Harlequin Dylan

CORBA User Guide

Version 1.1 Beta

http://www.harlequin.com/

Europe:

Harlequin Limited
Barrington Hall
Barrington
Cambridge CB2 5RG
UK

telephone +44 1223 873 800
fax +44 1223 873 873
support +44 1223 873 901

North America:

Harlequin Incorporated
One Cambridge Center
Cambridge, MA 02142
USA

telephone +1 617 374 2400
fax +1 617 252 6505
support +1 617 374 2435

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
support +61 2 6206 5522

Preface

Product

Harlequin Dylan ORB (HD-ORB) 1.1 is an implementation of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA). HD-ORB provides CORBA functionality to Dylan programmers, combining standardized distributed system development with a state-of-the-art dynamic object-functional language.

Audience

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

This guide is a very early draft of the documentation we intend to ship with the final version of the Harlequin Dylan Enterprise edition.

Standards compliance

HD-ORB 1.1 conforms to the CORBA 2.0 specification with some elements of CORBA 2.2, most notably the Portable Object Adaptor (POA). Work is underway to complete CORBA 2.2 compliance. See:

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

This address may change.

Contents

1

Getting Started with Harlequin Dylan CORBA

1.1 Introduction

This tutorial guides you through the development of a client-server application using Harlequin Dylan CORBA. The example application illustrates how to implement and use CORBA objects in Dylan.

1.2 Overview of the tutorial's example

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

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

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.

The user interface is implemented using the Harlequin Dylan DUIM library.

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

1.2.1 Where to find the code

The code developed in this tutorial can be found in the top-level Harlequin Dylan installation folder, under

`Examples\corba\bank`

There are several subfolders.

- **`Examples\corba\bank\bank`** contains the IDL that defines the CORBA interface to the server (file **`bank.idl`**.)
- **`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 directory also contains the pre-prepared Microsoft Access database file **`bankDB.mdb`** that is used to record account details.

Note: You need to have ODBC 3.x (ODBC 3.0 or later) and an ODBC-driver for Microsoft Access installed on the machine hosting the server in order to run the example. Both ODBC 3.x and the Microsoft Access driver are available as a free download from the Microsoft Universal Data Access web site. A copy of Microsoft Access is not required.

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

At the time of writing, the file to download is:

<URL:http://www.microsoft.com/data/download/mdacfull.exe/>

`mdacfull.exe` for x86 (3.4 Mb, contains ADO / RDS / OLE DB / ODBC, SQL Server Driver, Access Driver, and Oracle Driver) This is the installer for the MDAC 1.5c redistribution that installs, amongst other things, ODBC 3.x and the Microsoft Access driver.

This information may change in the future.

1.2.2 Quickstart: Building and running the demo

1.2.2.1 Registering the database with ODBC

Before we can access the **`bank.mdb`** database using ODBC, we need to register it under a *data source name* in the ODBC control panel.

This installation step also tells ODBC which driver to use when connecting to the database.

NOTE: Process may vary on Windows 95 and 98. These are draft instructions.

- Open the ODBC control panel.
- Select the User DSN page.
- Click **Add...**.
- Select "Microsoft Access Driver" from the list of available drivers.
- Click **Finish**.
- In the "Microsoft Access 97 setup" dialog, enter the "Data Source Name" as **`"bankDB"`**.

- Click **Select** to bring up a file dialog. Browse until you locate the **`bankDB.mdb`** file in the folder **`Examples\corba\bank\bank-server`**.
- Select **`bankDB.mbd`** from the list of available database files.
- Click **OK** to close the file dialog.
- Click **OK** to close the Microsoft Access 97 Setup dialog.
- Click **OK** to close the control panel.

Finally, make sure the **`bank.mdb`** file is writable: right-click on the file in an Explorer window, select Properties from the pop-up menu and clear the "Readonly" attribute if it is checked.

1.2.2.2 Building the client

In the Harlequin Dylan Environment, open project **`bank-client.hdp`** and choose **Project > Build** to compile and link the project.

The environment will automatically invoke the IDL compiler on the file **`bank.idl`** to generate the source code for the protocol and stubs libraries.

1.2.2.3 Building the server

In the Harlequin Dylan Environment, open project **`bank-server.hdp`** and choose **Project > Build** to compile and link the project.

The environment will automatically invoke the IDL compiler on the file **`bank.idl`** to generate the source code for the protocol and stubs libraries.

1.2.2.4 Running the server and client

In the Bank-Server project window, choose **Application > Start** to run the executable for the server. After performing some initialization, the application presents an information dialog to indicate that the server is ready. This dialog has a single **OK** button to shut down the server.

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 single 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 click the **OK** button in the server's dialog to close the connection to the database and exit the server.

1.3 Designing the IDL

The first step in developing a CORBA application it to define the interfaces to its distributed application objects. We can define these interfaces using the CORBA *Interface Definition Language* (IDL). Essentially, the IDL specification of an interface lists the names and types of operations that:

- Any CORBA object, satisfying that interface, must support.
- Any CORBA client, targeting such an object, may request.

Our application manages three types of CORBA object, representing accounts, checking accounts and banks. Each of these interfaces is defined within the same CORBA module , **`BankingDemo`**:

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

  interface checkingAccount : account {
    // details follow
  };

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

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

```
// in module BankingDemo
interface account {
    readonly attribute string name;

    readonly attribute long balance;

    void credit (in unsigned long amount);

    exception refusal {string reason;};
    void debit (in long amount)
      raises (refusal);
};
```

The name of an account is recorded in its **name** attribute. The state of an account is recorded in its **balance** attribute. To keep things simple, we use CORBA **long**s to represents monetary amounts. To prevent clients from directly altering the account's name or balance, these attributes are declared as **readonly** attributes.

The operations **credit** and **debit** must be used to update an account's balance.

The operation **credit** adds a given (non-negative) amount to the current balance.

The phrase **exception refusal {string reason;};** declares a named exception that the debit operation uses to signal errors. The **refusal** exception is declared to contain a **reason** field that documents the reason for failure as a **string**.

The operation **debit** subtracts a given amount from the current balance, provided doing so does not place the account in the red.

Qualifying **debit** by the phrase **raises refusal** declares that invoking this operation may raise the exception **refusal**. Although a CORBA operation may raise any CORBA system exception, its declaration must specify any additional CORBA user exceptions that it might raise.

The bank also manages another sort of account, called a *checking account*. While an ordinary account must maintain a positive balance, a **checkingAccount** may be overdrawn up to an agreed limit. We use IDL's notion of interface inheritance to capture the intuition that a checking account is a special form of account:

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

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

Aside: 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 specialised **debit** method for **checkingAccount**s.

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

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

```
// in module BankingDemo
interface bank {

    readonly attribute string name;

    exception duplicateAccount{};

    account openAccount (in string name)
      raises (duplicateAccount);
    checkingAccount openCheckingAccount(in string name,
                                        in long limit)
      raises (duplicateAccount);

    exception nonExistentAccount{};

    account retrieveAccount(in string name)
      raises (nonExistentAccount);
```

```
        void closeAccount (in account account);
    };
```

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

The operation **openAccount** is declared to take a CORBA **string** and return an account. Because **account** is a 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** may 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, the account's overdraft limit.

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

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

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

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

1.3.1 Compiling the IDL

The document

<URL:http://www.harlequin.co.uk/products/ads/dylan/tech.html>

entitled *An IDL Binding for Dylan* is a draft standard specifying a mapping from CORBA IDL to the Dylan language. According to this specification:

- 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;
- and CORBA exceptions are mapped to Dylan conditions.

The Harlequin Dylan Environment includes an IDL compiler that it uses to check and compile IDL files to Dylan protocol, skeletons and stubs libraries. Each IDL file that a project depends on must have a corresponding Harlequin Dylan Tool Specification file in the project. This file indicates the pathname of the IDL file and which of the generated libraries the project requires. The environment uses the file to invoke the IDL compiler as part of the build process for the project.

The **`bank-client`** and **`bank-server`** projects each contain a specification file for the **`bank.idl`** file. After building both the client and server, the environment will have produced three subdirectories:

Invoking IDL compiler on the file **`bank.idl`** produces three subdirectories:

- **`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.
- **`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.
- **`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.

1.3.2 Mapping IDL to Dylan

Just to give a flavor of the IDL to Dylan mapping, we can take a look at the result of applying the mapping to the file **`bank.idl`**. The Bank-Protocol library produced by the IDL compiler contains the following definitions:

```
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 generic checkingBankingDemo/Account/limit
    (object :: BankingDemo/<checkingAccount>)
  => (result :: CORBA/<long>);

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

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

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

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

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

```
define open generic BankingDemo/bank/openCheckingAccount
    (object :: BankingDemo/<bank>, name :: CORBA/<string>,
     limit :: CORBA/<long>)
  => (result :: BankingDemo/<checkingAccount>);

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

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

define open generic BankingDemo/bank/closeAccount
    (object :: BankingDemo/<bank>,
     account :: BankingDemo/<account>)
  => ();
```

To provide some more intuition for the mapping scheme, let's examine the Dylan counterparts of some representative IDL declarations in the file **`bank.idl`**.

1.3.2.1 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>`**.

1.3.2.2 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 withing its scope, resulting in 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>`**).

1.3.2.3 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 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 (**/**).

1.3.2.4 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 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.

1.3.2.5 Mapping for exceptions

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

1.4 The client

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

1.4.1 The client's perspective

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

The Bank-Protocol library, produced by the IDL compiler, 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. This library should be used by any application that wants to act as a client with respect to some CORBA object matching an interface in the **`bank.idl`** file.

The Bank-Stubs library defines concrete classes **`BankingDemo/<account-reference>`**, **`BankingDemo/<checkingAccount-reference>`** and **`BankingDemo/<bank-reference>`**, that subclass the abstract classes **`BankingDemo/<account>`**, **`BankingDemo/<checkingAccount>`** and **`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 on the server.

The Bank-Stubs library also defines a concrete *stub* method, specialised 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 by handled by the client code using standard Dylan constructs.

1.4.2 Implementing the bank client

To keep things simple, we organise the structure of the user interface to closely match the IDL description of the bank. Each CORBA object is presented in its own frame (or window). We define one subclass of **`<simple-`**

frame> for each CORBA interface. The definition of these subclasses is derived from the declaration of their corresponding CORBA interfaces. In particular, we use text fields to represent IDL attributes, and buttons to invoke IDL operations. Each frame contains a slot that contains the CORBA object it represents. Clicking on a button of the frame triggers a callback that invokes the corresponding operation on the CORBA object associated with that frame. The user is notified of any CORBA user-exceptions that these operations raise. The CORBA specific code resides, to a large extent, in these callbacks.

The bank client is implemented as a library:

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

that defines a single module:

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

Note that 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 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 source code for the client is in file **bank-client.dylan**.

1.4.2.1 Defining the Frames

In this section, we define three DUIM frame classes **<account-frame>**, **<checkingAccount-frame>** and **<bank-frame>**. These classes are used to present graphical interfaces to CORBA objects with the IDL interfaces **account**, **checkingAccount** and **bank**.

This section assumes some basic familiarity with the DUIM library.

We begin by defining the frame class **<account-frame>**:

```
define frame <account-frame> (<simple-frame>)
  constant slot account :: BankingDemo/<account>,
            required-init-keyword: account:;
  constant slot bank-frame :: <bank-frame>,
            required-init-keyword: bank-frame:;

// definition of fields
  pane balance-field (frame)
    make(<text-field>,
         label: "Balance",
         read-only?: #t);

// definition of buttons
  pane credit-button (frame)
    make(<push-button>,
         label: "Credit ...",
         max-width: $fill,
                    activate-callback: credit-callback);

pane debit-button (frame)
       make(<push-button>,
            label: "Debit ...",
            max-width: $fill,
            activate-callback: debit-callback);

// definition of tool bar
  pane account-tool-bar (frame)
    make(<tool-bar>,
         child: horizontally ()
                  frame.credit-button;
                  frame.debit-button;
                end);

// definition of layout
  pane account-layout (frame)
         labelling ("Balance:") frame.balance-field end;

// activation of frame elements
  tool-bar (frame) frame.account-tool-bar;
  layout (frame) frame.account-layout;
  status-bar (frame) make(<status-bar>,
                          label: "This is an account.");
end frame <account-frame>;
```

This is how we use an instance of class **<account-frame>**. We store the name of the customer owning this account in the title of the frame (using the methods **frame-title** and **frame-title-setter** that are inherited from the superclass **<simple-frame>**).

The **account** slot stores a CORBA object reference, of class **BankingDemo/<account>**, to the corresponding CORBA **account** object on the server.

The **bank-frame** slot stores the frame of the bank that manages this account. We will define the class **<bank-frame>** in a moment.

The pane **balance-field** reports the state of the CORBA object's **balance** attribute as a read-only text field. We delegate the initialization of this field value to an **initialize** method specialised on **<account-frame>**. The value needs to be updated after each invocation of a CORBA **debit** or **credit** operation.

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

The buttons are layed out in a tool bar **account-tool-bar**.

The pane **account-layout** simply attaches a descriptive label to the balance field.

Finally, the frame options **tool-bar** and **layout** bind the **<account-frame>**'s tool bar and the top-level layout to **account-tool-bar** and **account-layout**.

Mirroring the fact that the IDL interface **checkingAccount** inherits from **account**, we define the Dylan frame class **BankingDemo/<checkingAccount>** as a subclass of **BankingDemo/<account>**:

```
define frame <checkingAccount-frame> (<account-frame>)
 pane limit-field (frame)
        make(<text-field>, read-only?: #t);

 pane checkingAccount-layout (frame)
        vertically ()
          frame.account-layout;
          labelling ("Limit:") frame.limit-field end;
        end;

// activation of frame elements
  layout (frame) frame.checkingAccount-layout;
  status-bar (frame) make(<status-bar>, label: "This is a
checking account.");
end frame <checkingAccount-frame>;
```

The pane **limit-field** reports the state of the CORBA object's **limit** attribute as a read-only text field. Again, we can delegate the initialization of this field's value to an **initialize** method specialised on **<checkingAccount-frame>**.

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

The **layout** option ensures that the frame's top-level layout evaluates to **checkingAccount-layout**.

The definition of **<bank-frame>** class follows the same pattern:

```
define frame <bank-frame> (<simple-frame>)
  constant slot bank :: BankingDemo/<bank>, init-keyword: bank:;
  slot account-frames :: limited(<list>,
                                 of: <account-frame>) = list();

// definition of buttons

  pane openAccount-button (frame)
    make(<push-button>,
         label: "Open Account ...",
         max-width: $fill,
         documentation: "Open a new account.",
         activate-callback: openAccount-callback);
```

```
  pane openCheckingAccount-button (frame)
    make(<push-button>,
         label: "Open Checking Account ...",
         max-width: $fill,
         documentation: "Open a new checking account.",
         activate-callback: openCheckingAccount-callback);

  pane retrieveAccount-button (frame)
    make(<push-button>,
         label: "Retrieve Account ...",
         max-width: $fill,
         documentation:
           "Retrieve an existing account or checking account.",
         activate-callback: retrieveAccount-callback);

  pane closeAccount-button (frame)
    make(<push-button>,
         label: "Close Account ...",
         max-width: $fill,
         documentation:
           "Close an account, deleting it from the bank.",
         activate-callback: closeAccount-callback);

// definition of tool bar
  pane bank-tool-bar (frame)
    make(<tool-bar>,
         child: vertically ()
                  frame.openAccount-button;
                  frame.openCheckingAccount-button;
                  frame.retrieveAccount-button;
                  frame.closeAccount-button;
                end);

// activation of frame elements

  tool-bar (frame) frame.bank-tool-bar;
    status-bar (frame) make(<status-bar>,
                            label: "This is a bank.");

// frame title
  keyword title: =  "Bank";
end frame <bank-frame>;
```

The **bank** slot stores a CORBA object reference, of class **BankingDemo/<bank>**, to the corresponding CORBA **bank** object on the server.

The **account-frames** slot keeps track of the **<account-frame>**s created by the bank frame as the result of invoking operations on the CORBA **bank** object. This list is maintained to prevent the user from obtaining more than one interface to the same account. We need to update it whenever an account frame is exited, whether implicitly, as a side-effect of the method **closeAccount-callback**, or explicitly, by the user clicking on the close button in the top-right corner of the frame.

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

1.4.2.2 Initializing and exiting account frames

Each time we make a new account frame we want to ensure two things: that the account frame is registered in the bank frame that spawned it, and that its balance pane displays the correct value. An easy way to do this is to add an **initialize** method specialised on **<account-frame>**. (In Dylan, each call to **make** an instance of a given class is automatically followed by a call to **initialize** that instance; you are free to specialise the **initialize** generic function on particular classes.)

```
define method initialize (account-frame :: <account-frame>, #key)
  next-method();
  let bank-frame = account-frame.bank-frame;
    bank-frame.account-frames := add(bank-frame.account-frames,
                                     account-frame);
  let account = account-frame.account;
  gadget-value(account-frame.balance-field) :=
    integer-to-string(account.BankingDemo/account/balance);
end method initialize;
```

Here, we encounter our first example of invoking a CORBA operation on a CORBA object. The dylan variable **account**, of class **BankingDemo/<account>**, contains a proxy for a CORBA **account** object on the server. The application **(account.BankingDemo/account/balance)** invokes a stub method special-

ised on the proxy's class. The stub method forwards the request across the ORB to the actual object on the server. The request is executed on the object in the server and the result passed back across the ORB to the stub method, which returns the value to the client as a **`CORBA/<long>`**. This value is then used to set the initial value of the balance field.

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

```
define method initialize
    (checkingAccount-frame :: <checkingAccount-frame>, #key)
  next-method();
  let account = checkingAccount-frame.account;
  gadget-value(checkingAccount-frame.limit-field) :=
    integer-to-string(account.checkingBankingDemo/Account/limit);
end method initialize;
```

The call to **`next-method`** registers the frame and set up its balance field; a call to the **`BankingDemo/checkingAccount/limit`** stub determines the initial value of its limit field.

For convenience, we define a generic function **`make-account-frame`** that **`make`**s the right class of frame for a given account object reference:

```
define method make-account-frame
    (account :: BankingDemo/<account>,
     #key bank-frame :: <bank-frame>, title :: <string>)
    => (account-frame :: <account-frame>)
  make(<account-frame>,
       account: account,
       bank-frame: bank-frame,
       title: title,
       owner: bank-frame);
end method make-account-frame;

define method make-account-frame
    (checkingAccount :: BankingDemo/<checkingAccount>,
    #key bank-frame :: <bank-frame>, title :: <string>)
    => (checkingAccount-frame :: <checkingAccount-frame>)
  make(<checkingAccount-frame>,
       account: checkingAccount,
       bank-frame: bank-frame,
       title: title,
       owner: bank-frame);
end method make-account-frame;
```

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

Before we exit an account frame, we should ensure that it is removed from the list of account frames managed by its bank frame. Fortunately, events of class **<frame-exit-event>** are placed on a frame's event queue just before the frame is exited and unmapped from the display. So a simple way to update the list is to add a method to the DUIM generic function **handle-event**, specialised on the classes **<account-frame>** and **<frame-exit-event>**:

```
define method handle-event
    (account-frame :: <account-frame>,
     event :: <frame-exit-event>)
    => ()
  let bank-frame = account-frame.bank-frame;
    bank-frame.account-frames :=
      remove(bank-frame.account-frames, account-frame);
  next-method();
end method handle-event;
```

1.4.2.3 Defining the callbacks

Defining the callbacks attached to each button gadget 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** may be used to return the enclosing frame of a gadget.

The **credit-callback** is activated by the credit button of some account frame:

```
define method credit-callback (credit-button :: <push-button>)
    => ()
  let account-frame = credit-button.sheet-frame;
  let account = account-frame.account;
  let amount = prompt-for-amount(owner: account-frame,
                                 title: "Credit ...");
  if (amount)
    BankingDemo/account/credit(account, abs(amount));
    gadget-value(account-frame.balance-field) :=
      integer-to-string(account.BankingDemo/account/balance);
  end if;
end method credit-callback;
```

The callback first retrieves the **account-frame**. It then extracts the CORBA object reference stored in the frame's **account** slot and prompts the user for an amount. The function **prompt-for-amount** queries the user for an integer and returns **#f** if the user cancels the dialog or enters an invalid string. If the amount is valid, the callback invokes the stub method **BankingDemo/account/credit** on the CORBA object reference with the specified *absolute* value of the amount (recall that the **credit** operation expects an *unsigned* **long** as its argument). Finally, it updates the balance field of the frame with the current value of the object's **balance** attribute, obtained by invoking the stub method **BankingDemo/account/balance**.

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

```
define method debit-callback (debit-button :: <push-button>)
    => ()
  let account-frame = debit-button.sheet-frame;
  let account = account-frame.account;
  let amount = prompt-for-amount(owner: account-frame,
                                 title: "Debit ...");
  if (amount)
    block ()
      BankingDemo/account/debit(account, amount);
      gadget-value(account-frame.balance-field) :=
        integer-to-string(account.BankingDemo/account/balance);
      exception(refusal :: BankingDemo/account/<refusal>)
        notify-user(
          concatenate(
            "Debit refused for the following reason: ",
            refusal.BankingDemo/account/refusal/reason),
          owner: account-frame);
    end block;
  end if;
end method debit-callback;
```

The only difference is that **debit-callback** must deal with the additional possibility that the **debit** operation, when invoked on the target object, may fail, raising the IDL exception **refusal**.

If the object raises this exception, the **BankingDemo/account/debit** stub method signals it as a Dylan condition of class **BankingDemo/account/<refusal>**. The exception can then be caught and han-

dled in any of the standard Dylan ways. Here, we simply place the invocation in the body of a **block** statement with an appropriate **exception** clause to handle the condition:

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

```
define method openAccount-callback
    (openAccount-button :: <push-button>)
    => ()
  let bank-frame = openAccount-button.sheet-frame;
  let bank = bank-frame.bank;
  let name = prompt-for-name(owner: bank-frame,
                             title: "Open Account ...");
  if (name)
    block ()
      let account = BankingDemo/bank/openAccount(bank, name);
      let account-frame =
        make-account-frame(account,
                           bank-frame: bank-frame,
                           title: name);
      start-frame(account-frame);
      exception(
        duplicateAccount :: BankingDemo/bank/<duplicateAccount>)
          notify-user(
            concatenate("Cannot create another account for ",
                        name, "!"),
            owner: bank-frame);
    end block;
  end if;
end method openAccount-callback;
```

The callback retrieves the frame and extracts the CORBA object reference stored in the frame's **bank** slot. The function **prompt-for-name** queries the user for the new customer's **name** returning a string (or **#f** if the user cancels the dialog). If the dialog has not been cancelled, the callback invokes the stub method **BankingDemo/bank/openAccount** on the target object reference **bank**, passing the argument **name**.

If successful, the invocation returns an object reference, of class **BankingDemo/<account>**, to an IDL **account** object, which is then used to make and start a new **<account-frame>**, via a call to **make-account-frame**.

Recall that the IDL operation **`openAccount`** may fail, raising the IDL user exception **`duplicateAccount`**. As in the definition of **`debit-callback`**, we cater for this eventuality by placing the invocation in the body of a **`block`** statement and installing a handler on the corresponding Dylan condition of class **`BankingDemo/bank/<duplicateAccount>`**. This handler simply informs the user of the exception using the DUIM function **`notify-user`** to create and display a simple alert dialog box.

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

```
define method openCheckingAccount-callback
    (openCheckingAccount-button :: <push-button>) => ()
  let bank-frame = openCheckingAccount-button.sheet-frame;
  let bank = bank-frame.bank;
  let name = prompt-for-name(owner: bank-frame,
                             title: "Open Checking Account ...");
  if (name)
    let limit = prompt-for-amount(owner: bank-frame,
                                  title: "Limit ...");
    if (limit)
      block ()
        let checkingAccount =
          BankingDemo/bank/openCheckingAccount(bank,
                                               name,
                                               limit);
        let checkingAccount-frame =
          make-account-frame(checkingAccount,
                             bank-frame: bank-frame,
                             title: name);
        start-frame(checkingAccount-frame);
        exception
            (duplicateAccount ::
             BankingDemo/bank/<duplicateAccount>)
          notify-user(
            concatenate("Cannot create another account for ",
                        name,
                        "!"),
            owner: bank-frame);
      end block;
    end if;
  end if;
end method openCheckingAccount-callback;
```

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

```
define method retrieveAccount-callback
    (retrieveAccount-button :: <push-button>)
    => ()
  let bank-frame = retrieveAccount-button.sheet-frame;
  let bank = bank-frame.bank;
  let name = prompt-for-name(owner: bank-frame,
                             title: "Retrieve Account ...");
  if (name)
    begin
      if (any? (method (account-frame :: <account-frame>)
                  account-frame.frame-title = name;
                end method,
                bank-frame.account-frames))
        notify-user("Account already open!",
                    owner: bank-frame);
      else
        block ()
          let account = BankingDemo/bank/retrieveAccount(bank,
                                                         name);
          let account-frame =
            make-account-frame(account,
                               bank-frame: bank-frame,
                               title: name);
          start-frame(account-frame);
          exception
             (nonExistentAccount ::
              BankingDemo/bank/<nonExistentAccount>)
            notify-user(
              concatenate("No existing account for ", name, "!"),
              owner: bank-frame);
        end block;
      end if;
    end;
  end if;
end method retrieveAccount-callback;
```

This callback incorporates a test that prevents the user from being presented with more than one frame to the same account. It invokes the stub method **BankingDemo/bank/retrieveAccount** only if the account under that name is not already on display. Because of IDL inheritance, the server implementing the IDL **retrieveAccount** operation may return any object reference whose

interface inherits from the IDL **account** interface. In particular, the server may return an IDL **checkingAccount** as a special instance of an IDL **account**. In Dylan terms, this mean that the stub method **BankingDemo/bank/retrieveAccount** may return an object reference of class **BankingDemo/<checkingAccount>** as a special instance of **BankingDemo/<account>**. The call to **make-account-frame** dispatches on the actual, or most derived, class of the resulting object reference, making an **<account-frame>** or **<current account-frame>** as appropriate.

The definition of the **closeAccount-callback** is straightforward:

```
define method closeAccount-callback
    (closeAccount-button :: <push-button>)
    => ()
  let bank-frame = closeAccount-button.sheet-frame;
  let account-frames = bank-frame.account-frames;
  let account-frame =
    prompt-for-account-frame(account-frames, title: "Close ...",
                             owner: bank-frame);
    if (account-frame)
      let bank = bank-frame.bank;
      let account = account-frame.account;
      BankingDemo/bank/closeAccount(bank, account);
      exit-frame(account-frame);
    end if;
end method closeAccount-callback;
```

The function **prompt-for-account-frame** presents a dialog asking the user to select a frame from the list of available account frames (indexed by their titles), returning **#f** if the user decides to cancel the dialog. Given a valid selection, the callback invokes the stub method **BankingDemo/bank/closeAccount** on the target object reference, **bank**, passing the **account** field of the selected frame as the argument. The final call to **exit-frame** places a **<frame-exit-event>** on the account frame's event loop and unmaps the frame, removing it from display.

The implementations of the utility functions **prompt-for-amount**, **prompt-for-name** and **prompt-for-account-frame** are straightforward and are not described in this document. You can consult the source in the file

```
Examples\corba\bank\bank-client\bank-client.dylan
```

1.4.2.4 Initializing the ORB and obtaining the first object reference

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

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

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

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

Operations providing access to the ORB reside in the CORBA module (like an IDL interface declaration, a (P)IDL module declaration defines a new namespace for the body of declarations it encloses. What it does not do is define a new type of CORBA object.). Operations providing access to Object Adapters, Interface Repository, Naming Service, and other Object Services reside in the ORB interface defined within the CORBA module. To provide some flavour 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, we use a very simple technique to obtain the initial object reference. The client assumes that the server has published a reference to its implementation of the bank object, encoded as a string, in a shared file. After starting up, the client reads the file, decodes the string as an object reference, and then uses this reference as the target of further operations.

Here is the remaining Dylan code that completes the implementation of the client.

```
define constant $bank-ior-file = "c:\\temp\\bank-demo.ior";

define method file-as-string (ior-file :: <string>)
  with-open-file(stream = ior-file, direction: #"input")
    as(<string>, read-to-end(stream));
  end;
end method;
```

```
begin
  let orb = CORBA/ORB-init(make(CORBA/<arg-list>),
                           "Harlequin Dylan ORB");
  let bank =
    as(BankingDemo/<bank>,
      CORBA/ORB/string-to-object(orb,
                                 file-as-string($bank-ior-file)));
  let bank-frame = make(<bank-frame>, bank: bank);
  start-frame(bank-frame);
end;
```

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.

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

The top-level **begin** ... **end** statement first initializes the Harlequin Dylan ORB by calling the Dylan generic function **CORBA/ORB-init** corresponding to the PIDL **ORB_init** operation (notice 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>**.

Invoking **CORBA/ORB/string-to-object** on this ORB, passing the string read from the shared file, reconstitutes the string as an unspecific object reference of class **CORBA/<Object>**. Calling the **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 the user.

The implementation of the client is now complete.

1.5 The server

In this section, we use Dylan to implement a CORBA server using 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. This database is manipulated by the server using the Harlequin Dylan 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.

1.5.1 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 1.1 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.

1.5.2 A Bluffer's Guide to 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.

It 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 class **<dbms>** is used to identify a database management system. The SQL-ODBC library defines an instantiable class **<odbc-dbms>** for identifying the ODBC DBMS.
- The **<user>** class identifies users to a DBMS. Exactly what a user means depends on the DBMS. The **make** method on **<user>** takes two keyword arguments: 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 **<database>** class 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 **<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 **<connection>** class 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 when a connection is made to an ODBC database.
- The **<sql-statement>** class represents SQL statements and their state. This class has the following init-keywords. The required keyword **text:** expects a **<string>** that contains the text of the 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 (dbms)
  body
end with-dbms;
```

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

```
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 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;
```

1.5.3 Implementing CORBA objects on the server

The server has to provide an implementation object, called a servant, for each of the proxy objects manipulated by the client. In particular, it needs to implement the initial **`bank`** servant, and then create new servants for each of the account objects created in response to **`openAccount`**, **`openCheckingAccount`** and **`retrieveAccount`** requests. Each of these servants needs to be registered in the CORBA environment and assigned an object reference, so that the ORB can direct incoming requests to the appropriate servant.

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

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

The Harlequin 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 implementation are registered with a POA and assigned object references. When a client issues a request to perform an operations 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 (...);
    …
  };
```

```
    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, the **PortableServer** module. It declares several interfaces, of which only the **POA** and **POAManager** are shown here.

The **PortableServer** module specifies the **Servant** type. Values of type **Servant** represent programming 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 cause requests for a POA to be queued or discarded, and can cause the POA to be deactivated.

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

When a POA is in the *holding* state, it simply queues request 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.

The **servant_to_reference** operation has two behaviours. 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.

1.5.4 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.

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

The Bank-Skeletons library defines a concrete skeleton method, specialised 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 client.

1.5.5 Implementing the bank server

The server is implemented as a library:

```
define library bank-server
  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 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 make non-essential use of the DUIM and Threads libraries and modules to present the user with a dialog to shutdown the server. The source code for the server is in file **`bank-server.dylan`**.

1.5.5.1 Implementing the servant classes

The Bank-Skeletons library defines three abstract servant classes **`BankingDemo<account-servant>`**, **`BankingDemo/<checkingAccount-servant>`** and **`BankingDemo/<bank-servant>`** corresponding to the IDL inter-

faces **account**, **checkingAccount** and **bank**. The class **BankingDemo/<checkingAccount-servant>** is defined to inherit from **BankingDemo<account-servant>**, matching the inheritance relationship in the IDL.

Note that each class inherits from the abstract class **PortableServer/<Servant>**, allowing instances of the class to be registered with a POA.

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

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>;
```

We have included the **poa** slot to record the POA in which the bank servant is active, so that servants representing accounts at the bank can be registered in the same POA.

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>;
```

1.5.5.2 Implementing the servant methods

The next step in implementing the server is to define methods, specialized on the implementation classes, for each of the protocol functions corresponding to an IDL attribute or operation.

To support this, the abstract servant classes **BankingDemo/<account-servant>**, **BankingDemo/<checkingAccount-servant>** and **BankingDemo/<bank-servant>** are defined to inherit, respectively, from the abstract protocol classes **BankingDemo/<account>**, **BankingDemo/<checkingAccount>** and **BankingDemo/<bank-servant>**.

As a result, implementing a protocol function boils down to defining a concrete method for that function that specializes on the implementation class of its target object. Recall that the target object of a protocol function is the first parameter to that function.

We can now present the implementations of the protocol functions:

The **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 specialised **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 the raising of CORBA user exceptions:

```
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 **existsAccount?(bank, name)** test 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);
      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, using a call to the **`Portableserver/POA/reference-to-servant`** operation of the POA.

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

1.5.5.3 Initializing the ORB, obtaining the initial POA object and registering the first object reference

To complete the implementation of the server we need to enter 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 constant $dbms = make(<odbc-dbms>);

define constant $datasource-name :: <string> = "bankDB";

define constant $user-name :: <string> = "";

define constant $user-password :: <string>  = "";

define constant $bank-ior-file = "c:\\temp\\bank-demo.ior";

define method string-as-file
    (ior-file :: <string>, ior-string :: <string>)
  with-open-file(stream = ior-file, direction: #"output")
    write(stream, ior-string);
  end;
end method;

begin
  // 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 bank servant
    let bank = make(<bank-implementation>,
                    connection: connection,
                    poa: RootPOA);

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

```
    // create an ior string to pass to clients via the shared file
    string-as-file($bank-ior-file,
                   CORBA/ORB/object-to-string(orb,
                                              bank-reference));

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

    // create a separate thread to shut down the orb, unblocking
    // the main thread.
    make(<thread>,
         function:
           method ()
             notify-user("Click OK to shut it down.",
                         title: "The bank server is active.");
             CORBA/ORB/shutdown(orb, #t);
           end method);

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

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

The first four constants are used in establishing a connection to the database.

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. (The method **`string-as-file`** writes a string to a file.)

The top-level **`begin`** … **`end`** statement 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 intial 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, in the shared file.

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.

This completes the description of our implementation of the server.

2

Creating CORBA Projects

2.1 About CORBA projects

A CORBA project is a project which requires the use of protocol, stubs or skeletons code generated from IDL. The project can include information about the IDL source so that the Harlequin Dylan Environment can automatically invoke its IDL compiler to generate the required Dylan code.

Information about IDL source files and IDL compiler switches are described in a Harlequin Dylan Tool Specification file. These files are not only used to describe IDL to a project. For example, they may be used to describe COM Type Libraries to a project. For CORBA projects, specification files can be generated by hand and inserted into projects using **Project > Insert File...** in the project window. More simply, the New Project wizard can be used to generate a client and / or server project for a given IDL file, including then necessary specification files.

2.2 Creating CORBA projects with the New Project Wizard

1. Choose **File > New...** from the main window.
2. Select Project and click **OK**.

   The New Project wizard appears.

3. In the Project Type section, select “Interface to CORBA IDL file” and click **Next**.

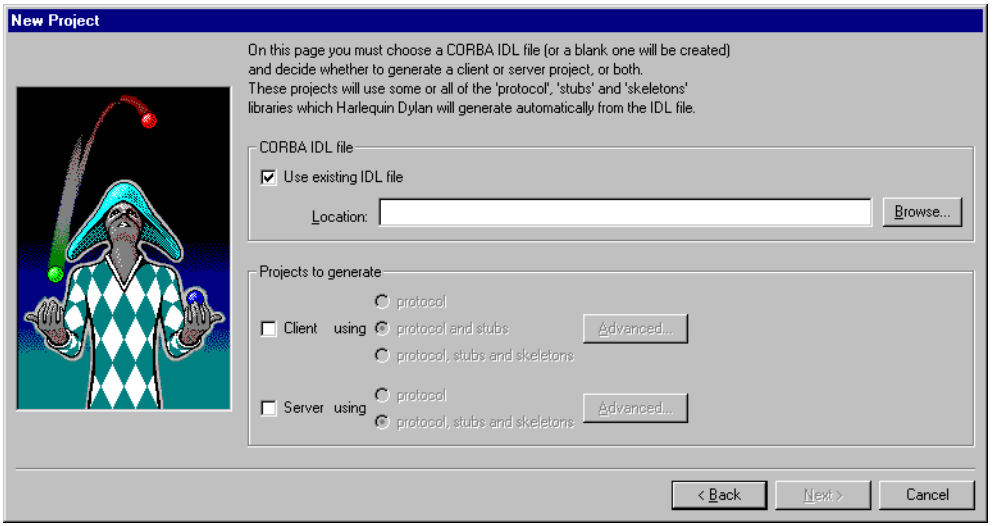

Figure 2.1 Selecting an IDL file.

4. Choose a CORBA IDL file on which to base the project.

The file will be copied into the project folder which you choose on the next page (or left there if it is already in that folder). Alternatively, you can have the wizard generate a blank IDL file for you.

5. Choose whether to generate a client or server project, or both.

These will be created in folders called **`client`** and **`server`**, within the project folder.

You can choose which of the libraries generated from the IDL file will be used by the client and server projects. You can also change some advanced settings, using the **Advanced...** button to bring up the Advanced ORB Settings dialog.

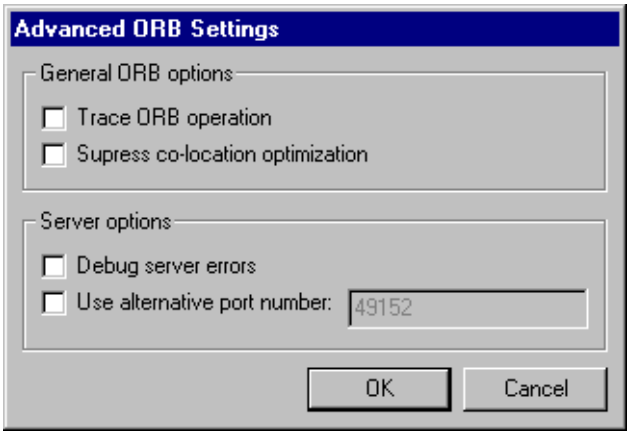

Figure 2.2 Advanced ORB settings.

The options in the Advanced ORB Settings dialog are translated into command line arguments on the **Project > Settings...** dialog's Debug page as follows.

Option	ORB command line argument
Trace ORB operation	-ORBtrace
Suppress co-location optimization	-ORBno-co-location
Debug server errors	-ORBdebug
Use alternative port number	-ORBport *number*

Table 2.1 Advanced ORB settings and ORB command line arguments

6. Click **Next** and continue through the wizard as normal.

Note that project settings and library choices will apply to both generated projects. If you want the two projects to have different settings or use different libraries, you must create one and then the other.

As with COM projects, the interface code will not be generated until you build your client or server project.

2.3 Specification files for CORBA

The first line of a specification file must contain the **Origin:** keyword. In the context of describing IDL source files to CORBA projects, the value of this keyword must be **OMG-IDL**. There is only one other required keyword in this context and that is **IDL-file:**. The value of this keyword must be a pathname to the IDL source file. The pathname is interpreted relative to the directory in which the specification file is located. Each specification file may include only one IDL source file. However, projects may have more than one specification file.

Thus, an example of the minimal legal specification file for IDL is:

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

By default the IDL compiler will generate three projects, one for each library that must be generated according to the Dylan IDL binding. All three of these projects will become subprojects of the CORBA project.

The projects will be placed in subdirectories named protocol, stubs and skeletons in the same directory that the IDL source file is located. There are specification file options which allow some control over where the subdirectories are located and what they should be called.

The next sections describe keyword options that are specific to writing a CORBA client or a CORBA server project.

2.3.1 CORBA server keywords

To indicate that the CORBA project contains a server implementation for the IDL source file use the keyword **Skeletons:** with the value **yes**. This will ensure the IDL compiler generates the skeletons library from the IDL source. The skeletons project will automatically be added as subproject of the CORBA project.

```
Skeletons: yes
```

2.3.2 CORBA client keywords

To indicate that the CORBA project is a client of the service described by the IDL source use the keyword **Stubs:** with the value **yes**. This will ensure the IDL compiler generates the stubs library from the IDL source. The stubs project will automatically be added as a subproject of the CORBA project.

```
Stubs: yes
```

Sometimes the CORBA client may need to use the skeletons generated from the IDL source; for example, where the IDL defines a callback interface which the client must implement. In this case the **Skeletons:** keyword should also be included in the specification file with the value **yes**.

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

2.3.3 Using only the protocol library

The protocol library is available separately for projects 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.

A CORBA project may indicate it is interested only in the protocol library by using the **Protocol:** keyword with the value **yes**. The IDL compiler will ensure the protocol library is generated, but will not generate the stubs or skeletons libraries unless the **Stubs:** or **Skeletons:** keywords are also present and have the value **yes**. The protocol library will automatically be added as a subproject of the CORBA project.

3

Compiling CORBA Projects

3.1 What happens when you compile a CORBA project

In a CORBA project, the IDL compiler will be invoked automatically by the Harlequin Dylan IDE or the console compiler when the CORBA project is built. The IDL compiler will not regenerate the stubs, skeletons or protocol libraries if they are up-to-date with respect to the IDL source file. However, it will ensure that the libraries required by the CORBA project have been generated.

If a specification 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 becuase some options in the specification file may cause the generated code to change, for example the **`IDprefix:`** keyword.

Specification files can include the **`clean:`** keyword which can take the values **`yes`** or **`no`**. A value of **`yes`** will force the IDL compiler to regenerate code from the IDL every time the CORBA project is built. This means that the generated libraries and the CORBA project will be recompiled by the Dylan compiler. The default value for this keyword is **`no`**.

3.2 IDL compiler switches

IDL compiler switches are controlled by keywords in the Harlequin Dylan Tool Specification files. As mentioned earlier, for CORBA IDL source files the first line of the specification file should contain the **`Origin:`** keyword with the value OMG-IDL.

```
Origin:  OMG-IDL
```

There is only one other required keyword. This is the **`IDL-file:`** keyword which indicates where the location of the IDL source file. This should be a pathname relative to the directory in which the specification file is located. Only one IDL source file may be included in each specification file.

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

The rest of the keywords are optional:

`directory:` *directory*

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

`prefix:` *directory-prefix*

The default names for the subdirectories are **`protocol`**, **`stubs`** and **`skeletons`**. You may specify a prefix for these names with this option. The prefix will be appended with a "-" character. This might be useful, for example, where you want to put the project subdirectories from more than one IDL file into a common directory.

`idprefix:` *prefix-string*

Unfortunately, the IDL compiler does not yet support **`#pragma`** directives. This option is a substitute for the prefix pragma directive. 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 values of the following keywords all default to **no**:

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 identifer or keyword definition in order to be recognised.

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.

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.

stubs: **yes | no**

If **yes** this option will ensure the IDL compiler generates the protocol and stubs libraries from the IDL source. The stubs project will be added as a subproject of the CORBA project. This option should be used by projects that wish to be a client of the service described by the IDL source.

skeletons: **yes | no**

If **yes** this option will ensure the IDL compiler generates the protocol, stubs and skeletons library from the IDL source. The skeletons project will automatically be

added as a subproject of the CORBA project. This option should be used by projects that contain a server implementation for the IDL source.

4

Running and Debugging CORBA Applications

4.1 Debugging client/server applications in the IDE

The following text is a quotation from the Harlequin Dylan 1.1 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 source pages such as 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 hit. 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.

4.2 ORB threading model

The text for this section has not been written yet.

4.3 ORB runtime switches

The full set of command line arguments that the Dylan ORB currently supports are 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 for harlequinorb is 3672.

-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.

5

Using the Dylan IDL Compiler

5.1 Introduction

The Harlequin Dylan CORBA IDL file compiler is a standalone executable called **`console-scepter.exe`**. We call the application Scepter. It can be found in the **`Bin`** folder of the top-level Harlequin Dylan installation folder.

Scepter reads an IDL file and generates appropriate stub and skeleton code in three projects. Each project defines one of the libraries specified in Chapter 4.2 of the document *An IDL Binding for Dylan*. 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`**.

5.2 General Usage

Scepter may be invoked from an MS-DOS prompt. The general form of the command is:

```
C:\> scepter [options] foo.idl
```

which will compile the IDL file **`foo.idl`**. Only one filename may be supplied. Options can be prefixed with the character "**`/`**" or "**`-`**". Options which take a value must be separated from the value by a "**`:`**" character or whitespace.

As an example, to compile the bank demo IDL file:

```
C:> scepter /language:dylan bank.idl
```

5.3 Code generation options

/language:dylan

The **/language** option specifies the language in which to generate the stub and skeleton code. Currently only the value **dylan** is supported. If this option is omitted Scepter will parse the IDL file but will not generate any code.

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

/prefix:*name* The default names for the subdirectories 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 subdirectories from more than one IDL file into a common folder.

/stubs By default Scepter generates three libraries for each IDL file; protocol, stubs and skeletons. 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 Scepter to generate only the protocol and stubs libraries.

5.4 Preprocessor options

/preprocess Runs the preprocessor on the IDL file and displays the result on standard output. The preprocesser 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.

5.5 Misc options

/help Display the command-line usage message.

/version Display version information.

/debugger Enter a debugger if Scepter crashes.

/case By default Scepter ignores case when recognising 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.

5.6 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.

```
scepter /language:dylan /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`**.

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

6

An IDL Binding for Dylan

Version 1.0 (Draft OMG Request For Comment submission)

6.1 Introduction

6.1.1 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 labelled.

Dylan names within the text and all example Dylan code appears in **`fixed point font`**.

6.1.2 References

[DP 97]	N. Feinberg, S. E. Keene, R.O. Mathews, P. T. Withington, *Dylan Programming*, Addison Wesley, 1997.

Table 6.1

[DRM 96]	L. M. Shalit, *The Dylan Reference Manual*, ISBN 0-201-44211-6, 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 6.1

6.2 Design Rationale

6.2.1 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.

6.2.2 Design Philosophy

6.2.2.1 Linguistic requirements

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.

6.2.2.2 Engineering requirements

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.

6.2.2.3 Miscellaneous requirements

Rationalized – provide a rationale for key design decisions.

6.2.3 Mapping Summary

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

IDL Construct	Dylan Construct
Foo_Bar	Foo-Bar
Foo::Bar	Foo/Bar

Table 6.2

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 ...
long[a]	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 / <condition>) ...
void Foo (...)	define open generic Foo (...) => ()
any	CORBA / <any>

Table 6.2

a. i

Notes: [a] or equivalent expression or literal.

6.3 Lexical Mapping

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

6.3.1 Identifiers

6.3.1.1 Background

IDL identifiers are 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 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: - + ~ ? /

6.3.1.2 Specification

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 an 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 6.4, "The Mapping of IDL to Dylan".

6.3.1.3 Examples

IDL	Dylan
fusion	fusion
Fusion	`Fusion` [a]
cold_fusion	cold-fusion
let	let`-%`
RED_SETTER	RED-SETTER-%
isExothermic	isExothermic[b]

Table 6.3

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.

6.3.2 Literals

IDL literals shall be mapped to lexically equivalent Dylan literals or semantically equivalent Dylan expressions.

6.3.2.1 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 append 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 append the characters “#x” in their place.

6.3.2.2 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(opt) decimal-integer(opt) . decimal-integer exponent(opt)
  sign(opt) decimal-integer . decimal-integer(opt) exponent(opt)
  sign(opt) decimal-integer exponent
exponent:
  E sign(opt) decimal-integer
sign:
  one of + -
```

The case of the exponent marker “E” is not significant.

Specification

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

6.3.2.3 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 6.4

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 6.5

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.

6.3.3 Fixed Point Decimals

6.3.3.1 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.

6.3.3.2 Specification

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

6.3.4 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.

6.3.4.1 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 ones 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 6.6

6.4 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.

6.4.1 Names

6.4.1.1 Identifiers

The lexical mapping of identifiers shall be as specified in Section 6.3.1.2, “Specification”

6.4.1.2 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 6.4.2, “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
```

6.4.2 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 6.4.3, "The DYLAN-ORB Library"), 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.

6.4.3 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.

6.4.4 Mapping Modules

6.4.4.1 Background

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

6.4.4.2 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 6.4.1.2, “Scoped Names”.

6.4.4.3 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 (e.g. in IDL **`foo::bar`**).

6.4.4.4 Examples

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

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

Note: The mapping of interfaces is covered in the next section.

6.4.5 Mapping for Interfaces

6.4.5.1 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.

6.4.5.2 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 6.5.2.1, “Object References”and servants (see section Section 6.5.5.1, “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 . 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.

6.4.5.3 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.

6.4.5.4 Examples

```
// IDL
interface T34 {};

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

6.4.6 Mapping for Interface Inheritance

6.4.6.1 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.

6.4.6.2 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.

6.4.6.3 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.

6.4.6.4 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;
```

6.4.7 Mapping for Constants

6.4.7.1 Specification

IDL constant declarations shall be mapped to Dylan constant definitions. IDL constant expressions are mapped as defined in Section 6.2.3, “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.

6.4.7.2 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;
```

6.4.8 Mapping for Basic Types

6.4.8.1 Overall

Background

IDL and Dylan both provide a number of basic built in types and 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.

6.4.8.2 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 6.7

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 either be mapped to the following Dylan classes.

IDL Integer Type	Dylan CORBA Library Integer Type

Table 6.8

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 6.8

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);
```

6.4.8.3 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 6.9

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 6.10

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;
```

6.4.8.4 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”);
```

6.4.8.5 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';
```

6.4.9 Wide Characters

6.4.9.1 Background

IDL wide characters are implementation defined.

Dylan defines a <unicode-string> type.

6.4.9.2 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>.

6.4.9.3 Rationale

The natural mapping to Dylan.

6.4.9.4 Examples

```
// IDL
const wchar ALEPH = 'a';

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

6.4.9.5 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;
```

6.4.9.6 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..255.

Rationale

As above.

Examples

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

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

6.4.9.7 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.

6.4.10 Mapping for Constructed Types

6.4.10.1 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>;
```

6.4.10.2 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>);
```

6.4.10.3 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>);
```

6.4.10.4 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 (e.g. 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;

…
```

6.4.10.5 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 runtime).

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>);
```

6.4.10.6 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>;
```

6.4.10.7 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>;
```

6.4.10.8 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 an 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));
```

6.4.11 Mapping for Exceptions

6.4.11.1 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.

6.4.11.2 Specification

IDL exceptions shall be mapped onto *sealed* Dylan conditions that are subclasses of `CORBA/<condition>`. 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.

6.4.11.3 Rationale

This is the natural mapping of IDL exceptions into the Dylan language.

6.4.11.4 Examples

```
// IDL
exception melt_down {
short seconds_remaining;
};

// Dylan (using slots)
define class <melt-down> (CORBA/<condition>)
  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>);
```

6.4.12 Mapping for Operations

6.4.12.1 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.

6.4.12.2 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.

6.4.12.3 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 doesn't have to worry about them by default.

6.4.12.4 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/<condition>)
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>);
```

6.4.13 Mapping for Attributes

6.4.13.1 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.

6.4.13.2 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.

6.4.13.3 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.

6.4.13.4 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>);

…
```

6.4.14 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.

6.4.15 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.

6.5 The Mapping of Pseudo-Objects to Dylan

6.5.1 Introduction

6.5.1.1 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.

6.5.1.2 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.

6.5.1.3 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.

6.5.2 ORB Interface

6.5.2.1 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;
```

6.5.2.2 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)
…
```

6.5.2.3 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
```

6.5.3 Dynamic Invocation Interface

6.5.3.1 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));
```

6.5.4 Dynamic Skeleton Interface

6.5.4.1 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;
```

6.5.5 The Portable Object Adapter

6.5.5.1 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;
```