client. Its project is automatically added to **`bank-client.hdp`** as a subproject.

`Examples\corba\bank\bank\skeletons`

Contains the project **`bank-skeletons.hdp`** that defines the Bank-Skeletons library. This library is used by the implementation of the bank server. Its project is automatically added to **`bank-server.hdp`** as a subproject.

`Examples\corba\bank\bank\protocol`

Contains the project **`bank-protocol.hdp`** that defines the Bank-Protocol library. This library is shared by both the Bank-Skeletons and Bank-Stubs libraries, and added automatically to the projects of both.

4.2.3 Compilation steps

If you have not built the Bank-Client and Bank-Server projects yet, you should do so now.

1. Open the Bank-Client and Bank-Server projects from the CORBA section of the Examples dialog.

   You can bring up the Examples dialog by choosing **Help > Open Example Project…**.

2. Choose **Project > Build** in each project window.

   Notice how the stubs, skeletons, and protocol projects are generated and added automatically to the top-level client and server projects.

4.3 Mapping IDL to Dylan

To get an impression of the mapping from IDL to Dylan, we can take a look at the result of applying the mapping to the file **bank.idl**. The following Dylan definitions are taken from **bank-protocol.dylan**, part of the Bank-Protocol project produced by compiling **bank.idl** IDL:

```
define open abstract class BankingDemo/<account> (<object>)
end class;

define open generic BankingDemo/account/name
    (object :: BankingDemo/<account>)
  => (result :: CORBA/<string>);

define open generic BankingDemo/account/balance
    (object :: BankingDemo/<account>)
  => (result :: CORBA/<long>);

define open generic BankingDemo/account/credit
    (object :: BankingDemo/<account>,
     amount :: CORBA/<unsigned-long>)
  => ();

define sealed class BankingDemo/account/<refusal>
    (CORBA/<user-exception>)
  slot BankingDemo/account/refusal/reason :: CORBA/<string>,
       required-init-keyword: reason:;
end class;
```

```
define open generic BankingDemo/account/debit
    (object :: BankingDemo/<account>, amount :: CORBA/<long>)
  => ();

define open abstract class BankingDemo/<checkingAccount>
  (BankingDemo/<account>)
end class;

define open generic BankingDemo/checkingAccount/limit
    (object :: BankingDemo/<checkingAccount>)
  => (result :: CORBA/<long>);

define open abstract class BankingDemo/<bank> (<object>)
end class;

define open generic BankingDemo/bank/name
    (object :: BankingDemo/<bank>)
  => (result :: CORBA/<string>);

define sealed class BankingDemo/bank/<duplicateAccount>
  (CORBA/<user-exception>)
end class;

…
```

To provide some more intuition for the mapping scheme, the following subsections examine the Dylan counterparts of some of the more representative IDL declarations from the file **bank.idl**. See Appendix A, "An IDL Binding for Dylan" for the full mapping description.

4.3.1 Mapping for interfaces

The IDL interfaces **account**, **checkingAccount** and **bank** map to the Dylan abstract classes **BankingDemo/<account>**, **BankingDemo/<checkingAccount>** and **BankingDemo/<bank>**.

Dylan does not support nested namespaces, so in Dylan, the IDL scope identifier **BankingDemo** is prefixed to the name of each interface defined within its scope. This is how we get the Dylan class identifiers **BankingDemo/<account>**, **BankingDemo/<checkingAccount>** and **BankingDemo/<bank>**.

Notice how IDL interface inheritance (**checkingAccount**: **account**) maps naturally onto Dylan class inheritance: the class **BankingDemo/<checkingAccount>** is defined as a subclass of **BankingDemo/<account>**).

4.3.2 Mapping for basic types

The IDL types **`string`**, **`long`**, and **`unsigned long`** are mapped to the Dylan classes **`CORBA/<string>`**, **`CORBA/<long>`** and **`CORBA/<unsigned-long>`**, which are simply aliases for the Dylan classes **`<string>`**, **`<integer>`** and a positive subset of **`<integer>`**.

4.3.3 Mapping for attributes

The read-only **`balance`** attribute of an IDL **`account`** gives rise to the Dylan generic function:

```
define open generic BankingDemo/account/balance
    (object :: BankingDemo/<account>)
  => (result :: CORBA/<long>)
```

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

```
define open generic BankingDemo/account/balance-setter
    (value :: CORBA/<long>, object :: BankingDemo/<account>)
  => (value :: CORBA/<long>)
```

Recall that, in the IDL source, the **`balance`** attribute is declared within the definition, and thus the subordinate namespace, of the **`BankingDemo`** module and the **`account`** interface. Again, because Dylan does not support nested namespaces, the IDL scope identifiers **`BankingDemo`** and **`account`** are simply prefixed to the name of the attribute's getter method, resulting in the Dylan function identifier **`BankingDemo/account/balance`**.

Aside: More generally, the Dylan language IDL binding specifies that an IDL identifier is mapped to a Dylan identifier by appending together all the enclosing scope identifiers and the scoped identifier itself, separating the identifiers by forward slashes (`/`).

4.3.4 Mapping for operations

The IDL operation **`credit`** is mapped to the Dylan generic function:

```
define open generic BankingDemo/account/credit
    (object :: BankingDemo/<account>,
     amount :: CORBA/<unsigned-long>)
  => ();
```

In IDL, the **credit** operation is defined within the **account** interface, declaring it to be an operation on **account** objects. The Dylan language IDL binding adopts the convention that an operation's target object should be passed as the first argument of the corresponding Dylan generic function. Thus the first parameter of the generic function **BankingDemo/account/credit** is an object of class **BankingDemo/<account>**.

The operation's **in** and **inout** arguments become the remaining parameters of the corresponding Dylan 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 :: CORBA/<long>**, if the **BankingDemo/account/credit** generic function.

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

4.3.5 Mapping for exceptions

The IDL exception **refusal** maps onto the Dylan class **BankingDemo/account/<refusal>**. Its member, **string reason;**, maps onto a slot **BankingDemo/account/refusal/reason :: CORBA/<string>**.

Note that **BankingDemo/account/<refusal>** is a subclass of **CORBA/<user-exception>** and, by inheritance, of Dylan **<condition>**. This means that CORBA user exceptions can be raised on the server, and handled in the client, using the standard Dylan condition mechanism.

5

The Bank Client

5.1 The bank client

In this chapter, we design and implement a CORBA client, using Dylan and the Harlequin Dylan ORB.

Our client presents a graphical user interface to a bank object and its operations. We implement the user interface using DUIM, Harlequin Dylan's window programming toolkit. Since 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, together with some documentation, fully determines its functionality. This means that in writing the client we need only rely on the information in the documented IDL to be able to interact with a bank object. Knowing the IDL description, we can implement the client before our bank object implementation is available.

The Bank-Protocol library, which was produced by the IDL compiler, Scepter, merely specifies the protocol for interacting with CORBA objects satisfying the interfaces in the IDL file **`bank.idl`**. The client-side implementation of this protocol resides the Bank-Stubs library. Any application that wants to act as a client with respect to some CORBA object matching an interface in the **`bank.idl`** file should use the Bank-Stubs library.

The Bank-Stubs library defines the following concrete classes:

```
BankingDemo/<account-reference>
BankingDemo/<checkingAccount-reference>
BankingDemo/<bank-reference>
```

These classes subclass the following abstract classes:

```
BankingDemo/<account>
BankingDemo/<checkingAccount>
BankingDemo/<bank>
```

The class **`BankingDemo/<checkingAccount-reference>`** is defined to inherit from **`BankingDemo/<account-reference>`**, matching the inheritance relationship in the IDL. Instances of these classes act as proxies for CORBA objects running on the server.

The Bank-Stubs library also defines a concrete *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 Dylan condition of the appropriate class. This condition can then be handled by the client code using standard Dylan constructs.

5.3 Requirements for implementing the bank client

There are two parts to implementing the bank client:

- Write the code to initialize the CORBA ORB, and obtain a reference to a bank server object.
- Write the code for the client's GUI.

We start with the GUI implementation.

5.4 Implementing the bank client's GUI

Note: This section assumes some basic knowledge of DUIM (Dylan User Interface Manager), Harlequin Dylan's window programming toolkit. See the manual *Building Applications Using DUIM* for details. However, you do not need to know about DUIM to follow the rest of the tutorial.

Since this demonstration principally concerns CORBA, and because we would like to revamp the look-and-feel of the demonstration occasionally, we no longer describe the GUI implementation in great detail. Instead only a brief outline of the current design is given.

The bank client consists of one window that shows a table of retrieved accounts. Each row in the table shows the name, the current balance, and the overdraft limit (if applicable). CORBA operations are mapped on to menu items whose callbacks make the necessary requests.

The bank client is implemented as a library:

```
define library bank-client
  use harlequin-dylan;
  use dylan-orb;
  use bank-stubs;
  use duim;
  ...
end library bank-client;
```

that defines a single module:

```
define module bank-client
  use harlequin-dylan;
  use dylan-orb;
  use bank-stubs;
  use duim;
  ...
end module bank-client;
```

(See **`library.dylan`** and **`module.dylan`** in the Bank-Client project.)

Any application that wants to use the Harlequin Dylan ORB should use the Dylan-ORB system library and module, in addition to any application-specific libraries. Because our application acts as a client of CORBA objects satisfying interfaces defined in the **`bank.idl`** file, it also needs to uses the Bank-Stubs library and module. It also needs to use the DUIM library and module to construct the graphical user interface.

The focal point of the bank client GUI is the **`<bank-frame>`** class defined by DUIM's **`frame-definer`** macro. This maintains a set of account references and organizes the tabular layout of their details.

Defining the callbacks attached to each menu item is straightforward. Recall that in DUIM, the argument passed to a callback is the gadget whose activation triggered that callback, and that the DUIM function **`sheet-frame`** can be used to return the enclosing frame of a gadget.

The source code for the client GUI is in file **`bank-client.dylan`**.

5.5 Implementing CORBA initialization for the bank client

Having written the client GUI we are now ready to set up the client's CORBA environment. 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, an IDL (or PIDL) 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 CORBA features such as Object Adapters, the Interface Repository, the 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 Dylan class inherits from the class **<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 **ORB_init** is defined outside the scope of any interface, providing a means of bootstrapping into the CORBA world. Calling **ORB_init** 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 Dylan 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, as in the Hello World example of Chapter 2, the client can obtain the initial object reference from a shared file, in which the server has published a reference to its implementation of the bank object, encoded as a string. After starting up, the client reads the file, decodes the string into an object reference (using the ORB utility operation **`file_to_object`**, which in turn uses **`string_to_object`**), and then uses this reference as the target of further operations.

Alternatively, this demonstration can also use a Name Service to communicate the initial bank reference between the client and server. A Name Service acts as an intermediary, allowing the server to register a reference against *name*, and then allowing the client to query for the associated reference. To use the Name Service pass

```
-location-service:naming-service
```

on the command line of the client.

To change the command line arguments given to the program, choose the **Project > Settings...** dialog and switch to the Debug tab page. By default the command line arguments for the Bank demo are

```
-ORBname-service-file c:\temp\ns.ior -location-service:shared-
file
```

which tells the ORB where the Name Service is, but that it should use a shared file to pass the initial Bank reference.

Here is some the Dylan code that implements the initialization of the client:

```
define method initialize-client ()
  let orb = CORBA/ORB-init(make(CORBA/<arg-list>),
                           "Harlequin Dylan ORB");
  let ls = get-location-service();
  block ()
    let bank = lookup-bank(orb, ls);
    let bank-frame = make(<bank-frame>, bank: bank);
    start-frame(bank-frame);
  exception (lookup-bank-failure(orb, ls))
    notify-user("Cannot locate the Bank. Click OK to Exit.");
  end block;
end method;
```

This method first initializes the Harlequin Dylan ORB by calling the Dylan generic function **`CORBA/ORB-init`** corresponding to the PIDL **`ORB_init`** operation. (Note that the IDL module name **`CORBA`** forms a prefix of the Dylan operation name, and that IDL underscore “_” maps to a Dylan dash “**`-`**”.) The first argument to this call evaluates to an empty **`CORBA/<arglist>`**. Passing an empty sequence instructs the **`CORBA/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 Dylan Orb"`**, merely identifies the ORB to use. The call returns an object of class **`CORBA/<ORB>`**.

The function **`get-location-service`** reads the command line to see whether to look for a shared file or use a Name Service. It then passes this information to the function **`lookup-bank`**, which knows how to get a bank reference using either method. For example, for the shared file case **`lookup-bank`** does the following:

```
define method lookup-bank (orb :: corba/<orb>,
                        location-service == #"shared-file")
    => (bank :: bankingdemo/<bank>)
  as(BankingDemo/<bank>, corba/orb/file-to-object(orb,
                                          $bank-ior-file));
end method;
```

The constant **`$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.

Invoking **`CORBA/ORB/file-to-object`** on this ORB, passing the shared file name, reconstitutes the IOR string contained in the file as an unspecific object reference of class **`CORBA/<Object>`**. Calling the **`as`** method on this object reference narrows (that is, coerces) it to a more specific object reference of class **`BankingDemo/<bank>`**. (The **`as`** method, which is generated by the IDL compiler and defined in the Bank-Stubs library, 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`**, of class **`BankingDemo/<bank>`**, is used to make and start a new bank frame, displaying the initial GUI to the user.

The full implementation of the client initialization can be found in the file **`init-client.dylan`**.

The implementation of the client is now complete.

6 The Bank Server

6.1 The server

In this chapter, we design and implement a CORBA server, using Dylan and the Harlequin Dylan 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 store the account records in a Microsoft Access™ relational database. The server manipulates this database using Harlequin Dylan's SQL-ODBC library.

Since the primary motivation for this tutorial is to illustrate the use of CORBA, we will not spend too much time on the use of the SQL-ODBC library. Instead, we concentrate on how to provide implementations of CORBA objects, and how to make those implementation available to the CORBA environment.

A very small amount of SQL knowledge is assumed in early sections of this chapter.

6.2 The ODBC database

The folder **`Examples\corba\bank\bank-server`** contains a prepared Microsoft Access database file **`bankDB.mdb`** that the server uses to record account details.

This database contains a single SQL table called **`Accounts`**:

Name	Balance	Limit
Jack	0	NULL
Jill	-100	200

Table 6.1 The Accounts table in database file bank.mdb.

The **`Accounts`** table has three columns: **`Name`**, **`Balance`**, and **`Limit`**. Each record in the table represents an account.

- The **`Name`** column contains values of SQL type **`string`** that are used to uniquely identify account records.
- The **`Balance`** column contains values of SQL type **`long`**, reflecting the current balance of each account.
- The **`Limit`** column contains either the distinguished SQL value NULL that indicates an absent value, or a value of SQL type **`long`**, reflecting the overdraft limit of a checking account.

Note that both accounts and checking accounts are stored as records in the same table. By convention, we interpret a record with a NULL **`Limit`** value an ordinary account. We interpret a record with a non-NULL, and thus integral, **`Limit`** value as a checking account with the given overdraft limit.

For instance, the **`Accounts`** table above contains two records, one for Jack's account and one for Jill's checking account. (Recall that checking accounts can have negative balances while ordinary accounts cannot.)

6.3 Overview of the Harlequin Dylan SQL-ODBC library

The SQL-ODBC library is built on top of a generic SQL library. The SQL library does not include the low-level code necessary to communicate with any particular database management system (DBMS): it simply provides a convenient high-level mechanism for integrating database operations into Dylan applications.

The SQL library is designed to form the high-level part of implementation libraries that contain lower-level code to support a particular DBMS protocol, such as ODBC. The SQL-ODBC library is one such implementation library.

The SQL library defines the following abstract classes:

- The abstract class **`<dbms>`** is used to identify a database management system.

  The SQL-ODBC library defines an instantiable subclass **`<odbc-dbms>`** of **`<user>`** for identifying the ODBC DBMS.

- The abstract class **`<user>`** identifies users to a DBMS.

  Exactly what a user means depends on the DBMS. The **`make`** method on **`<user>`** takes two keyword arguments, **`user:`** and **`password:`**. The **`user:`** init-keyword takes an instance of **`<string>`** that should be a valid user name for the DBMS in question. The **`password:`** init-keyword should be the password that accompanies the user name.

  The SQL-ODBC library defines the instantiable subclass **`<odbc-user>`** of **`<user>`** for identifying a user to an ODBC DBMS.

- The abstract class **`<database>`** identifies a database to a DBMS.

  Exactly what a database is depends on the DBMS in question. The SQL-ODBC library defines the class **`<odbc-database>`**, a subclass of **`<database>`**, whose instances identify databases to an ODBC DBMS. In particular, the **`make`** method on **`<odbc-database>`** accepts the **`datasource-name:`** keyword argument to specify the name of the ODBC datasource, as a **`<string>`**.

- The abstract class **`<connection>`** represents database connections.

  Instances of this class identify an execution context for executing SQL statements. The exact composition of a connection depends on the DBMS.

  The SQL-ODBC library defines the class **`<odbc-connection>`**, a subclass of **`<connection>`**. Instances of this class are created upon making a connection an ODBC database.

The **`<sql-statement>`** class represents SQL statements and their state. This class has the init-keywords **`text:`**, **`input-indicator:`**, and **`output-indicator:`**. The required keyword **`text:`** expects a **`<string>`** that contains the text of an SQL statement. *Host variables* can be included in the statement by placing a question mark (**`?`**) at the point in the string at which the value of the host variable should be substituted. The optional keyword argument **`output-indicator:`** expects an instance of **`<object>`**. The output indicator is a substitution value to be used whenever the column of a retrieved record contains the SQL NULL value. The optional keyword **`input-indicator:`** expects an instance of **`<object>`**. The input indicator is a marker value used to identify SQL NULL values in host variables.

The SQL library defines two convenient macros that we use in this tutorial: **`with-dbms`** and **`with-connection`**. Here is the form of a **`with-dbms`** call:

```
with-dbms (dbms)
  body
end with-dbms;
```

The **`with-dbms`** statement macro considers *dbms*, which must be a general instance of class **`<dbms>`**, to be the DBMS in use throughout *body*. For example, if *dbms* is an instance of **`<odbc-dbms>`** and *body* contains a call to **`connect`**, then the call actually returns an **`<odbc-connection>`**.

Here is the form of a **`with-connection`** call:

```
with-connection (connection)
  body
end with-connection;
```

The **`with-connection`** statement macro considers *connection*, which must be an instance of class **`<connection>`**, to be the default database connection in use throughout *body*. For instance, each call to **`execute`** an SQL statement within *body* uses *connection* by default, so that the call's **`connection:`** keyword argument need not be supplied.

A call to the generic function **`connect(`***database*, *user*) returns a new connection of class **`<connection>`** to the specified database *database* as the user *user*. The connection can be closed by a call to **`disconnect(`***connection*).

A call to the generic function **`execute(`***sql-statement*, **`parameter:`** *vector*) executes the SQL statement *sql-statement* on the default connection. The (optional) **`parameter:`** argument supplies the vector of values to be substituted for any host variables appearing in the statement's text. The *n*th entry of this vector determines the value of the *n*th host variable. Vector entries that equal the value of the statement's **input-indicator:** keyword argument are sent as SQL NULL values.

If the SQL statement is a **select** statement, then the result of executing the statement (with **execute**) is a value of class **<result-set>**, which is itself a subclass of Dylan's built in **<sequence>** class. Each element of a result set is a record and each element of a record is a value. The various Dylan collection protocols and functions work as you would expect on a result set. For the purpose of this tutorial, it suffices to think of a result set as a sequence of vectors.

Just to illustrate the use of the SQL-ODBC library without worrying about the implementation of our CORBA server, here is a code fragment that might be used to extract the entries in the **Name** and **Balance** columns of the **bankDB.mdb** database:

```
begin
  // choose the DBMS
  let odbc-dbms = make(<odbc-dbms>);

  with-dbms (odbc-dbms)
    // identify the database
    let database = make(<database>, datasource-name: "bankDB");

    // identify the user
    let user = make(<user>, user-name: "", password: "");

    // establish a connection for this database and user
    let connection = connect(database, user);

    with-connection (connection) // make it the default
      let query1 =               // construct the query…
        make(<sql-statement>,
             text: "select (Name, Balance) from Accounts ");

       // … and execute it on the default connection
       let result-set = execute(query);

       // extract the first record
       let first-record = result-set[0];
```

```
          // extract the first field of the first record.
          let first-name = result-set[0][0];
          let first-balance = result-set[0][1];
          let second-record = result-set[1];
          …
       end with-connection;
       disonnect(connection); // disconnect from the database
     end with-dbms;
   end;
```

6.4 Implementing CORBA objects in a server

A CORBA server has to provide an implementation object, called a servant, for each of the proxy objects that might be manipulated by a 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.*

6.4.1 Object adapters

An 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 Dylan ORB library 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

Harlequin Dylan interface to the POA conforms closely to the interface obtained by applying the Dylan 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 which servant the operation should be invoked on, and to 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 the 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 will 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 are used in this tutorial:

```
module PortableServer {

  native Servant;

  interface POAManager {
    exception AdapterInactive{};

    void activate() raises (...);
    …
  };
```

(see next page)

```
    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 **PortableServer**. That module 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 Dylan mapping, the **Servant** type maps to the abstract class **PortableServer/<Servant>**. User-defined Dylan classes that are meant to implement CORBA objects and be registered with a POA must inherit from this abstract 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 make requests for a POA to be queued or discarded, and can have the POA 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 POA, to enter the active state.

A POA object provides two useful operations that map between object references and servants: **`servant_to_reference`** and **`reference_to_servant`**.

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.

6.4.2 The server's perspective

From the perspective of the server, the Bank-Protocol library specifies the protocol that its servants must implement in order to satisfy the interfaces in the IDL file **`bank.idl`**. A partial implementation of this protocol resides in the Bank-Skeletons library generated by the IDL compiler. This library should be used by any application that wants to act as a server by providing an implementation for a CORBA object matching an interface in the **`bank.idl`** file.

The Bank-Skeletons library defines an abstract servant class for each of the protocol classes corresponding to an IDL interface. Each of these classes inherits from the abstract class **`PortableServer/<Servant>`**, allowing instances of these classes to be registered with a POA.

A server provides an implementation of an abstract servant class by defining a concrete subclass of that class, called an implementation class, and defining methods, specialized on the implementation class, for each of the protocol functions corresponding to an IDL attribute or operation.

The Bank-Skeletons library defines a concrete skeleton method, specialized on the appropriate abstract 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 Dylan condition corresponding to a CORBA user or system exception, the skeleton method sends the CORBA exception back to the client.

6.5 Requirements for implementing the bank server

As there were for the bank client, there are three parts to implementing the bank server:

- Write the code to initialize the CORBA ORB, set up the POA and POA manager, and get an initial object reference.
- Write the code for the CORBA objects that the server provides.
- Write the code for the server GUI.

We start by writing the CORBA object code. As noted in Section 6.4, this entails writing

6.5.1 The bank server GUI

Since this demonstration principally concerns CORBA, and because we would like to revamp the look-and-feel of the demonstration occasionally, we do not describe the GUI implementation in great detail. Instead, only a brief outline of the current design is given.

The bank server consists of one window that shows a table of raw account data. Each row in the table shows the name, the current balance, and the overdraft limit data.

There is also a log window for viewing incoming requests. The full implementation of the server GUI can be found in the file **`server-frame.dylan`**.

6.5.2 The bank server library and module

The bank server is implemented as a library:

```
define library bank-server
  use harlequin-dylan;
  use dylan-orb;
  use bank-skeletons;
  use sql-odbc;
  use duim;
  use threads;
  ...
end library bank-server;
```

that defines a single module:

```
define module bank-server
  use harlequin-dylan;
  use dylan-orb;
  use bank-skeletons;
  use sql-odbc;
  use duim;
  use threads;
  ...
end module bank-server;
```

Like the client, our server needs to use the Dylan-ORB system library and module, in addition to its application specific libraries. Because the server provides implementations (or servants) for CORBA objects satisfying interfaces defined in the **bank.idl** file, it also needs to use the Bank-Skeletons library and module.

Interoperating with ODBC requires the SQL-ODBC library and module.

Finally, our implementation of the server makes non-essential use of the DUIM and Threads libraries and modules to present the user with a dialog to shutdown the server. The full source code for the server is in file **bank-server.dylan**.

6.5.3 Implementing the servant classes

The Bank-Skeletons library defines three abstract servant classes:

```
BankingDemo<account-servant>
BankingDemo/<checkingAccount-servant>
BankingDemo/<bank-servant>
```

These classes correspond 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 declared in the IDL.

Each class inherits from the abstract class **`PortableServer/<Servant>`**. This allows instances of the class to be registered with a POA.

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

```
<bank-implementation>
<account-implementation>
<checkingAccount-implementation>
```

The **`<bank-implementation>`** class implements **`BankingDemo/<bank-servant>`** by representing a bank as a connection to a database:

```
define class <bank-implementation> (BankingDemo/<bank-servant>)
  slot connection :: <connection>,
           required-init-keyword: connection:;
  constant slot poa :: PortableServer/<POA>,
           required-init-keyword: poa:;
  constant slot name :: CORBA/<string>,
           required-init-keyword: name:;
end class <bank-implementation>;
```

The bank implementation class includes the slot **`poa`** 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.

The **`<account-implementation>`** class implements **`BankingDemo/<account-servant>`**:

```
define class <account-implementation>
    (BankingDemo/<account-servant>)
  constant slot bank :: <bank-implementation>,
    required-init-keyword: bank:;
  constant slot name :: CORBA/<string>,
    required-init-keyword: name:;
end class <account-implementation>;
```

An instance of this class represents an account. The **`bank`** slot provides a connection to the database that holds the account's record. The **`name`** slot identifies the record in the database.

Finally, the **`<checkingAccount-implementation>`** class implements **`BankingDemo/<checkingAccount-servant>`** simply by inheriting from **`<account-implementation>`**:

```
define class <checkingAccount-implementation>
  (<account-implementation>,
   BankingDemo/<checkingAccount-servant>)
end class <checkingAccount-implementation>;
```

6.5.4 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.

To support this, the abstract servant classes,

```
BankingDemo/<account-servant>
BankingDemo/<checkingAccount-servant>
BankingDemo/<bank-servant>
```

are defined to inherit, respectively, from the abstract protocol classes

```
BankingDemo/<account>
BankingDemo/<checkingAccount>
BankingDemo/<bank-servant>
```

As a result, implementing a protocol function boils down to defining a concrete method for that function, where the method 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 **`BankingDemo/account/name`** method returns the value of the account's **`name`** slot:

```
define method BankingDemo/account/name
    (account :: <account-implementation>)
    => (name :: CORBA/<string>)
  account.name;
end method BankingDemo/account/name;
```

The **`BankingDemo/account/balance`** method retrieves the balance field from the corresponding record on the database by executing an SQL **`select`** statement:

```
define method BankingDemo/account/balance
    (account :: <account-implementation>)
    => (balance :: CORBA/<long>)
  with-connection(account.bank.connection)
    let query = make(<sql-statement>,
                     text: "select Balance  from Accounts "
                           "where Name = ?");
    let result-set = execute(query,
                             parameters: vector(account.name));
    as(CORBA/<long>, result-set[0][0]);
  end with-connection;
end method BankingDemo/account/balance;
```

The **BankingDemo/account/balance** method increments the record’s balance field by executing an SQL **update** statement:

```
define method BankingDemo/account/credit
    (account :: <account-implementation>,
     amount :: CORBA/<unsigned-long>)
    => ()
  with-connection(account.bank.connection)
    let amount = abs(amount);
    let query = make(<sql-statement>,
                     text: "update Accounts "
                           "set Balance = Balance + ? "
                           "where Name = ?");
    execute(query, parameters: vector(as(<integer>, amount),
            account.name));
  end with-connection;
end method BankingDemo/account/credit;
```

The **BankingDemo/account/debit** method executes an SQL **update** statement that decrements the record’s balance field, provided the balance exceeds the desired amount:

```
define method BankingDemo/account/debit
    (account :: <account-implementation>, amount :: CORBA/<long>)
    => ()
  with-connection(account.bank.connection)
    let amount = abs(amount);
    let query = make(<sql-statement>,
                     text: "update Accounts "
                           "set Balance = Balance - ? "
                           "where Name = ? and Balance >= ?");
    execute(query,
            parameters: vector(as(<integer>, amount),
                               account.name,
                               as(<integer>, amount)));
  end with-connection;
end method BankingDemo/account/debit;
```

The **BankingDemo/checkingAccount/limit** method is similar to the **BankingDemo/account/balance** method defined above:

```
define method BankingDemo/checkingAccount/limit
    (account :: <checkingAccount-implementation>)
    => (limit :: CORBA/<long>)
  with-connection(account.bank.connection)
    let query = make(<sql-statement>,
                     text: "select Limit from Accounts "
                           "where Name = ?");
    let result-set = execute(query,
                             parameters: vector(account.name));
    as(CORBA/<long>, result-set[0][0]);
  end with-connection;
end method BankingDemo/checkingAccount/limit;
```

Because we defined **<checkingAccount-implementation>** to inherit from **<account-implementation>**, there is no need to re-implement the **BankingDemo/account/balance** and **BankingDemo/account/credit** methods for this implementation class. However, we do want to define a specialized **BankingDemo/account/debit** method, to reflect that a checking account can be overdrawn up to its limit:

```
define method BankingDemo/account/debit
    (account :: <checkingAccount-implementation>,
     amount :: CORBA/<long>)
     => ()
  with-connection(account.bank.connection)
    let amount = abs(amount);
    let query = make(<sql-statement>,
                     text: "update Accounts "
                           "set Balance = Balance - ? "
                    "where Name = ? and (Balance + Limit) >= ?");
    execute(query,
            parameters: vector(as(<integer>, amount),
                               account.name, as(<integer>,
                               amount)));
  end with-connection;
end method BankingDemo/account/debit;
```

The **BankingDemo/bank/name** method returns the value of the bank's **name** slot:

```
define method BankingDemo/bank/name
    (bank :: <bank-implementation>)
  => (name :: CORBA/<string>)
  bank.name;
end method BankingDemo/bank/name;
```

The **BankingDemo/bank/openAccount** method illustrates how CORBA user exceptions are raised:

```
define method BankingDemo/bank/openAccount
    (bank :: <bank-implementation>, name :: CORBA/<string>)
    => (account :: BankingDemo/<account>)
  if (existsAccount?(bank, name))
    error (make(BankingDemo/bank/<duplicateAccount>));
  else
    begin
      with-connection(bank.connection)
        let query = make(<sql-statement>,
              text: "insert into Accounts(Name, Balance, Limit) "
                    "values(?, ?, ?)",
              input-indicator: #f);
        execute(query, parameters: vector(name, as(<integer>, 0),
                                          #f));
      end with-connection;
```

```
          let new-account = make(<account-implementation>,
                                 bank: bank, name: name);
          as(BankingDemo/<account>,
             PortableServer/POA/servant-to-reference(bank.poa,
                                                     new-account));
        end;
      end if;
    end method BankingDemo/bank/openAccount;
```

If the test **existsAccount?(bank, name)** succeeds, the call to

```
    error (make(BankingDemo/bank/<duplicateAccount>));
```

raises a Dylan condition. (We omit the definition of **existsAccount?**, 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 IDL **duplicateAccount** exception back to the client.

If there is no existing account for the supplied name, the **BankingDemo/bank/openAccount** method creates a new record in the database by executing an SQL **insert** statement, initializing the "Limit" field of this record with the SQL NULL value. (Recall that the presence of the NULL value serves to distinguish ordinary accounts from checking accounts on the database.)

Finally, the method makes a new servant of class **<account-implementation>**, registers it with the bank's POA with a call to **PortableServer/POA/servant-to-reference**, and narrows the resulting object reference to the more specific class **BankingDemo/<account>**, the class of object references to account objects, as required by the signature of the protocol function.

The **BankingDemo/bank/openCheckingAccount** method is similar, except that it initializes the **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>**:

```
define method BankingDemo/bank/openCheckingAccount
    (bank :: <bank-implementation>, name :: CORBA/<string>,
     limit :: CORBA/<long>)
    => (checkingAccount :: BankingDemo/<checkingAccount>)
  if (existsAccount?(bank, name))
    error (make(BankingDemo/bank/<duplicateAccount>));
  else
    begin
      with-connection(bank.connection)
        let limit = abs(limit);
        let query =
          make(<sql-statement>,
              text: "insert into Accounts(Name, Balance, Limit) "
                    "values(?, ?, ?)",
              input-indicator: #f);
        execute(query, parameters: vector(name, as(<integer>, 0),
                                          as(<integer>, limit)));
      end with-connection;
      let new-account = make(<checkingAccount-implementation>,
                             bank: bank, name: name);
```

(see next page)

```
      as(BankingDemo/<checkingAccount>,
      PortableServer/POA/servant-to-reference(bank.poa,
                                              new-account));
     end;
  end if;
end method BankingDemo/bank/openCheckingAccount;
```

The **BankingDemo/bank/retrieveAccount** method uses the **name** parameter to select the **Limit** field of an account record. If there is no record with that name, indicated by the query returning an empty result set, the method raises the CORBA user exception **nonExistentAccount** by signalling the corresponding Dylan error.

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

```
define method BankingDemo/bank/retrieveAccount
    (bank :: <bank-implementation>, name :: CORBA/<string>)
    => (account :: BankingDemo/<account>)
  with-connection(bank.connection)
    let query = make(<sql-statement>,
                     text: "select Limit from Accounts "
                           "where Name = ?",
                     output-indicator: #f);
    let result-set = execute(query, parameters: vector(name),
                             result-set-policy:
                               $scrollable-result-set-policy);
    if (empty? (result-set))
      error (make(BankingDemo/bank/<nonExistentAccount>));
    else if (result-set[0][0])
               as(BankingDemo/<checkingAccount>,
               PortableServer/POA/servant-to-reference(bank.poa,
                 make(<checkingAccount-implementation>,
                      bank: bank,
                      name: name)));
          else
            as(BankingDemo/<account>,
               PortableServer/POA/servant-to-reference(bank.poa,
                 make(<account-implementation>,
                      bank: bank,
                      name: name)));
          end if;
    end if;
  end with-connection;
end method BankingDemo/bank/retrieveAccount;
```

(Unlike the other queries in this example, this query is executed with **`result-set-policy: $scrollable-result-set-policy`** to ensure that testing the emptiness of the result set does not invalidate its records.)

Finally, the **`closeAccount`** removes the record of an account from the database by executing an SQL **`delete`** statement:

```
define method BankingDemo/bank/closeAccount
    (bank :: <bank-implementation>,
     account-reference :: BankingDemo/<account>)
    => ()
  let account =
    Portableserver/POA/reference-to-servant(bank.poa,
                                            account-reference);
  with-connection(bank.connection)
    let query = make(<sql-statement>,
                     text: "delete from Accounts "
                           "where Name = ?");
    execute(query, parameters: vector(account.name));
  end with-connection;
end method BankingDemo/bank/closeAccount;
```

Note that we need to dereference the object reference **`account`** that is passed in as the parameter of the **`BankingDemo/bank/closeAccount`** operation. We call the **`Portableserver/POA/reference-to-servant`** operation of the POA to do so. 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.

6.6 Implementing CORBA initialization for the bank server

To complete the implementation of the bank 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 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
- Prevent the process from exiting, providing availability

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 run();
    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 knows as an **`ObjectId`**. For instance, the **`ObjectID`** for an initial POA object is **`"RootPOA"`**. (References to a select few other objects, such as the **`"Interface Repository"`** and **`"NamingService"`**, 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`**.

The **`run`** and **`shutdown`** operations are useful in multi-threaded programs, such as servers, which, apart from the main thread, need to run a separate request receiver thread for each POA.

(A single threaded application, such as a pure ORB client, does not generally need to use these operations.)

A thread that calls an ORB's **`run`** operation simply waits until it receives notification that the ORB has shut down.

Calling **`run`** in a server's main thread can then be used to ensure that the server process does not exit until the ORB has been explicitly shut down.

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.

```
define method initialize-server ()

  let location-service = get-location-service();

  // get reference to ORB
  let orb = CORBA/ORB-init(make(CORBA/<arg-list>),
                       "Harlequin Dylan ORB");

  // get reference to root POA, initially in the holding state
  let RootPOA = CORBA/ORB/resolve-initial-references(orb,
                                                    "RootPOA");

  with-dbms ($dbms)
     // connect to the database
     let database = make(<database>,
                       datasource-name: $datasource-name);
     let user =  make(<user>, user-name: $user-name,
                     password: $user-password);
     let connection = connect(database, user);

     // make the server frame, initialize and refresh it.
     let server-frame = make(<server-frame>,
                           connection: connection);
     server-frame.refresh-check-button.gadget-value := #t;
     refresh(server-frame);

     //  make the bank servant
     let bank = make(<bank-implementation>,
                    connection: connection,
                    poa: RootPOA,
                    name: "Dylan Bank",
                    server-frame: server-frame);

     // get the servant's object reference from the poa
     let bank-reference =
          PortableServer/POA/servant-to-reference(bank.poa,
                                                  bank);

     // activate the bank's POA using its POA manager.
     let POAManager =
           PortableServer/POA/the-POAManager(bank.poa);
     PortableServer/POAManager/activate(POAManager);

     // register the bank with the location service
     register-bank(orb, location-service, bank-reference);
```

```
          // create a separate thread to shut down the orb,
          // unblocking the main thread.
          make(<thread>,
               function: method ()
                           start-frame(server-frame);
                           CORBA/ORB/shutdown(orb, #t);
                         end method);

          // block the main thread
          CORBA/ORB/run(orb);

          // remove from location service
          unregister-bank(orb, location-service, bank-reference);

          // close the bank's connection.
          disconnect(connection);
       end with-dbms;
     end method;
```

The **`initialize-server`** function first initializes the Harlequin Dylan ORB by calling the Dylan generic function **`CORBA/ORB-init`**, just as we initialized the ORB in the client. The call returns a **`CORBA/<ORB>`** pseudo object.

Invoking **`CORBA/ORB/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 RootPOA 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, RootPOA, and publish the resulting object reference, encoded as a string, according to the location-service requested in the command line arguments. By default this is via a shared file. However, if the following is specified on the command line:

```
-location-service:naming-service
```

then a Name Service is used instead. Use the ORB command line option **`-ORBname-service`** to specify the IOR of the Name Service. Be sure to use the same command line options for the client and the server or they will not find each other!

We then obtain the POA Manager for the POA using the POA operation **`PortableServer/POA/the-POAManager`**. The call to **`PortableServer/POAManager/activate`** moves the POA out of the holding state, into the *active* state, ready to receive and process incoming requests.

To prevent the server from exiting before having the chance to process any requests, we introduce a new thread. This thread waits until the user responds to a DUIM dialog and then proceeds to shut down the ORB with a CORBA standard call to **`CORBA/ORB/shutdown`**. Meanwhile, back in the main thread, the subsequent call to **`CORBA/ORB/run`** causes the main thread to block, waiting for notification that the ORB has shut down.

Once the ORB has shut down, the main thread resumes, closes the connection to the bank, and exits, terminating the server application.

The full implementation of the server initialization is in the file **`init-server.dylan`**.

This completes the description of our implementation of the server.

7

Creating CORBA Projects

7.1 About CORBA projects

CORBA projects, such as those we saw in previous examples, are just Dylan projects whose executable or DLL can operate as a CORBA server or client. Servers and clients built from CORBA projects use the Harlequin Dylan ORB to communicate with other CORBA-based applications across an interface specified in OMG IDL.

The most noticeable distinction between a CORBA project and other projects is that a CORBA project contains a Harlequin Dylan Tool Specification (*spec*) file which identifies an IDL file containing an IDL interface declaration.

Additionally, CORBA projects use protocol, stubs or skeletons projects that are generated from an IDL file. The combination of these generated projects used in a CORBA project depends on whether it is a server or client project. Automatic generation of client stubs and server skeletons from an IDL file is the standard way of providing a transparent native-language interface for writing clients and servers that can communicate with one another across an IDL interface using CORBA.

7.2 Creating CORBA projects

To create a CORBA project with the New Project wizard:

1. Choose **File > New...** from any window or click the New Project ( ) button in the main window.

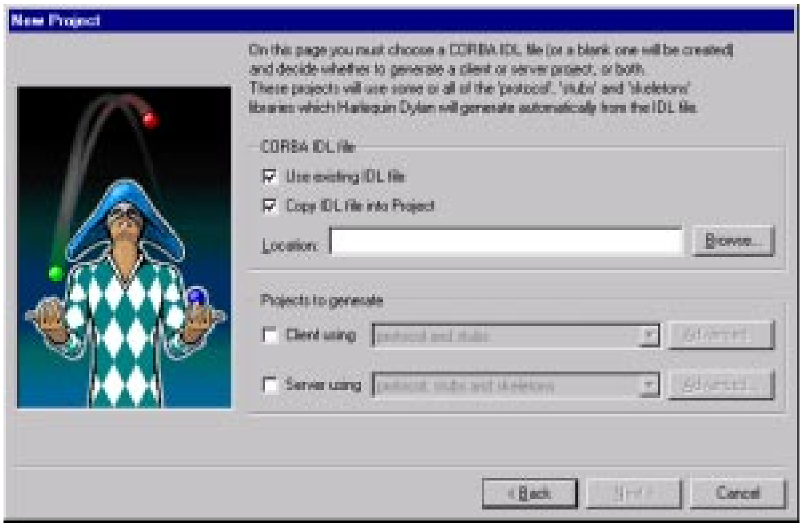
   If you chose **File > New...** from a project window, remember to uncheck the “Insert file into project?” box unless you intend your new project to be a subproject.

2. Select “Project” and click **OK**.

   The New Project wizard appears.

3. In the “Project Type” section, select “CORBA Client and / or Server” and click **Next**.

   The wizard’s CORBA options page appears.



On this page you must choose a CORBA IDL file (or a blank one will be created) and decide whether to generate a client or server project, or both.
These projects will use some or all of the 'protocol', 'stubs' and 'skeletons' libraries which Harlequin Dylan will generate automatically from the IDL file.

CORBA IDL file
☑ Use existing IDL file
☑ Copy IDL file into Project
Location:

Projects to generate
☐ Client using: protocol and stubs
☐ Server using: protocol, stubs and skeletons

Figure 7.1 Selecting an IDL file.

4. In the “CORBA IDL file” section, choose an IDL file on which to base the project.

The wizard copies the IDL file into the project folder which you choose on the next wizard page. If the IDL file is already in that folder, it is left there.

Alternatively, the wizard can generate a blank IDL file for you. Uncheck the “Use existing IDL file” to make that happen. The blank IDL file will be written into the project folder you choose on the next wizard page. The IDL file will have the same name as the project, with the extension **`.idl`**.

5. In the “Projects to generate” section, choose whether to generate a client or server project, or both.

The **Advanced...** button next to the client and server options produces a dialog containing several ORB-related options. See “Setting ORB options” on page 76 for details.

6. Click **Next**.

The next page is the standard wizard page for naming a project and specifying its location. (See Chapter 4 of *Getting Started with Harlequin Dylan* for details.)

The project name you specify here is not used literally when the wizard creates the project or projects. Instead, if you chose to create a client project, the wizard creates a project called *name*-**`client`**, where *name* is the project name you specified. If you chose to create a server project, the project will be called *name*-**`server`**.

Similarly, the project folder you specify is used as the overall project folder name, with the project files and sources for clients and servers in subfolders of the appropriate name. For a project folder **`phat`**, you get the following structure on disk (assuming you chose to create both client and server projects):

```
phat\
  client\
  server\
```

If you chose to create a blank IDL file for your project by unchecking the “Use existing IDL file” box, your blank IDL file is created in the top-level project folder:

```
phat\
  client\
  server\
  phat.idl
```

The client and server projects themselves are created in subfolders of the top-level project folder called **`client`** and **`server`**.

7. Enter a project name and location, and click **Next**.

   The wizard moves on to the Use libraries page. This and the remaining wizard pages are as for ordinary projects.

8. Complete the remaining pages in the wizard to finish specifying the non-CORBA parts of your project or projects.

Note: If you choose to generate client and server projects, note that the project settings and library choices you make in the wizard will apply to both generated projects. If you want the two projects to have different settings or use different libraries, you must create both separately. The only exception to this is your choice of Advanced ORB Settings, which can be set differently for client and server projects in the "Projects to generate" section of the wizard's CORBA options page. See "Setting ORB options" on page 76.

Note: The interface projects (that is, the client stubs, server skeletons, and protocol projects that provide a static interface to the main project's IDL) are not generated until you first build your client or server project. When you do so, the IDL compiler is invoked and the necessary interface projects are created automatically. See "How the spec file affects IDL compilation" on page 80 for details of what projects are generated and where.

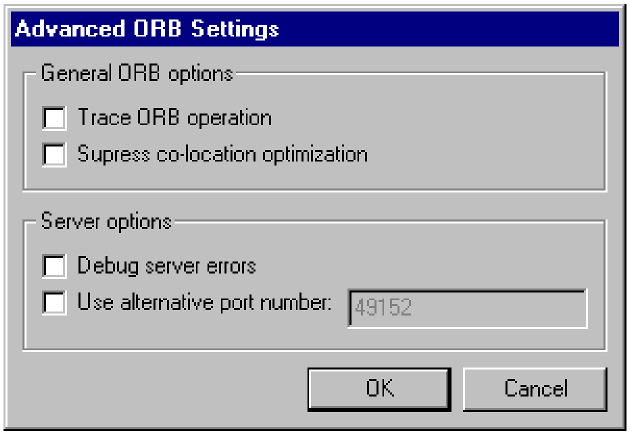
7.3 Setting ORB options

The Harlequin Dylan ORB is supplied as a Dylan library called **`dylan-orb`**, which both client and server CORBA projects use.

You can modify the default behavior of the ORB that a client or server project will run on while you are specifying the project in the New Project wizard. To set ORB options:

1. Click the appropriate **Advanced...** button on the New Project wizard's CORBA options page.

A client and server can use different settings, so there are two **Advanced...** buttons: one for a server project and one for a client project.


Advanced ORB Settings

General ORB options

☐ Trace ORB operation

☐ Supress co-location optimization

Server options

☐ Debug server errors

☐ Use alternative port number: 49152

Figure 7.2 The Advanced ORB settings dialog.

The options you set in the Advanced ORB Settings dialog are translated into command-line arguments and added to the Command Line in the **Project > Settings...** dialog's Debug page. This command-line specification what is used to invoke the EXE or DLL for the project when you choose **Application > Start**, **Debug**, or **Interact**. Thus, the Advanced ORB Settings arguments are passed to the server or client for use by the underlying ORB.

Note: Whether the ORB actually uses these settings is a different matter. Recall that, to initialize and gain an initial reference to an ORB, a client or server calls the **`corba/orb-init`** method. (The name is as mapped to Dylan from the pseudo-IDL **`CORBA::ORB_init`** operation.) If the client or server does not pass any arguments in this call, the ORB will use the arguments from the command-line. If some arguments are passed, those arguments will override the command-line arguments.

The options available are as follows.

Trace ORB Operation

Turns on debug messages inside the ORB. These messages are mainly internal debugging messages, but may help you to understand what is going on inside the ORB, or help you report problems to Harlequin technical support.

Corresponds to the ORB command-line argument **`-ORBtrace`**.

Suppress co-location optimization

Forces the ORB to always use sockets and IIOP for marshalling, even when it might have detected an in-process (that is, co-located) server that could be communicated with more directly.

Corresponds to the ORB command-line argument **`-ORBno-co-location`**.

Debug server errors

Suppresses handling of application implementation errors in server code. That is, instead of the errors being translated into CORBA exceptions for transmission to the client, they are left unhandled in the server so that they can be debugged.

Corresponds to the ORB command-line argument **`-ORBdebug`**.

Use alternative port number

Sets the default socket port for listening. The default port number registered with IANA for the Harlequin Dylan ORB is 3672.

Corresponds to the ORB command-line argument **`-ORBport`** *port-number*.

7.4 The role of spec files in IDL compilation

Harlequin Dylan Tool Specification files, or *spec files* (extension: **`.spec`**), allow tools other than the Dylan compiler to be invoked on a project at build time. When the build encounters a spec file in a project's sources list, it examines its contents and invokes the tool that the file specifies.

In the Harlequin Dylan implementation, you can use spec files to call the Harlequin Dylan IDL compiler, Scepter, on an IDL file. This compilation occurs during an ordinary project build, along with the compilation of Dylan source files.

Spec files are ASCII-format files that you can write or modify yourself. The following subsections describe the format for writing spec files for CORBA projects.

To arrange for IDL compilation to occur during an ordinary project build, the spec file must be part of the project's sources list. The New Project wizard inserts a spec file into CORBA projects automatically. The spec file has the same name as the project, with the extension **`.spec`**.

The IDL compiler will not regenerate stubs, skeletons, or protocol projects if they are up to date with respect to the IDL source file. If a spec file is changed, this will force the IDL compiler to regenerate code from the IDL source even if the IDL source itself has not changed. This is necessary because some options in the spec file may cause the generated code to change, for example the **`IDprefix:`** keyword.

Including the spec file in the project is useful because it brings all the build work for a CORBA project together into a single build. Also, by bringing the IDL file for the project under the control of Harlequin Dylan's project management system, you no longer have to worry about keeping the client stubs, server skeletons and protocol libraries up to date with any changes to the IDL — Harlequin Dylan takes care of that.

7.4.1 How the spec file affects IDL compilation

As we have seen in previous chapters, the IDL compiler is capable of generating client stubs, server skeletons, and protocol support projects from an IDL file. When there is a spec file in the project being compiled, any projects the IDL compiler generates automatically become subprojects of the main project.

This automatic inclusion only occurs because of the spec file in the project. If you compiled the IDL file separately, using the IDL compiler on the command line (see Chapter 9, “Using the Dylan IDL Compiler”) the generated projects would not be made subprojects. You would have to add them to the main project by hand in order to make them subprojects.

By default, the generated projects are placed in subfolders of the folder containing the IDL file. If you used the New Project wizard to create the project you are compiling, and you chose to generate a blank IDL file (by unchecking the “Use existing IDL file” box), the IDL file will have been generated in the top-level project folder, and thus the IDL compiler will put any projects it generates in subfolders of the top-level project folder. This very conveniently puts all subprojects associated with your main project under one folder.

Again by default, the subproject folders are named **`protocol`**, **`stubs`** and **`skeletons`**. There are spec file options which allow some control over what the subfolders are called and also where they are located.

7.4.2 Header information for CORBA spec files

The first line of any spec file must contain the **`Origin:`** keyword. For describing IDL source files to CORBA projects, the value of this keyword must be **`OMG-IDL`**. Thus the first line of a spec file for IDL must be:

```
Origin: OMG-IDL
```

Also required in the header of this kind of spec file is the keyword **`IDL-file:`**, with a value that is a pathname for the IDL file that is to be compiled. The pathname is interpreted relative to the folder containing the spec file. As noted in Section 7.4.1, the folder containing the IDL file will be used to store any projects generated automatically from the IDL.

Note that a spec file can describe only one IDL file. However, projects can contain more than one spec file.

For example, this is the minimum spec file for describing an IDL file called **`bank.idl`** that is in the folder bank, a sibling folder of the folder containing the spec file.

```
Origin:     OMG-IDL
IDL-file:   ..\bank\bank.idl
```

7.4.3 Server keywords for CORBA spec files

To indicate that the CORBA project contains a server implementation for the IDL file, use the keyword **`Skeletons:`** with the value **`yes`**.

```
Skeletons: yes
```

This ensures that Scepter generates a server skeletons project from the IDL source. The server skeletons project is automatically made a subproject of the CORBA project.

7.4.4 Client keywords for CORBA spec files

To indicate that the CORBA project is a client of the service described by the IDL file, use the keyword **`Stubs:`** with the value **`yes`**. This ensures that Scepter compiler generates a client stubs project from the IDL source. The client stubs project is automatically made a subproject of the CORBA project.

```
Stubs: yes
```

Sometimes, the CORBA client also needs to use the server skeletons code generated from the IDL source; for example, where the IDL defines a callback interface which the client must implement. In this case, you should also include the **`Skeletons:`** keyword in the spec file with the value **`yes`**.

```
Stubs: yes
Skeletons: yes
```

7.4.5 Other keywords for CORBA spec files

The following keywords control other aspects of IDL compilation.

`directory:` *directory*

By default the IDL compiler puts the Dylan projects it generates into subdirectories / subfolders in the same directory / folder as the IDL source file. You can force it to put the subdirectories / subfolders in another elsewhere using this option.

The *directory* can be an absolute or relative name. Relative names are resolved with respect to the directory / folder the spec file is in.

`prefix:` *directory-prefix*

The default names for the generated subdirectories / subfolders are **`protocol`**, **`stubs`** and **`skeletons`**.

You can specify a prefix for these names with this **`prefix:`** option. The prefix will be appended with a leading "`-`" character.

This might be useful when you want to put the project subdirectories / subfolders from more than one IDL file in a common location.

`idprefix:` *prefix-string*

Though the IDL compiler now supports **`#pragma`** directives, this option, designed as a substitute for the prefix pragma directive, remains available. It sets the prefix for generated repository IDs as if the first line of the IDL file were:

`#pragma prefix` *prefix-string*

`include:` *directory*

Adds *directory* to the search path list which the preprocessor uses to find files mentioned in **`#include`** directives.

The *directory* can be an absolute or relative name. Relative names are resolved with respect to the directory / folder the spec file is in.

`case:` **`yes | no`**

By default the IDL compiler ignores case when recognising keywords and identifiers, but when this option is set to **yes**, the characters in identifiers and keywords must have exactly the same case as the identifier or keyword definition in order to be recognized.

Default value: **no**.

`clean:` **`yes | no`**

If **yes**, forces the IDL compiler to regenerate code irrespective of whether or not it thinks any generated code that exists is up to date with respect to the IDL source file.

Default value: **no**.

`protocol:` **`yes | no`**

If **yes**, this option will ensure the IDL compiler generates the protocol library from the IDL source. The protocol project will be added as a subproject of the CORBA project. This option should be used by projects that wish to use the Dylan framework generated from the IDL source but not the stubs or skeletons code.

Default value: **no**.

7.5 Using IDL for non-CORBA work

By default, when Scepter generates a stubs or skeletons project from an IDL file, it also generates a *protocol* project. The protocol project contains Dylan code mapped directly from the IDL file: open generic functions and open classes. The protocol project does not contain any low-level CORBA marshalling code, which is left to the stubs and skeletons projects.

Thus the protocol project contains a pure Dylan protocol for implementing a piece of software according to an IDL contract. This can be useful for projects wishing to make use of the protocol without necessarily using an ORB for communication. For example, an application team might want to introduce the discipline of an IDL description early on in the project lifecycle so that development work on the clients and servers can proceed in parallel using dummy local implementations of the other components.

To have Scepter generate a protocol project only, choose the appropriate option in the Projects to Generate area of the New Project wizard's CORBA options page. Alternatively, after the project has been created you can modify the its spec file to use the **`Protocol:`** keyword with the value **`yes`**. Scepter will ensure the protocol project is generated, but will not generate the stubs or skeletons projects unless the **`Stubs:`** or **`Skeletons:`** keywords are also present and have the value **`yes`**.

The protocol project is automatically made a subproject of the CORBA project.

8

Running and Debugging CORBA Applications

8.1 Debugging client/server applications in the IDE

The following text is a quotation from the Harlequin Dylan manual *Getting Started with Harlequin Dylan*. See Chapter 6 of that manual for general information about running, debugging, and interacting with applications.

> If you have a client / server application, where both the client application and server application are written in Dylan, you can debug them in parallel.
>
> Start by opening both projects in the environment. It is not possible to run two instances of the environment, with one debugging the client and the other debugging the server: if any libraries are shared between the applications, both environment instances will attempt to lock the compiler database files for those libraries. Since all applications ultimately use the Dylan library, and most share other libraries — not the least of which in this case being networking libraries — using two Harlequin Dylan processes is never a practical debugging method.
>
> This is not a disadvantage. By running both client and server in one Harlequin Dylan, you can be debugging in the client, and then when the client invokes the server you can smoothly start debugging that instead. This can be very useful for tracking down synchronization bugs.

Once you have both projects open, you can start both applications up. Note that by default the action of starting a project will switch the active project, so the last project you start will be the active one by default. You can change this behavior in the main window with **Options > Environment Options…** so that the active project does not switch in this situation. See “The active project” on page 111 [of *Getting Started with Harlequin Dylan*] for more information.

If you need to rebuild a library shared between the client and server, you need to stop both running applications, since Windows forbids writing to a DLL that is currently in use.

Be careful when setting breakpoints if the client and server library share source files. If you set a breakpoint when editing a shared file, the breakpoint will be set in the editor’s active project. You can change the active project using the popup in the main window.

Breakpoints set in other windows’ source pages (such as in the browser) act on the project associated with the window. Note that this makes it possible to set breakpoints in both the client and the server so that the debugger correctly opens up on the appropriate project as the breakpoints are reached. However, you cannot set the same breakpoint in both projects at once. Instead you have to go into each project and set the breakpoint separately.

8.2 Browsing for supported CORBA operations

Until we have a CORBA reference manual for Harlequin Dylan, you can work out what CORBA PIDL operations are available to Dylan applications using the **`dylan-orb`** library by using the browser in the development environment.

1. In a CORBA project window, go to the Libraries page.
2. Double-click on the **`dylan-orb`** item.

   This browses the library **`dylan-orb`**.
3. In the browser, go to the Definitions page.
4. Double-click on the **`module dylan-orb:dylan-orb`** item.

   This browses the **`dylan-orb`** library’s **`dylan-orb`** module.

5. Go to the Names page.

Having done this you can see the names in the module. Use the pop-up list to show only the locally exported names — the names exported from this module.

The locally exported names contain the PIDL translations into Dylan as well as various Dylan protocols for using CORBA. These are the names that the browser's Module column shows were imported from the **`corba-protocol`** module of the **`corba-protocol`** library.

8.3 ORB runtime

8.3.1 Implicit activation

Servants may be returned from operations and if the POA ID Assignment policy is set to **`#"System-ID"`**, then they are automatically converted to object references. For instance, instead of

```
as(<account>, create-reference(poa, make(<demo-account>, ...)))
```

you can simply use:

```
make(<demo-account>, ...)
```

8.3.2 Port assignment

Socket port numbers are assigned dynamically, by default. If you wish to use the original (1.2 version) fixed port number of the Harlequin Dylan ORB, specify the following on the command line:

```
-ORBport orb
```

As of version 2.0, this port number is reserved for use by the Harlequin Dylan ORB's Implementation Repository. However, you can still explicitly select a port number, for example:

```
-ORBport 9999
```

8.3.3 POA threading

The Harlequin Dylan ORB has one request processing thread per POA, by default. You can set the number of request processing threads when creating a POA by using the thread policy, for example:

```
thread-policy: 4
```

This creates a pool of worker 4 threads that service requests dispatched to that POA. Therefore, up to 4 requests could be processed concurrently. In such a situation the underlying application code that is called from the POA must be multi-thread safe.

8.3.4 ORB runtime switches

The full set of command line arguments that the Dylan ORB supports is listed below. When running the application under the control of Harlequin Dylan, these command-line options can be set using the Debug page of the **Project > Settings…** dialog, in the Arguments field of the Command Line section.

`-ORBtrace` Turns on debug messages inside the ORB. These are mainly internal debugging messages, but may help you to understand what is going on inside the ORB, or help you report problems to technical support.

`-ORBdebug` Suppresses handling of application implementation errors in server code. That is, instead of them being translated into CORBA exceptions for transmission to the client, they are left unhandled in the server so that they can be debugged.

`-ORBport` *number*

Sets default socket port for listening. The default port number registered with IANA f2.

`-ORBid` *name*

Sets name of ORB.

`-ORBno-co-location`

Suppresses co-location optimization. That is, forces the ORB to always use sockets and IIOP marshalling even when it might have detected an in-process server.

The next command line options are concerned with the initial services offered by the ORB.

`-ORBname-service-file` *filename*

Sets filename containing IOR for name service. The string in the file is converted to an object reference and returned by

`CORBA/ORB/ResolveInitialServices("NameService")`

This option persists from session to session via the Windows Registry.

`-ORBname-service` *ior* (*)

Sets IOR for name service (takes precedence over file-based alternative above). The string is converted to an object reference and returned by

`CORBA/ORB/ResolveInitialServices("NameService")`

This option persists from session to session via the Windows Registry.

`-ORBinterface-repository-file` *filename* (*)

Sets filename containing IOR for interface repository. The string in the file is converted to an object reference and returned by

`CORBA/ORB/ResolveInitialServices("InterfaceRepository")`

This option persists from session to session via the Windows Registry.

`-ORBinterface-repository` *ior* (*)

Sets IOR for interface repository (takes precedence over file-based alternative above). The string is converted to an object reference and returned by

`CORBA/ORB/ResolveInitialServices("InterfaceRepository")`

This option persists from session to session via the Windows Registry.

`-ORBsettings` Prints out a list of the configuration options to the standard output.

9

Using the Dylan IDL Compiler

9.1 Introduction

The Harlequin Dylan IDL compiler, Scepter, is available as a standalone executable called **console-scepter.exe**. The executable is available in the **Bin** subfolder of your top-level Harlequin Dylan installation folder.

The IDL compiler reads an IDL file and generates three Dylan projects from it. The projects contain stub and skeleton code. Each project defines one of the libraries specified in the document *An IDL Binding for Dylan*. (See Section A.2 on page 98.)

The project's name is identical to the name of the library which it defines. By default the projects are placed in subfolders of the current working folder called **protocol**, **stubs** and **skeletons**.

9.2 General usage

The IDL compiler (**console-scepter.exe**) is invoked from an MS-DOS prompt. The general form of the command is:

```
console-scepter [options] file
```

which compiles the IDL file *file*. Only one filename can be supplied. Options can be prefixed with the character "**/**" or "**-**". Options that take a value must be separated from the value by a "**:**" character or whitespace.

The following invocation compiles the file **bank.idl**:

```
C:> console-scepter bank.idl
```

9.3 Code generation options

/language:*name*

The **/language** option is deprecated, but is still supported for backward compatibility. It has been replaced with the **/back-end** option.

/back-end:*name*

The **/back-end** option selects which back-end the compiler should use on the IDL file. Two values are supported:

dylan – This is the default value. The Dylan back-end generates Dylan protocol, stubs and skeletons libraries.

ir – This value causes the compiler to load the definitions in the IDL file into an Interface Repository. It will overwrite any definitions in the repository with the same scoped names. The compiler uses a call to the standard CORBA ORB operation **ResolveInitialReferences** to obtain a reference to the Interface Repository. Note that Harlequin Dylan does not have its own Interface Repository, but you can use one supplied by a third party. You may use the **-ORB-interface-repository-file** and **-ORBinterface-repository** options (described in Section 8.3.4, "ORB runtime switches") to specify an object reference for **ResolveInitialReferences** to return.

More than one **/back-end** option may be supplied if you want to load the IDL into an Interface Repository and generate Dylan code.

/parse Parse the IDL file only. This option overrides **/back-end**.

The following three options only apply to the Dylan back-end.

/directory:*dir* By default **console-scepter.exe** puts the Dylan projects it generates into subfolders in the current working folder. You can force it to put the subfolders in another folder *dir* by using the **/directory** option.

/prefix:*name* The default names for the subfolders are **protocol**, **stubs** and **skeletons**. You may specify a prefix for these names with the **/prefix** option. This might be useful, for example, where you want to put the project subfolders from more than one IDL file into a common folder.

/stubs By default **console-scepter.exe** generates three libraries for each IDL file: a protocol library, a stubs library, and a skeletons library. However, you may not always want to generate both the stubs and skeletons libraries. For example if you are developing a client application you will only need the stubs library. The **/stubs** option causes **console-scepter.exe** to generate only the protocol and stubs libraries.

9.4 Preprocessor options

/preprocess Runs the preprocessor on the IDL file and displays the result on standard output. The preprocessor output is not compiled.

/define:*name***[=***value***]**

Define the macro *name*. If the optional value is supplied the name is defined to have that value.

/undefine:*name*

Undefine the macro *name*.

/include:*dir*

Adds *dir* to the search path list which the preprocessor uses to find files mentioned in **#include** directives. More than one **/include** option may be supplied.

9.5 Misc options

/help	Display the command-line usage message.
/version	Display version information.
/debugger	Enter a debugger if console-scepter crashes.
/case	By default console-scepter ignores case when recognizing keywords and identifiers, but this option requires the characters in identifiers and keywords to have exactly the same case as the identifer or keyword definition in order to be recognised.
/nowarnings	Suppress compilation warning messages.
/trace	Trace compilation stages.

9.6 Examples

The following example would compile the bank demo IDL file. The generated projects are placed in the subfolder called **bank-protocol**, **bank-stubs** and **bank-skeletons** in the current working folder.

```
console-scepter /prefix:bank bank.idl
```

The next example would compile the bank demo IDL file and place the generated projects in the folders **c:\bank-client\protocol**, **c:\bank-client\stubs** and **c:\bank-client\skeletons**.

```
console-scepter /language dylan /directory c:\bank-client
bank.idl
```

The following example loads the Interface Repository with the definitions in the Bank demo IDL file:

```
console-scepter /back-end ir bank.idl
```

You can generate the Dylan projects at the same time using:

```
console-scepter /back-end:ir /back-end:dylan bank.idl
```

Appendix A

An IDL Binding for Dylan

Version 1.0 (Draft OMG Request For Comment submission)

A.1 Introduction

This chapter proposes an IDL binding for Dylan. It is based on the draft OMG Request For Comment submission.

A.1.1 Document conventions

The requirements of this specification are indicated by the verb “shall”. All other statements are either explanatory, amplifying, or relaxing. All implementation notes are so labeled.

Dylan names within the text and all example Dylan code appears in fixed-point Courier font, `like this`.

A.1.2 Bibliography

This specification mentions the following documents.

[DP 97]	N. Feinberg, S. E. Keene, R.O. Mathews, P. T. Withington, *Dylan Programming*, Addison Wesley, 1997.
[DRM 96]	L. M. Shalit, *The Dylan Reference Manual*, Addison Wesley, 1996.
[HQN 98]	*Harlequin Dylan, Library Reference: C FFI and Win32*, Version 1.0, Part No. DYL-1.0-RM4, Harlequin, 1998.
[OMG 94.3.11]	T. J. Mowbray, and K. L. White, *OMG IDL Mapping for Common Lisp*, OMG 94.3.11, 1994.
[OMG 98.07.01]	*The Common Object Request Broker: Architecture and Specification*, Revision 2.2, formal/98-07-01, <URL:http://www.omg.org/corba/c2indx.htm>, July 1998.

Table A.1 Bibliography.

A.2 Design rationale

A.2.1 Glossary of terms

This document uses terms from both the CORBA 2.2 specification [OMG 98.7.1] and the Dylan Reference Manual [DRM 96]. Any additional terminology is described below.

A.2.2 Design philosophy

A.2.2.1 Linguistic requirements

The design should be:

Conformant	Provide a mapping that conforms to the CORBA 2.0 specification.

Complete	Provide a complete language mapping between IDL and Dylan.
Correct	Provide a mapping that correctly maps legal IDL definitions to equivalent Dylan definitions.
Consistent	Provide a mapping that is consistent in its translation of IDL constructs to Dylan constructs.
Natural	Provide a mapping that produces Dylan definitions that would be judged to be expressed in "natural Dylan".
Stable	Small changes in an IDL description should not lead to disproportionately large changes in the mapping to Dylan.
Concise	Common constructs should be mapped in as simple and direct a manner as possible. If necessary to achieve this goal, rare pieces of syntax can be traded off and absorb the linguistic fallout.

A.2.2.2 Engineering requirements

The design should be:

Extension-free	Do not require extensions to CORBA.
Implementation-Free	Do not provide implementation descriptions, except as explanatory notes.
Reliable	The mapping should not adversely impact the reliability of clients or implementations of services built on an ORB.
Efficient	The mapping should allow for as efficient an implementation as is provided by other language bindings.

Portable	The mapping should use standard Dylan constructs.
Encapsulated	The mapping should hide and not constrain the implementation details.

A.2.2.3 Miscellaneous requirements

The design should also be:

Rationalized	Provide a rationale for key design decisions.

A.2.3 Mapping summary

The following table summarizes the mapping of IDL constructs to Dylan constructs.

This IDL construct …	… Maps to this Dylan construct
`Foo_Bar`	**`Foo-Bar`**
`Foo::Bar`	**`Foo/Bar`**
`Foo + Bar`	**`Foo + Bar`**[a]
`filename.idl`	**`define library filename-protocol`** … **`define library filename-stubs`** … **`define library filename-skeletons`** …
`module Foo { … Bar … }`	**`Foo/Bar`**
`interface Foo { … }`	**`define open abstract class <foo> (<object>)`** …
`interface Foo : Bar { … }`	**`define open abstract class <foo> (<bar>)`** …
`const long FOO` …	**`define constant $FOO`** ...

Table A.2 IDL constructs mapped to Dylan constructs.

	`CORBA/<long>`
`typedef Foo ...`	`define constant <Foo> ...`

`enum Foo {Bar ... }`	`define constant <Foo>`
	`= type-union(singleton(#"Bar"), ...`

`struct Foo { ... }`	`define class <Foo> (CORBA/<struct>) ...`
`union Foo { ... }`	`define class <Foo> (CORBA/<union>) ...`
`sequence`	`CORBA/<sequence>`
`string`	`CORBA/<string>`
`array`	`CORBA/<array>`
`exception Foo { ... }`	`define class <Foo> (CORBA/<user-exception>) ...`
`void Foo ( ... )`	`define open generic Foo ( ... ) => ()`
`any`	`CORBA/<any>`

Table A.2 IDL constructs mapped to Dylan constructs.

Notes: [a] or equivalent expression or literal.

A.3 Lexical mapping

This section specifies the mapping of IDL identifiers, literals, and constant expressions.

A.3.1 Identifiers

A.3.1.1 Background

IDL identifiers are defined as follows [OMG 98.7.1]:

> An identifier is an arbitrarily long sequence of alphabetic, digit, and underscore (“_”) characters. The first character must be an alphabetic character. All characters are significant.

> Identifiers that differ only in case collide and yield a compilation error. An identifier for a definition must be spelled consistently (with respect to case) throughout a specification.
>
> …
>
> There is only one namespace for OMG IDL identifiers. Using the same identifier for a constant and an interface, for example, produces a compilation error.

Dylan identifiers defined are as follows [DRM 96]:

> A name is one of the following four possibilities:
>
> > An alphabetic character followed by zero or more name characters.
> >
> > A numeric character followed by two or more name characters including at least two alphabetic characters in a row.
> >
> > A graphic character followed by one or more name characters including at least one alphabetic character.
> >
> > A "\" (backslash) followed by a function operator.

where:

> Alphabetic case is not significant except within character and string literals.
>
> …
>
> An alphabetic character is any of the 26 letters of the Roman alphabet in upper and lower case.
>
> A numeric character is any of the 10 digits.
>
> A graphic character is one of the following: ! & * < = > | ^ $ % @ _
>
> A name character is an alphabetic character, a numeric character, a graphic character, or one of the following: - + ~ ? /

A.3.1.2 Specification

From Section A.3.1.1 it can be seen that Dylan identifiers are a superset of IDL identifiers, and therefore they shall be left unmodified in the mapping except as follows.

IDL provides only underscores to separate individual words, while Dylan identifiers conventionally use hyphens to separate individual words. A mapping shall therefore translate underscores to hyphens.

There are some reserved words in Dylan:

> A reserved word is a syntactic token that has the form of a name but is reserved by the Dylan language and so cannot be given a binding and cannot be used as a named value reference. There are seven reserved words in Dylan: **`define`**, **`end`**, **`handler`**, **`let`**, **`local`**, **`macro`**, and **`otherwise`**.

When an IDL identifier collides with a reserved Dylan word, the string “**`-%`**” shall be appended to the end of the identifier. The “**`%`**” character is used instead of, say, the string “**`idl`**”, because it cannot occur in an IDL identifier, and so we avoid having to deal with cases where the Dylan identifier together with an existing “**`-idl`**” suffix also appears in the IDL description.

This “**`-%`**” suffix shall also be added to IDL identifiers ending in “**`-setter`**” in order to prevent potential collisions with setter functions mapped from IDL attributes.

There are further grammar-driven modifications of identifiers due to scope and convention described in Section A.4, “The mapping of IDL to Dylan”.

A.3.1.3 Examples

Some example mappings of IDL identifiers to Dylan identifiers:

This IDL identifier…	… maps to this Dylan identifier
`fusion`	**`fusion`**

Table A.3 IDL identifiers mapped to Dylan identifiers.

This IDL identifier…	… maps to this Dylan identifier
`Fusion`	**`Fusion`** [a]
`cold_fusion`	**`cold-fusion`**
`let`	**`let-%`**
`RED_SETTER`	**`RED-SETTER-%`**
`isExothermic`	**`isExothermic`**[b]

Table A.3 IDL identifiers mapped to Dylan identifiers.

Notes: [a] The case of all characters is not lowered in order to avoid modifying acronyms like: **`do_TLA`** -> **`do-tla`**.

[b] Similarly hyphens are not inserted at lower-to-upper case boundaries in order to avoid mistaken translations like **`LaTex_Parser`** -> **`la-te-x-parser`**.

A.3.2 Literals

IDL literals shall be mapped to lexically equivalent Dylan literals or semantically equivalent Dylan expressions. The following subsections describe the mapping for integers, floating-point numbers, and characters.

A.3.2.1 Integers

Background

Integer literals are defined as follows in IDL [OMG 98.7.1]:

> An integer literal consisting of a sequence of digits is taken to be decimal (base ten) unless it begins with 0 (digit zero). A sequence of digits starting with 0 is taken to be an octal integer (base eight). The digits 8 and 9 are not octal digits. A sequence of digits preceded by **`0x`** or **`0X`** is taken to be a hexadecimal integer (base sixteen). The hexadecimal digits include **`a`** or **`A`** through **`f`** or **`F`** with decimal values ten through fifteen, respectively.

The corresponding integer literals are defined as follows in Dylan:

- A sequence of decimal digits denote a decimal number.

- The characters “`#o`” followed by a sequence of octal digits denote an octal number.
- The characters “`#x`” followed by a sequence of hexadecimal digits denote a hexadecimal number.

Specification

A mapping shall therefore prepend the characters “`#o`” to the beginning of an octal literal. For a hexadecimal literal a mapping shall therefore remove the characters “`0x`” or “`0X`” from the beginning and prepend the characters “`#x`” in their place.

A.3.2.2 Floating point numbers

Background

Floating point literals are defined as follows in IDL [OMG 98.7.1]:

> A floating-point literal consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (but not both) may be missing; either the decimal point or the letter `e` (or `E`) and the exponent (but not both) may be missing.

The corresponding floating point literals are defined similarly in Dylan:

floating-point:
  sign$_{\texttt{opt}}$ ***decimal-integer***$_{\texttt{opt}}$ `.` ***decimal-integer*** ***exponent***$_{\texttt{opt}}$
  sign$_{\texttt{opt}}$ ***decimal-integer*** `.` ***decimal-integer***$_{\texttt{opt}}$ ***exponent***$_{\texttt{opt}}$
  sign$_{\texttt{opt}}$ ***decimal-integer*** ***exponent***

***exponent*:**
  `E` ***sign***$_{\texttt{opt}}$ ***decimal-integer***

***sign*:**
  `one of + -`

The case of the exponent marker `E` is not significant.

Specification

No modification shall be made to floating point literals during translation.

A.3.2.3 Characters

Background

IDL character literals are single printing characters, or escape sequences, enclosed by single quotes. The escape sequences are as follows:

Description	Escape Sequence
newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
form feed	\f
alert	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\"
octal number	\ooo
hexadecimal number	\xhh

Table A.4 IDL character literal escape sequences.

Dylan character literals are defined as follows:

***character-literal*:**
    ' ***character*** **'**

character **:**
  `any printing character (including space) except for ' or \`
  `\` ***escape-character***
  `\ '`

escape-character **:**
  `one of \ a b e f n r t 0`
  `<` ***hex-digits*** **`>`**

Specification

A mapping shall leave a single printing character unmodified during translation. A mapping shall leave escape sequences unmodified except as follows:

Description	IDL Escape Sequence	Dylan Translation
vertical tab	\v	\<0B>
question mark	\?	?
double quote	\"	"
octal number	\ooo	\<hh>
hexadecimal number	\xhh	\<hh>

Table A.5 IDL character-literal escape sequences mapped to Dylan.

Background

IDL defines a string literal as follows:

> A string literal is a sequence of characters (as defined in “Character Literals” …) surrounded by double quotes, as in `"…"`. Adjacent string literals are concatenated. Characters in concatenated strings are kept distinct.

Dylan defines a string literal as follows:

string **:**
  `"` ***more-string***

more-string:
  string-character more-string
  `"`

string-character:
  `any printing character (including space) except for " or \`
  `\` *escape-character*
  `\ "`

Specification

A mapping shall leave string literals unmodified during translation except as follows. Escape sequences shall be modified in accordance with the specification for character literals, with one exception: `\"` is left unmodified.

A.3.3 Fixed point decimals

A.3.3.1 Background

IDL defines a fixed point decimal literal as follows:

> A fixed point decimal literal consists of an integer part, a decimal part, a fraction part, and a `d` or a `D`. The integer and fraction parts both consist of a sequence of decimal (base 10) digits. Either the integer part of the fractional part (but not both) may be missing; the decimal point (but not the letter `d` (or `D`)) may be missing.

Dylan has no defined fixed point decimal literal format.

A.3.3.2 Specification

A fixed point decimal literal shall be mapped to any available Dylan representation of the value.

A.3.4 Constant expressions

A mapping shall either interpret the IDL constant expression yielding an equivalent Dylan literal or build a Dylan constant expression that will yield the same value.

A.3.4.1 Operators

The IDL operators shall be interpreted as, or translated to, Dylan as defined by the following table. Note that the Dylan expressions will necessarily have whitespace around the operators even if the IDL expressions do not.

Operation	IDL	Dylan
Bitwise Or	`x | y`	`logior(x, y)`
Bitwise Xor	`x ^ y`	`logxor(x, y)`
Bitwise And	`x & y`	`logand(x, y)`
Bitwise Not	`~ x`	`lognot(x)`
Shift Left	`x << y`	`ash(x, y)`
Shift Right	`x >> y`	`ash(x, -y)`
Add	`x + y`	`x + y`
Subtract	`x - y`	`x - y`
Multiply	`x * y`	`x * y`
Divide (integer)	`x / y`	`truncate/(x, y)`
Divide (float)	`x / y`	`x / y`
Remainder	`x % y`	`modulo(x, y)`
Plus	`+ x`	`+ x`
Minus	`- y`	`- y`

Table A.6 IDL operators mapped to Dylan.

A.4 The mapping of IDL to Dylan

This section specifies the syntactic and semantic mapping of OMG IDL to Dylan. Unless otherwise noted, the mapping is applicable to both client-side and server-side interfaces. Issues specific to the server-side only are covered in clearly marked subsections.

A.4.1 Names

A.4.1.1 Identifiers

The lexical mapping of identifiers shall be as specified in Section A.3.1.2, "Specification"

A.4.1.2 Scoped names

Specification

Dylan has very different scoping rules from IDL. In particular, Dylan is not able to introduce new subordinate namespaces at all the linguistic points that IDL allows: files, modules, interfaces, structures, unions, operations, and exceptions. Except for files, the mapping shall handle this by appending together all the enclosing scope identifiers and the scoped identifier, separating them by forward slashes. See Section A.4.2, "IDL Files" for the mapping of IDL files to Dylan.

Rationale

This is basically what several other languages have done [OMG 98.7.1]. Dylan has the concept of modules, but these are more linguistically heavyweight than the nested scopes they would be trying to model. Modules would also not allow out-of-scope references in the way that IDL does through its scope delimiter "`::`".

The slash character is used instead of the hyphen character so in examples like the following the two IDL identifiers below do not clash after translation:

```
Moorcock::Michael
Moorcock_Michael
```

Examples

```
// IDL
eco::umberto
SOCIETIES::Secret::knights_templar

// Dylan
eco/umberto
SOCIETIES/Secret/knights-templar
```

A.4.2 IDL Files

Specification

An IDL file shall be mapped to three Dylan libraries each exporting a single module with the same name as its respective library.

The three libraries shall be given the same name as the original IDL file minus its `.idl` extension, but adding the suffixes `-protocol`, `-stubs` and `-skeletons` respectively.

The protocol library shall minimally use the Dylan library, or a library that uses it an re-exports its bindings. The stubs and skeletons libraries shall similarly minimally use the Dylan library, the Dylan-ORB library (see Section A.4.3), and the protocol library; and shall re-export the latter's bindings.

Unless otherwise specified, Dylan constructs introduced as part of the IDL mapping shall be created in the protocol library.

Rationale

The advantages of a mandatory mapping of a complete IDL description to a particular structure of Dylan libraries is:

- Libraries are the natural large-scale unit of reuse in Dylan, and will match expectations.
- Dylan programmers, applications, and tools will be able to rely on the Dylan "signature" of a service defined by an IDL description.
- Enforcing the creation of a libraries allows any required runtime support to be naturally separated into and used from subordinate libraries.

If multiple IDL files are required to be combined into a single trio of libraries, then a single top level file can be used to include them together with any extra IDL module declarations required to prevent name clashes.

For example, an application library wishing to invoke the operations described in the IDL file **`http.idl`** should use the **`http-stubs`** library. Similarly, an application library wishing to implement the operations described in the same IDL file should use the **`http-skeletons`** library.

The protocol library is available separately for applications wishing to make use of the Dylan framework generated from the IDL without necessarily using an ORB for communication. For example, an application team may wish to introduce the discipline of an IDL description early on in the project lifecycle so that development work on the clients and servers can proceed in parallel using dummy local implementations of the other components.

Implementation notes

An implementation is free to map an IDL file to as many Dylan source files as is convenient.

An implementation is encouraged to transfer comments from IDL source files to the generated Dylan files.

A.4.3 The DYLAN-ORB library

The Dylan mapping relies on some runtime support, for example the built in types like **`corba/<short>,`** and this shall be provided in a Dylan library called Dylan-ORB that shall be used by a Dylan library generated from an IDL file. Similarly, the Dylan-ORB library shall define and export a Dylan module called Dylan-ORB that shall be used by a Dylan module generated from an IDL file.

The Dylan-ORB library can be used independently of libraries generated from IDL files to build generic applications without specific knowledge of particular services.

A.4.4 Mapping modules

A.4.4.1 Background

IDL modules define a name scope for other declarations including subordinate modules.

A.4.4.2 Specification

An IDL module shall be mapped to a Dylan identifier prefix for identifiers declared in the scope of the module declaration as defined by Section A.4.1.2, “Scoped names”.

A.4.4.3 Rationale

Although mapping an IDL module to a Dylan module would seem to be more natural, Dylan modules do not support out-of-scope references to identifiers, for example, in IDL, **`foo::bar`**.

A.4.4.4 Examples

```
// IDL
module physics {
  module quantum_mechanics {
    interface schroedinger {};
  };
};
```

```
//  Dylan
define open class physics/quantum-mechanics/<schroedinger>
  (<object>)
end class;
```

Note: The next section covers the mapping of interfaces.

A.4.5 Mapping for interfaces

A.4.5.1 Background

The CORBA standard [OMG 98.7.1] states:

> An interface is a description of a set of possible operations that a client may request of an object. An object satisfies an interface if it can be specified as the target object in each potential request described by the interface.

In practice, an interface declaration introduces a name scope and defines a set of operations on, and attributes of, the interface.

A.4.5.2 Specification

An IDL interface shall be mapped to an *open*, *abstract*, Dylan class with no superclasses other than **`<object>`** and classes generated from inherited IDL interfaces.

The implementation dependent classes used to represent object references (see Section A.5.2.1, "Object references" and servants (see section Section A.5.5.1, "Servants") shall be subclasses of the open abstract classes, and shall be defined in the stubs and skeletons libraries respectively.

An IDL interface shall also be mapped to a Dylan identifier prefix for identifiers declared in the scope of the class declaration as defined by Section A.4.1.2. Angle brackets shall be added to the start and end of the Dylan class name in accordance with Dylan programming conventions.

A forward declaration of an IDL interface shall not be mapped to anything.

A.4.5.3 Rationale

Dylan classes are the natural focus of protocol definition and also allow IDL interface inheritance to be modeled by Dylan class inheritance (see below).

The class is *abstract* because it is an interface to an implementation. On the client side the class is uninstantiable since it is meaningless for Dylan client code to call the **`make`** generic function on an arbitrary remote class. Instead the client must acquire object references by invoking operations on factory objects as defined in the IDL description of the particular service concerned.

The class is *open* to allow users of the server module to implement the interface by subclassing.

The class has no superclasses other than **`<object>`** or those that might be mapped from inherited IDL interfaces in order that the protocol library, generated from the IDL interface, may be used independently of any runtimes, stubs, or skeletons, as an abstract Dylan protocol.

IDL allows name-only "forward declarations" of interfaces in order to allow interfaces to refer to one another. Two Dylan classes can refer to one another, but there is no special forward reference declaration form. Furthermore, the definition of the language only encourages implementations to support forward references, repeated definitions are not allowed, and bindings are not created in any prescribed order.

It therefore appears that there is nothing particular which the definition of the mapping can do concerning ordering or extra definitions, to ensure that mapped Dylan classes can be compiled by all Dylan implementations.

A.4.5.4 Examples

```
// IDL
interface T34 {};

// Dylan
define open abstract class <T34> (<object>)
end class;
```

A.4.6 Mapping for interface inheritance

A.4.6.1 Background

The CORBA standard [OMG 98.7.1] states:

> Interface inheritance provides the composition mechanism for permitting an object to support multiple interfaces. The principal interface is simply the most-specific interface that the object supports, and consists of all operations in the transitive closure of the interface inheritance graph.

CORBA interfaces can inherit from multiple other interfaces, and Dylan also supports multiple inheritance. However, Dylan's multiple inheritance is more constrained. The class precedence order must be consistent. That is, any pair of classes must be in the same order with respect to each other wherever they occur together.

A.4.6.2 Specification

Interface inheritance shall be mapped to class inheritance in Dylan. The superclass list for the resulting Dylan class shall be canonicalized using lexicographic order. That is, the order shall be alphabetic on the character set used for Dylan identifiers.

A.4.6.3 Rationale

Class inheritance is the natural means of sharing protocols in Dylan.

The superclass list needs to be canonicalized to avoid legal IDL interface inheritance lists mapping to illegal Dylan class precedence lists. Reordering cannot affect method selection for IDL operations because IDL explicitly prohibits this kind of overloading. An alphabetic order is as good as any.

A.4.6.4 Examples

```
// IDL
interface T34 : tank soviet_made {};

interface T48 : soviet_made tank {};

interface T1000 : T48 T34 {};

// Dylan
define open abstract class <T34> (<soviet-made>, <tank>)
end class;

define open abstract class <T48> (<soviet-made>, <tank>)
end class;

define open abstract class <T1000> (<T34>, <T48>)
end class;
```

A.4.7 Mapping for constants

A.4.7.1 Specification

IDL constant declarations shall be mapped to Dylan constant definitions. IDL constant expressions are mapped as defined in Section A.2.3, “Mapping summary”. In addition, a dollar character shall be added to front of the main identifier, but after the scope prefix, in accordance with Dylan programming conventions.

A.4.7.2 Examples

```
// IDL
module time {
  const unsigned long SECS_IN_100_YRS
    = 100 * 365 * 24 * 60 * 60;
};
```

```
// Dylan
define constant time/$SECS-IN-100-YRS
    :: CORBA/<unsigned-long>
  = 100 * 365 * 24 * 60 * 60;
// Or alternatively
define constant time/$SECS-IN-100-YRS
    :: CORBA/<unsigned-long>
= 3153600000;
```

A.4.8 Mapping for basic types

A.4.8.1 Overall

Background

IDL and Dylan both provide a number of basic built in types and the means of constructing and naming new types.

Specification

The mapping shall introduce new Dylan types for each CORBA type. The mapping of the new Dylan types to the built-in Dylan types shall be constrained as specified in the following sections, but within these constraints the mapping is implementation dependent.

A generic constraint shall be that legal Dylan literals within the ranges allowed by the type are allowed as values for these new Dylan types. That is, it shall not be necessary to call a constructor function on these values.

Rationale

This allows implementations of the mapping some latitude, but also allows CORBA applications written in Dylan to succinctly and portably declare their data types and gain such efficiency as is provided by the combination of the mapping implementation and the Dylan compiler. The CORBA-specific types also protect application source code against underlying changes, and ensure that the code automatically benefits from any improvements.

The literals constraint means that programmers can expect to be able to use, say, the literal “`2`” where a CORBA **`short`** is expected.

A.4.8.2 Integers

Background

IDL defines six types of integer with the following ranges:

IDL integer type	Range
`short`	-2^{15} .. 2^{15}-1
`long`	-2^{31} .. 2^{31}-1
`long long`	-2^{63} .. 2^{63}-1
`unsigned short`	0 .. 2^{16}-1
`unsigned long`	0 .. 2^{32}-1
`unsigned long long`	0 .. 2^{64}-1

Table A.7 IDL integer types.

Dylan has the class **`<integer>`** which is required to be at least 28 bits of precision. Overflow behavior is implementation defined.

Specification

All IDL integer types shall be mapped to the following Dylan classes.

IDL integer type	Dylan CORBA library integer type
`short`	**`CORBA/<short>`**
`long`	**`CORBA/<long>`**
`long long`	**`CORBA/<long-long>`**
`unsigned short`	**`CORBA/<unsigned-short>`**
`unsigned long`	**`CORBA/<unsigned-long>`**
`unsigned long long`	**`CORBA/<unsigned-long-long>`**

Table A.8 IDL and Dylan integer types.

These classes, in turn, shall be defined as aliases for, or subclasses of, some Dylan implementation's integer classes, and shall be capable of representing the specified range of values.

Rationale

The rationale is as given for the general case above. In this particular instance, although an individual Dylan compiler *could* convert a **`limited`** expression to the best concrete class that the runtime supports, this is not guaranteed. The runtime may have a good class for implementing a CORBA class, but the compiler may not be capable of translating a **`limited`** expression into it.

Even if the translation of the **`limited`** expression to the best runtime class was guaranteed, the expressions are quite long an cumbersome to use repeatedly in code, and an alias is convenient.

Examples

```
// IDL
const long DIM_OF_UNIV = 11;

// Dylan
define constant $DIM_OF_UNIV :: CORBA/<long> = 11;
```

```
// Some alternative binding implementations
define constant CORBA/<long> = <integer>;
define constant CORBA/<long> = <machine-word>;
define constant CORBA/<long> =
  limited(<integer>, min: -(2 ^ 31), max: (2 ^ 31) -1);
```

A.4.8.3 Floating-point numbers

Background

IDL defines three types of floating-point numbers:

IDL float type	Range
`float`	ANSI / IEEE 754-1985 single precision
`double`	ANSI / IEEE 754-1985 double precision
`long double`	ANSI / IEEE 754-1985 double-extended precision

Table A.9 IDL floating point number types.

The Dylan types **`<single-float>`**, **`<double-float>`**, and **`<extended-float>`** are intended to correspond to the IEEE types but may not.

Specification

The IDL floating-point types shall be mapped to Dylan as follows:

IDL float type	Dylan float type
`float`	**`CORBA/<float>`**
`double`	**`CORBA/<double>`**
`long double`	**`CORBA/<long-double>`**

Table A.10 IDL floating point types mapped to Dylan.

These classes, in turn, shall be aliases for or subclasses of the Dylan `<float>` class and shall be capable of representing the specified range of values.

Rationale

As above.

Examples

```
// IDL
const double E = 2.71828182845904523536;
const float LYRS_TO_ALPHA_CENTAURI = 4.35;

// Dylan
define constant $E :: CORBA/<double> = 2.71828182845904523536;
define constant $LYRS-TO-ALPHA-CENTAURI :: CORBA/<float> = 4.35;
```

A.4.8.4 Fixed-point decimals

Background

IDL fixed-point decimals are defined as follows:

> The **`fixed`** data type represents a fixed-point decimal number of up to 31 significant digits. The scale factor is normally a non-negative integer less than or equal to the total number of digits (note that constants with effectively negative scale, such as 10000, are always permitted.). However, some languages and environments may be able to accommodate types that have a negative scale or scale greater then than the number of digits.

Dylan has no defined fixed-point decimal type, but does have a rational type.

Specification

The IDL **`fixed`** type shall be mapped to the Dylan class **`CORBA/<fixed>`**, which shall be a subtype of the Dylan type **`<rational>`**.

Subtypes of the IDL **`fixed`** type of the form **`fixed<`***d***`,`***s***`>`**, where *d* is the number of digits and *s* is the scale, shall be mapped to Dylan types of the form

```
limited(CORBA/<fixed>, digits: d, scale: s)
```

The Dylan language operators and functions on rationals shall have methods defined on instances of **`CORBA/<fixed>`**.

In addition, instances of **`CORBA/<fixed>`** shall support the following functions:

```
CORBA/Fixed/digits(x :: CORBA/<Fixed>)
  => (digits :: CORBA/<unsigned-short>)

CORBA/Fixed/scale(x :: CORBA/<Fixed>)
  => (scale :: CORBA/<short>)

as(class == CORBA/<long-double>, x :: CORBA/<Fixed>)
  => (value :: CORBA/<long-double>)

as(class :: subclass(CORBA/<Fixed>),
                     x :: CORBA/<Fixed>)
  => (fixed :: CORBA/<Fixed>)

as(class :: subclass(CORBA/<Fixed>),
                    x :: CORBA/<long-double>)
  => (fixed :: CORBA/<Fixed>)

as(class :: subclass(CORBA/<Fixed>),
                    x :: CORBA/<long>)
  => (fixed :: CORBA/<Fixed>)
```

Rationale

This seems to be the natural mapping to Dylan and approximately mirrors the C++ mapping.

Examples

```
// IDL
const fixed<6,2> salary_increment = 0100.50d;

// Dylan
define constant $foo
    :: limited(CORBA/<Fixed>, digits: 6, scale: 2)
  = make(limited(CORBA/<Fixed>, digits: 6, scale: 2),
         digits: “100.5”);
```

A.4.8.5 Characters

Background

IDL characters are elements of the 8 bit ISO Latin-1 (8859.1) character set. Dylan’s characters are unspecified. Dylan has a **`<character>`** class and has three string classes: **`<string>`**, **`<byte-string>`**, and **`<unicode-string>.`** Objects of these string types have elements that are subtypes of **`<character>`**.

Specification

The IDL **`char`** type shall be mapped to the Dylan class **`CORBA/<char>`**, which will be an alias for or a subclass of the Dylan class **`<character>`**.

Rationale

As above.

Examples

```
// IDL
const char ALEPH = ’a’;

// Dylan
define constant $ALEPH :: CORBA/<char> = ’a’;
```

A.4.9 Wide characters

A.4.9.1 Background

IDL wide characters are implementation defined.

Dylan defines a **`<unicode-string>`** type.

A.4.9.2 Specification

The IDL **`wchar`** type shall be mapped to **`CORBA/<wchar>`**, which shall be a subclass of **`<character>`**. Instances of **`CORBA/<wchar>`** shall be allowed as elements of instances of **`<unicode-string>`**.

A.4.9.3 Rationale

The natural mapping to Dylan.

A.4.9.4 Examples

```
// IDL
const wchar ALEPH = ‘a’;

// Dylan
define constant $ALEPH :: CORBA/<wchar> = ‘a’;
```

A.4.9.5 Boolean values

Background

IDL **`boolean`** type can take the values **`TRUE`** or **`FALSE`**.

Dylan's **`<boolean>`** class similarly can take the values **`#t`**, and **`#f`**.

Specification

The IDL **`boolean`** type shall be mapped onto the Dylan **`CORBA/<boolean>`** class which shall be an alias for the Dylan **`<boolean>`** class.

Rationale

The extra CORBA prefix class is introduced for completeness and consistency, but there is no need to allow it to be a subclass of the built-in class.

Examples

```
// IDL
const boolean CANTORS_HYPOTHESIS = TRUE;

// Dylan
define constant $CANTORS-HYPOTHESIS :: CORBA/<boolean> = #t;
```

A.4.9.6 Octets

Background

An IDL octet is an 8-bit quantity that undergoes no conversion of representation during transmission.

Specification

The IDL **`octet`** type shall be mapped to the Dylan class **`CORBA/<octet>`** which shall be an alias for or a subclass of some Dylan implementation's integer class allowing values in the range 0 to 255.

Rationale

As above.

Examples

```
// IDL
const octet BOND_ID = 007;
```

```
// Dylan
define constant $BOND-ID :: CORBA/<octet> = #o007;
```

A.4.9.7 The “any” type

Background

The IDL **any** type permits the specification of values that can express any OMG IDL type.

Dylan is a dynamic language with runtime type information.

Specification

The IDL **any** type shall be mapped to a sealed Dylan class **CORBA/<any>** with *sealed* getter and setter generic functions and initialization keywords for the underlying value and the associated typecode:

```
define generic CORBA/any/type
    (any :: CORBA/<any>)
  => (typecode :: CORBA/<typecode>);

define generic CORBA/any/type-setter
    (typecode :: CORBA/<typecode>, any :: CORBA/<any>)
  => (typecode :: CORBA/<typecode>);

define generic CORBA/any/value
    (any :: CORBA/<any>)
  => (value :: <object>);

define generic CORBA/any/value-setter
    (value :: <object>, any :: CORBA/<any>)
  => (value :: <object>);
```

In addition, anywhere that an object of type **Any** is required the Dylan programmer can supply objects that are instances of any mapped IDL type.

At least one of the **value:** initialization and **typecode:** keywords shall be required. If the latter is not supplied then it is coerced from the value in an implementation defined manner.

Explicit coercion to and from objects of type **Any** shall be provided by *sealed* methods on the Dylan generic function **as**.

The function **CORBA/any/value** shall signal an error if the value cannot be coerced to a native Dylan type corresponding to a mapped IDL type.

Rationale

Although it is awkward for the Dylan programmer to have to deal with an explicit **`Any`** type, it allows the typecode to be preserved across requests and replies in cases where it matters.

In some cases, for example where the **`Any`** contains a structure whose Dylan type is unknown (to the current program) it is not possible for **`CORBA/any/value`** to return a meaningful value. In these cases the **`DynAny`** interface should be used to navigate the data inside the **`Any`**.

Examples

```
// IDL
long goedel_number(in any thing);

// Dylan
define open generic goedel-number (thing :: CORBA/<any>)
  => (result :: CORBA/<long>);
```

Note: The mapping of operations is described in more detail later.

A.4.10 Mapping for constructed types

A.4.10.1 Mapping for typedefs

Background

An IDL **`typedef`** declaration introduces aliases for a given type.

Dylan has a single namespace for identifiers and so no separate defining form is needed to introduce a new alias for a class.

Specification

An IDL **`typedef`** declaration shall be mapped to as many Dylan **`define constant`** definitions as there are *declarators* being introduced by the IDL declaration.

Examples

```
// IDL
typedef short mozart_symphony_no, layston_park_house_no;
```

```
// Dylan
define constant <mozart-symphony-no> = CORBA/<short>;
define constant <layston-park-house-no> = CORBA/<short>;
```

A.4.10.2 Mapping for enumeration type

Background

An IDL enumerated type consists of ordered lists of identifiers.

Specification

An IDL enumerated type shall be mapped to a type union of singleton symbol types. In addition, four *sealed* generic functions shall be defined on the enumerated type (as if in its scope) for traversing and comparing the enumerated values: **`successor`**, **`predecessor`**, **`<`**, and **`>`**.

It shall be an error to call these functions on symbols outside the enumeration.

Rationale

This is the straightforward implementation of enumerated types described in [DP 97]. We retain this basic format with a view to benefiting from the compiler optimizations encouraged in [DP 97]. However, we also specify successor, predecessor, and comparison functions for convenience.

Examples

```
// IDL
enum planet {Mercury, Venus, Earth, Mars,
             Jupiter, Saturn, Uranus, Neptune, Pluto};

// Dylan
define constant <planet>
  = apply(type-union,
      map(singleton, #(#“Mercury”, #“Venus”,
                       #“Earth”, #“Mars”, #“Jupiter”, #“Saturn”,
                       #“Uranus”, #“Neptune”, #“Pluto”)));

define generic planet/successor
(value :: <planet>) => (succ :: <planet>);

define generic planet/predecessor (value :: <planet>)
=> (pred :: <planet>);

define generic planet/<(lesser :: <planet>, greater :: <planet>)
=> (lesser? :: <boolean>);
```

```
define generic planet/> (greater :: <planet>, lesser :: <planet>)
=> (greater? :: <boolean>);
```

A.4.10.3 Mapping for structure type

Background

IDL defines a **`structure`** type that aggregates together multiple pieces of data of potentially heterogeneous types.

Dylan programmers define new classes for this purpose.

Specification

An IDL structure shall be mapped to a *sealed*, *concrete*, Dylan subclass of **`CORBA/<struct>`** together with pairs of *sealed* getter and setter generic functions and a required initialization keyword for each struct member. The initialization keywords shall be Dylan symbols mapped using the normal identifier mapping rules, but without any scope prefixes. It shall be an error to call the getter and setter functions on instances of types other than those mapped from the IDL structure. Furthermore the Dylan protocol functions **`make`** and **`initialize`** shall be sealed over the domain of the mapped class.

Rationale

The Dylan class, the getters, the setters, and initializers, are defined to be *sealed* in the anticipation that operations on structures are expected to be as efficient as possible without any need for extension.

There is no need to specify whether the getter and setter generic functions are defined as Dylan slots. The data may in fact be maintained in a foreign internal format convenient for network transmission.

The initialization keywords are required so as not to introduce complicated defaulting rules.

The superclass **`CORBA/<struct>`** is made explicit to allow **`instance?`** tests.

Examples

```
// IDL
struct meeting {
  string topic, venue, convenor;
  long date, duration;
  sequence<string> attendees, agenda, hidden_agenda,
minutes;
};

// Dylan (using slots)
define class <meeting> (CORBA/<struct>)
  slot meeting/topic :: CORBA/<string>,
      required-init-keyword: topic:;
    slot meeting/venue :: CORBA/<string>,
      required-init-keyword: venue:;
    slot meeting/convenor :: CORBA/<string>,
      required-init-keyword: convenor:;
    slot meeting/date :: CORBA/<long>,
      required-init-keyword: date:;
    slot meeting/duration :: CORBA/<long>,
      required-init-keyword: duration:;
    slot meeting/attendees ::
       limited(CORBA/<sequence>, of: CORBA/<string>)>,
       required-init-keyword: attendees:;
    slot meeting/agenda ::
       limited(CORBA/<sequence>, of: CORBA/<string>)>,
       required-init-keyword: agenda:;
    slot meeting/hidden-agenda ::
      limited(CORBA/<sequence>, of: CORBA/<string>)>,
      required-init-keyword: hidden-agenda:;
    slot meeting/minutes ::
      limited(CORBA/<sequence>, of: CORBA/<string>)>,
      required-init-keyword: minutes:;
end class;

define sealed domain make (singleton(<meeting>));
define sealed domain initialize (<meeting>);
```

A.4.10.4 Mapping for discriminated union type

Background

IDL defines a **union** type that allows data of heterogeneous types used interchangeably in places like parameters, results, arrays, and sequences. An explicit tag called a *discriminator* is used to determine the type of the data in a given object that is of the union type.

Dylan is a dynamic language with runtime type information and has no explicit tagging mechanism.

Specification

An IDL union type shall be mapped to a *sealed*, *concrete*, Dylan subclass of **`CORBA/<union>`** with pairs of *sealed* getter and setter functions and an initialization keyword for each union branch. Every mapped union shall also have the following sealed getter and setter functions:

```
corba/union/discriminator
corba/union/discriminator-setter
corba/union/value
corba/union/value-setter
```

and the following initialization keywords:

```
discriminator:
value:
```

It is an error to call these functions on instances of types other than those mapped from the IDL union definition. Furthermore the Dylan protocol functions **`make`** and **`initialize`** shall be sealed over the domain of the mapped class.

The initialization keywords shall be mapped as for structs. However, they are not required in the same manner. Instead, either the caller shall supply the **`discriminator:`** and the **`value:`** or an initialization keyword mapped from one of the branches.

In addition, wherever a union is required (for example, in the parameter of an operation) the Dylan programmer shall be able to give any Dylan object that is an instance of one of the types of the branches of the union.

Explicit coercion to and from a union shall also be available as *sealed* methods on the Dylan **`as`** generic function. It is undefined which discriminator is used in ambiguous cases.

Rationale

Although it is unnatural for a Dylan programmer to have to manipulate explicit union discriminators, there are ambiguous cases that require this explicit treatment. By reifying the union the Dylan programmer is given as much direct control as a static language provides, and yet can also use the implicit coercion and value getter to ignore the details if so desired.

It is not necessary to state whether the getter and setter functions are implemented by slots.

The superclass **`CORBA/<union>`** is made explicit to allow **`instance?`** tests.

Examples

```
// IDL
union RLE_entity switch (short) {
  case 1: long length;
  case 2: char character;
};

// Dylan (sample)
define class <RLE-entity> (CORBA/<union>)
end class;

define sealed domain make (singleton(<RLE-entity>));
define sealed domain initialize (<RLE-entity>);

define sealed method as
    (class == <RLE-entity>, length :: CORBA/<long>)
    => (object :: <RLE-entity>)
  make(<RLE-entity>, length: length);
end method;

define sealed method as
    (class == CORBA/<long>, object :: <RLE-entity>)
    => (length :: CORBA/<long>)
  RLE-entity/length(object);
end method;

define method RLE-entity/length (union :: <RLE-entity>)
    => (length :: CORBA/<long>)
  select (corba/union/discriminator(union))
    1 => corba/union/value(union);
    otherwise => error(...);
  end select;
end method;
```

```
define method RLE-entity/length-setter
    (length :: CORBA/<long>, union :: <RLE-entity>)
    => (length :: CORBA/<long>)
  corba/union/value(union) := length;
  corba/union/discriminator(union) := 1;
end method;

…
```

A.4.10.5 Mapping for sequence type

Background

IDL defines a **sequence** type. A sequence is a one-dimensional array with an element type, an optional maximum size (fixed at compile time), and a current length (determined at run time).

Dylan defines several sequence-like classes including **<sequence>** itself.

Specification

The IDL **sequence** type shall be mapped onto the a new Dylan **CORBA/<sequence>** class that shall be an alias for or a subclass of the Dylan **<stretchy-vector>** class. An element type shall be mapped to a **limited** type of **CORBA/<sequence>**. The maximum size is not modeled in Dylan and must be checked on marshalling.

Rationale

Dylan's **<stretchy-vector>** class appears closest in intent to IDL's **sequence** type.

Examples

```
// IDL
typedef sequence<long, CHAIN-MAX> chromosomes;

// Dylan
define constant <chromosomes> =
  limited(CORBA/<sequence>, of: CORBA/<long>);
```

A.4.10.6 Mapping for string type

Background

IDL defines a **`string`** type. A string is a one-dimensional array of all possible 8-bit quantities except null, with an optional maximum size.

A string is similar to a sequence of char.

Dylan defines **`<string>`**, **`<byte-string>`**, and **`<unicode-string>`** classes.

Specification

The IDL **`string`** type shall be mapped onto a Dylan **`CORBA/<string>`** class that shall be an alias for or a subclass of the Dylan **`<string>`** class.

Rationale

The **`<byte-string>`** class is not mandated in the mapping to retain the flexibility and efficiency of the underlying Dylan implementation. The effect of storing a null in a **`CORBA/<string>`** that is passed as an argument to a request is undefined.

Examples

```
// IDL
typedef string constellation;

// Dylan
define constant <constellation> = CORBA/<string>;
```

A.4.10.7 Mapping for wide string type

Background

The IDL **`wstring`** data type represents a sequence of **`wchar`**.

Dylan defines a **`<unicode-string>`** type.

Specification

The **`wstring`** type shall be mapped to the Dylan type **`CORBA/<wstring>`** which shall be an alias for **`<unicode-string>`**.

Rationale

The natural mapping to Dylan.

Examples

```
// IDL
typedef wstring local_name;
```

```
// Dylan
define constant <local-name> = CORBA/<wstring>;
```

A.4.10.8 Mapping for array type

Background

IDL defines an **array** type for multidimensional fixed-size arrays, with explicit sizes for each dimension.

Dylan has a similar **<array>** class.

Specification

The IDL array type shall be mapped onto the Dylan **CORBA/<array>** class. The new class shall be an alias for or a subclass of the Dylan **<array>** class. An element type shall be mapped to a limited type of **CORBA/<array>**.

Rationale

This is the straightforward, natural mapping, albeit hidden behind a CORBA-specific class for portability across implementations and versions.

Examples

```
// IDL
typedef long tensor[3][3][3];

// Dylan
define constant <tensor> =
  limited(CORBA/<array>,
    of: CORBA/<long>, dimensions: #(3,3,3));
```

A.4.11 Mapping for exceptions

A.4.11.1 Background

IDL defines exceptions as:

> … struct-like data structures which may be returned to indicate that an exceptional situation has occurred during the performance of a request.

Dylan defines a rich, object-oriented, condition signalling and handling facility.

A.4.11.2 Specification

IDL exceptions shall be mapped onto *sealed* Dylan conditions that are subclasses of **`CORBA/<user-exception>`**, which shall be a subtype of **`CORBA/<exception>`**, which itself shall be a subtype of the Dylan **`<condition>`** class. As with IDL structures, any members shall be mapped to pairs of *sealed* setter and getter generic functions and corresponding initialization keywords. It shall be an error to call these functions on instances of types other than those mapped from the IDL exception definition. Furthermore the Dylan protocol functions **`make`** and **`initialize`** shall be sealed over the domain of the mapped class.

Conditions shall be signalled in the standard Dylan manner by the CORBA runtime and not returned or passed as arguments.

Standard system exceptions shall be direct or indirect subclasses of **`CORBA/<system-exception>`** which shall be a subtype of **`CORBA/<exception>`**.

A.4.11.3 Rationale

This is the natural mapping of IDL exceptions into the Dylan language.

A.4.11.4 Examples

```
// IDL
exception melt_down {
short seconds_remaining;
};

// Dylan (using slots)
define class <melt-down> (CORBA/<user-exception>)
  slot melt-down/seconds-remaining :: CORBA/<short>,
  required-init-keyword: seconds-remaining:;
end class;

define sealed domain make (singleton(<melt-down>));
define sealed domain initialize (<melt-down>);
```

A.4.12 Mapping for operations

A.4.12.1 Background

IDL uses operations as the basic means by which CORBA-compliant programs communicate with each other. Operation declarations are akin to C function declarations, but they also have to deal with parameter directions, exceptions, and client contexts. All operations are defined within the scope of an *interface*.

Dylan programs call generic functions to communicate with other Dylan programs. The generic functions are implemented by methods.

A.4.12.2 Specification

An IDL operation shall be mapped to an *open* Dylan generic function. The generic function name is subject to the usual identifier translation specified earlier. It shall be an error to call the function on instances of types not mapped from the IDL operation definition.

An IDL operation declared as **`oneway`** shall be mapped on to a generic function that returns zero results.

The IDL interface object shall become the first parameter to the Dylan generic function. A IDL operation parameter declared as **`in`** shall become a parameter of the Dylan generic function. A parameter declared as **`out`** shall become a result of the Dylan generic function. A parameter declared as **`inout`** shall become both a parameter and a result of the Dylan generic function. The Dylan parameters and results shall maintain the order of the original parameters, with the interface object and operation return value coming before any further parameters and results defined by the IDL parameters.

An IDL **`raises`** declaration describes the additional, non-standard, exceptions that may be raised by invocation of the operation. This is not mapped to any visible feature of the generic function that is mapped from the operation declaration. If any exceptions are raised, however, they shall be signalled and not returned from an operation request.

An IDL **`context`** declaration describes which additional pieces of client state the service is passed. When a **`context`** clause is present this shall be mapped to a **`context:`** keyword argument. When invoking an operation, if a context is passed in then this shall be used instead of the ORB's default context for looking up the property names listed in the operation's **`context`** clause. When being invoked, an operation's context keyword argument shall be filled by applying the proper names to the given client context.

A.4.12.3 Rationale

This is the natural mapping. The generic function is open to allow the server module to implement the function by adding methods.

Contexts are mapped are keyword arguments so that client code does not have to worry about them by default.

A.4.12.4 Examples

In Parameters

```
// IDL
interface stealth {
  exception power_failure {};
  void engage_cloak (in long power)
    raises (power_failure);
};

// Dylan
define open abstract class <stealth> (<object>)
end class;

define open generic stealth/engage-cloak
  (stealth :: <stealth>, power :: CORBA/<long>)
  => ();

define class stealth/<power-failure>
  (CORBA/<user-exception>)
end class;

define sealed domain make
  (singleton(stealth/<power-failure>));

define sealed domain initialize (stealth/<power-failure>);
```

Out Parameters

```
// IDL
interface fuel_cell : power_source {
short burn_hydrogen
  (in long burn_rate, out sequence<short,7> emissions);
};

// Dylan
define open abstract class <fuel-cell> (<power-source>)
end class;

define open generic fuel-cell/burn-hydrogen
    (fuel-cell :: <fuel-cell>, burn-rate :: CORBA/<long>)
  => (result :: CORBA/<short>,
      emissions ::limited(CORBA/<sequence>, of: CORBA/<short>));
```

InOut Parameters

```
// IDL
interface frame {
  void request-sizes (inout long width, inout long height);
};

// Dylan
define open abstract class <frame> (<object>)
end class;

define open generic frame/request-sizes
    (frame :: <frame>,
     width :: CORBA/<long>,
     height :: CORBA/<height>)
 => (width :: CORBA/<long>,
     height :: CORBA/<height>);
```

A.4.13 Mapping for attributes

A.4.13.1 Background

IDL attributes implicitly define a pair of accessor functions, one for retrieving the value of the attribute, and another for setting its value. The names of the accessor functions are language-mapping specific, but must not collide with legal operation names specifiable in IDL. Attributes may be defined as **`readonly`**.

Dylan's slots provide a similar mechanism, including **`constant`** slots.

A.4.13.2 Specification

IDL attributes shall be mapped to a pair of *open* generic functions, one *getter* and one *setter*. The names of the functions shall be derived by the usual identifier translation rules, with the addition that the setter function has the suffix **`-setter`**. Attributes declared as **`readonly`** will only have a getter function. It shall be an error to call these functions on instances of types other than those mapped from the IDL attribute definition.

If a mapped attribute name would clash with a mapped operation name then the mapped attribute name shall have **`%`** prepended.

A.4.13.3 Rationale

It is not necessary to specify whether the generic functions are defined by virtual slots.

The identifier translation rules prevent potential name collisions with the setter functions by appending **`-%`** to the end of an existing identifier that ends with **`-setter`**.

In the main, attribute names are unlikely to clash with operation names so it seems harsh to punish the normal case with some additional name mangling. Instead, the penalty is put on defining a clashing operation name for an existing attribute. Existing source code would probably then have to be rewritten in order to continue to get at the old attribute.

A.4.13.4 Examples

```
// IDL
interface prisoners_dilemma {
  attribute short mutual_cooperation_reward;
  attribute short mutual_defection_punishment;
  attribute short defectors_temptation;
  attribute short suckers_payoff;
};

// Dylan
define open abstract class <prisoners-dilemma> (<object>)
end class;

define open generic prisoners-dilemma/mutual-cooperation-reward
  (object :: <prisoners-dilemma>) => (value :: CORBA/<short>);
```

```
define open generic
  prisoners-dilemma/mutual-cooperation-reward-setter
    (value :: CORBA/<short>, object :: <prisoners-dilemma>)
    => (value :: CORBA/<short>);

…
```

A.4.14 Memory management considerations

Dylan is a garbage collected language, however a Dylan program may be interacting, via an ORB, with remote data in another address space, and / or with local data not built using the same Dylan implementation or even the same language. In each case, explicit freeing of some data may be necessary.

By default, the ORB shall provide copying semantics for arguments and results and the Dylan garbage collector shall take care of reclaiming the storage occupied by provably unreferenced data. This includes data of basic types, constructed types, and object references.

However, a Dylan ORB may optionally provide non-copying semantics data. That is, Dylan programs may have to deal with pointers to shared data, because the ORB is acting as an efficient in-process interoperability layer. In this case, it is expected that the memory management protocol will be explicitly described in IDL as part of the contract between the interoperating programs.

A.4.15 Multi-threading considerations

It should be assumed that invocation of a CORBA operation from Dylan is not thread-safe. That is, if two threads in a Dylan program invoke the same operation it is not guaranteed that they will be properly serialized.

An implementation of the IDL-to-Dylan mapping should document whether it is thread-safe , and if it is should document whether the whole program or just the thread is blocked.

A.5 The mapping of pseudo-objects to Dylan

A.5.1 Introduction

A.5.1.1 Background

In addition to defining how application objects and data are accessed, a language mapping must describe how to access some services implemented directly by the ORB. These services are concerned with operations like converting an object reference to a string, making a request through the Dynamic Invocation Interface (DII), building a **`Context`**, and so on.

All that is required is that there be some defined mechanism for doing these things in each language-binding specification. However, most language-binding specifications take the *pseudo-objects* approach in which these ORB services are accessed by applying the binding's normal mapping rules to OMG's IDL descriptions of the interfaces to the services.

The advantage of this is that programmers can read the OMG's descriptions of the interfaces and know how to access them from their preferred language without learning any additional language-specific access methods.

The disadvantage is that some of the interfaces turn out to be particularly clumsy for a given programming language.

A.5.1.2 Specification

The IDL pseudo-object interfaces shall be mapped according to the Dylan language binding as specified in the earlier parts of this document, except where it explicitly deviates. In some cases, also where explicitly specified, there will be an additional access path that is more convenient for Dylan programmers. A conforming Dylan language mapping shall support both interfaces.

A.5.1.3 Rationale

The pseudo object approach is taken so that experienced CORBA developers can leverage more of their knowledge and so that any new pseudo-object services defined by the OMG will automatically be covered.

However, some of the potential pseudo-object IDL descriptions lead to awkward interfaces from the Dylan programmer's point of view and incompatible abstractions might arise on top of them; built by different binding. Therefore, in order to provide more natural Dylan abstractions, there are some additional Dylan-friendly interfaces in the following sections.

A.5.2 ORB Interface

A.5.2.1 Object references

Background

All CORBA interfaces all inherit from the **`CORBA::OBJECT`** interface.

Specification

An IDL interface shall be mapped to a reference class that is a direct or indirect subclass of both the interface class itself and **`corba/<object>`**.

Rationale

This allows object references and servants to inherit from the same protocol class and yet retain their own separate behaviors.

Example

```
// IDL
interface ion {};

// Dylan (protocol)
define open abstract class <ion> (<object>)
end class;

// Dylan (possible reference implementation)
  define class <ion-reference> (<ion>, corba/<object>)
  end class;
```

A.5.2.2 Object reference equality

Background

CORBA allows for (imprecise) equality testing via **`corba::object::is_equivalent`**.

Specification

In addition to mapping the PIDL function above to **`corba/object/is-equivalent`**, a Dylan ORB shall provide a method on **`=`** that performs the same function.

Rationale

It is natural for Dylan programmers to want to use **`=`** on object references.

Example

```
// Dylan
…
if (corba/object/is-equivalent(grid1, grid2))
…
// versus
…
if (grid1 = grid2)
…
```

A.5.2.3 Nil object references

Background

CORBA allows for nil object references. These are returned from some operations to indicate situations where, for example, no real object reference could be found.

Dylan programmers normally indicate such a situation by returning the rogue value **`#f`** and by declaring the type of the return value to be **`false-or(<some-type>)`**.

Specification

NIL object references shall be mapped to interface-specific values that can be tested with **`CORBA::IS_NIL`**. A nil object reference for a particular interface shall be obtainable by calling the function **`make-nil`** on the associated protocol class.

Rationale

Using **`#f`** to represent nil object references would be natural and convenient in some respects, notably during testing. However, all object reference type declarations would have to be wrapped with **`false-or`** which would be very awkward. Even if we did this then we would create ambiguous, unorderable methods which all accepted **`#f`** for some generic functions. A less pervasive approach where **`false-or`** was only used for return values would be better, but then it would be harder to glue operations together. The glue code would have to check for **`#f`** output from one operation before calling the next. This seems an inappropriate place to check this, and should be left to the receiving operation.

There are also advantages in terms of type safety. A spurious **`#f`** is more easily passed to, or returned from, an operation than a more explicit nil reference.

Example

```
// Dylan
let philosopher = make-nil(<philosopher>);
corba/object/is-nil(philosopher); // returns #t
```

A.5.3 Dynamic Invocation Interface

A.5.3.1 NVList

Background

CORBA NVLists are partially opaque in that they can normally only be created by the ORB **`create_list`** operation and added to by the **`add_item`** operation.

Specification

NVLists shall be mapped to CORBA sequences of NamedValues.

Rationale

Since CORBA sequences shall be mapped to a type that supports the **`<stretchy-vector>`** protocol, they can be created and added to using the normal Dylan calls.

This is in addition to the pseudo IDL interfaces **`create_list`** and **`add_item`**.

Example

```
// Dylan
let args = make(corba/<nvlist>);
args := add!(args,
            make(corba/<namedvalue>,
            name: “foo”,
            argument: as(corba/<any>, 0),
            len: 0,
            arg-modes: 0));
```

A.5.4 Dynamic Skeleton Interface

A.5.4.1 Dynamic Implementation Routine

Background

The Dynamic Implementation Routine (DIR) is intended to support a variety of uses, including dynamic programming / scripting languages, bridge building, debugging, monitoring, and so on.

The idea is that a particular kind of registered servant is invoked via the DIR for all operations instead of invoking particular skeletons and thence particular implementation methods.

An auxiliary function is required for the application to inform the POA of the repository-ID of the most derived interface supported by the dynamic servant.

Specification

There shall be a subclass of **`portableserver/<servant>`** called **`portableserver/<dynamic-servant>`**. Instances of subclasses of this class registered with the adapter as servants for objects shall have operations invoked via the DIR function **`corba/serverrequest/invoke`**.

The shall be an open generic function **`portableserver/servant/primary-interface`** which shall be called by the POA to determine the repository-ID of the most derived interface supported by the dynamic servant.

The pseudo IDL of the above is:

```
// IDL
module CORBA {
  interface PortableServer::Dynamic_Servant :
    PortableServer::Servant {};
  interface ServerRequest {
    …
    void invoke (in PortableServer::Dynamic_Servant servant);
};
};

module PortableServer {
  interface Servant {
  …
  RepositoryID primary_interface
    (in ObjectID id, in POA poa)
};
};
```

Rationale

We just need to specify the function that is called on the registered servant for all operations, plus the auxiliary function for determining the repository-ID.

Example

```
// Dylan
define class <buckstop> (portableserver/<dynamic-servant>)
end class;

define method corba/serverrequest/invoke
    (request :: corba/<serverrequest>, servant :: <buckstop>)
    => ()
    …
end method;

define method portableserver/servant/primary-interface
    (servant :: <buckstop>,
    id :: <string>,
    poa :: portableserver/<poa>)
   => (repositoryid :: <string>)
   "LOCAL:buckstop"
end method;
```

A.5.5 The Portable Object Adapter

A.5.5.1 Servants

Background

The `PortableServer` module for the Portable Object Adapter (POA) defines the native `Servant` type.

Specification

The Dylan mapping for the `Servant` type shall be the *open abstract* class **`portableserver/<servant>`**.

An IDL interface shall be mapped to a skeleton class that is a direct or indirect subclass of both the interface class itself and **`portableserver/<servant>`**.

The name of the skeleton class shall be formed by appending **`-servant`** to the interface name and applying the normal identifier mapping rules.

The skeleton class shall be exported from the skeletons library generated from the IDL.

Rationale

Only instances of subclasses of **`portableserver/<servant>`** should be created.

Examples

```
// Dylan Skeleton (generated from IDL)
define class <grid-servant>
  (<grid>, portableserver/<servant>)
end class;

// Dylan Implementation
define class <grid-implementation> (<grid-servant>)
end class;
```

Index