IONA®

Orbix®

First Northern Bank Developer's Introduction

Version 6.1, December 2003

Making Software Work Together™

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Updated: 13-Feb-2004

M3125

Contents

List of Figures vii

Preface ix

Part I CORBA Bank Application

Chapter 1 Back-End CORBA Server 3
- **Design of the Back-End Server** 4
- **IDL for the Back-End Server** 7
- **Architecture** 13
- **Designing the POA Hierarchy** 14
- **Implementing the Account Interfaces** 17
  - Implementing Interfaces Using the Delegation Approach 18
  - Implementation of the Account Interface 21
  - Persistence Mechanism for Account Objects 30
- **Lifecycle of Account Objects** 32
- **Implementing the AccountMgr Interface** 36
  - Implementing Interfaces Using the Inheritance Approach 37
  - Implementation of the AccountMgr Interface 39
- **Publishing the AccountMgr Object Reference** 43

Chapter 2 Middle-Tier CORBA Server 47
- **Design of the Middle-Tier Server** 48
- **IDL for the Middle-Tier Server** 51
- **Designing the POA Hierarchy** 55
- **Resolving the AccountMgr Object Reference** 57
- **Implementing the BusinessSession Interfaces** 60

Chapter 3 Java CORBA Client 63
- **Design of the CORBA Client** 64
  - The Open Account Dialog 66

The New Account Dialog 67
The Lodge Funds Dialog 69
The Withdraw Funds Dialog 70
The Transfer Funds Dialog 71
Using Forte for Java and NetBeans 72
Resolving the BusinessSessionManager Object Reference 77
Implementation of the Java CORBA Client 81
Implementation of the Open Account Dialog 83

Part II J2EE Internet Banking

Chapter 4 J2EE AllDayBanking Application 89
Architecture of the J2EE Application 90
Overview of the J2EE Development Cycle 92
Source Code Organization (EARSCO) 94
Building and Packaging the J2EE Application 98
Directory Structure in an EAR File 100
Directory Structure in an EJB Module JAR File 102
Directory Structure in a Web Module WAR File 105

Chapter 5 Accessing the CORBA Back-End 109
Overview of the EJB to CORBA Link 110
Using Orbix Connect and JBoss 112
Creating the IDL Stub JAR File 119
Accessing the Stub JARs from EJB 121

Chapter 6 EJB Middle-Tier 123
The InetAccount Session Bean 124
Anatomy of a Session Bean 125
EJB Session Bean Life Cycle Methods 128
Session Bean Configuration 131
The User Entity Bean 135
Anatomy of an Entity Bean 136
EJB Entity Bean Life Cycle Methods 139
Entity Bean Configuration 142
Container-Managed Persistence in JBoss 145

Chapter 7 J2EE Presentation Layer 149
Overview of the Presentation Layer 150
Worker Beans 154
The CustomerSession Bean 155
The NewRegSession Bean 158
The WebHelper Class 160
Using a JSP to Process a Web Form 161
The Login Web Form 163
The New User Registration Web Form 167
Using a JSP to Access an Enterprise Bean 173

Part III COMet and .NET Clients

Chapter 8 Visual Basic COMet Client 177
Overview of the Visual Basic Client 178
Implementation of the Visual Basic Client 182
Starting the ATM Session 183
Showing Account Details 186
Withdrawing Cash 188

Chapter 9 C# .NET Client 191
Overview of the C# Client 192
Implementation of the C# Client 196
Importing .NET Metadata 197
Initializing the Online Purchasing Manager Client 198

Index 201

List of Figures

Figure 1: Inheritance Hierarchy for Account Types 5
Figure 2: POA Hierarchy for the Back-End Server 15
Figure 3: Relationship Between a Tie Object and its Delegate 18
Figure 4: Classes and Interfaces Needed for the Delegation Approach 19
Figure 5: Java Inheritance Hierarchy for Account Types 21
Figure 6: Lifecycle of an Account Object 32
Figure 7: Creating an Account Object 33
Figure 8: Opening an Account Object 34
Figure 9: Updating an Account Object 35
Figure 10: Classes Needed for the Inheritance Approach 38
Figure 11: Publishing an Object Reference in the CORBA Naming Service 44
Figure 12: Inheritance Hierarchy for BusinessSession Types 48
Figure 13: POA Hierarchy for the Middle-Tier Server 55
Figure 14: Resolving the AccountMgr Object Reference 57
Figure 15: Java Inheritance Hierarchy for the Delegate Objects 60
Figure 16: The Main Screen of the Java CORBA Client 65
Figure 17: The Open Account Dialog Window 66
Figure 18: The New Account Dialog Window 67
Figure 19: The Lodge Funds Dialog Window 69
Figure 20: The Withdraw Funds Dialog Window 70
Figure 21: The Transfer Funds Dialog Window 71
Figure 22: Editing the Transfer Funds Dialog within the Forte for Java IDE 73
Figure 23: Viewing the transferFunds.java file in the Source Editor 75
Figure 24: Establishing a Connection to the Middle-Tier Server 77
Figure 25: Relationship Between the Main Screen and a Dialog Window 81
Figure 26: Architecture of the J2EE AllDayBanking Application 90

Figure 27: The J2EE Development Cycle 92
Figure 28: The Enterprise Application Archive Source Code Organization 95
Figure 29: Standard Layout of an EAR File 100
Figure 30: Layout of an EJB Module JAR File 102
Figure 31: Standard Layout of a Web Module WAR File 105
Figure 32: The EJB Middle-Tier Accesses the CORBA Back-End 110
Figure 33: EJB to CORBA Connectivity Using Orbix Connect and JBoss 113
Figure 34: Structure of the InetAccountBean Session Bean Class 126
Figure 35: Structure of the UserBean Entity Bean Class 137
Figure 36: Overview of Container-Managed Persistence 145
Figure 37: Overview of the J2EE Presentation Layer for AllDayBanking 150
Figure 38: Processing Web Form Data Using a JSP 161
Figure 39: The Login Page of the AllDayBanking Application 163
Figure 40: The New User Registration Page of the AllDayBanking Application 168
Figure 41: Architecture of the Visual Basic ATM Client Application 178
Figure 42: The ATM Client Welcome Screen 180
Figure 43: Architecture of the C# Online Purchasing Client Application 192
Figure 44: The Online Purchasing Manager C# Client 195

Preface

This document provides developers with a compact overview of the technologies supported by Orbix product. The CORBA, J2EE, COMet, and .NET connector technologies are introduced and discussed in the context of the First Northern Bank demonstration, which provides a source of examples throughout. Detailed discussions of the topics introduced in this document can be found in the relevant Orbix developer guides.

Audience

This book is aimed at the following developers:

- *CORBA developer*s—who want to develop server or client applications in Java. The prerequisites are a good knowledge of Java and familiarity with basic CORBA concepts.
- *J2EE developer*s—who want to develop Enterprise JavaBean servers and Web applications. The prerequisites are a good knowledge of Java and a basic knowledge of XML.
- *Visual Basic developers*—who want to write an application that communicates with a CORBA server through IONA's COMet bridge.
- *C# developers*—who want to write an application that communicates with a CORBA server through IONA's .NET Connector.

Organization of this guide

This guide is divided as follows:

Part I "CORBA Bank Application"

This part discusses the CORBA components of the First Northern Bank demonstration. The CORBA bank application has three tiers: CORBA back-end, CORBA middle-tier, and Java CORBA client.

Part II "J2EE Internet Banking"

This part begins with an overview of the J2EE development cycle and then discusses the J2EE components of the First Northern Bank demonstration. The J2EE Internet banking application has three tiers: CORBA back-end, EJB middle-tier, and Web presentation layer.

Part III "COMet and .NET Clients"

This part provides a brief introduction to developing Visual Basic COMet clients and C# .NET clients.

Related documentation

The following documents also discuss the FNB demonstration:

- *First Northern Bank Business Case*
- *First Northern Bank Tutorial*

The following documents complement this guide by providing a more detailed discussion of the concepts introduced here:

- *CORBA Programmer's Guide*
- *J2EE Technology Developer's Guide*

The latest updates to the Orbix documentation can be found at http://www.iona.com/support/docs.

Additional resources

The IONA knowledge base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

The IONA update center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to docs-support@iona.com .

Typographical conventions

This guide uses the following typographical conventions:

`Constant width` — Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on the screen. For example:

```
#include <stdio.h>
```

Italic — Italic words in normal text represent *emphasis* and *new terms*.

Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:

```
% cd /users/your_name
```

Note: Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt — When a command's format is the same for multiple platforms, a prompt is not used.

`%` — A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.

`#` — A number sign represents the UNIX command shell prompt for a command that requires root privileges.

`>` — The notation > represents the DOS or Windows command prompt.

```
. . .
.
.
.
```

Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

`[ ]` — Brackets enclose optional items in format and syntax descriptions.

`{ }` — Braces enclose a list from which you must choose an item in format and syntax descriptions.

`|` — A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Part I

CORBA Bank Application

In this part

This part contains the following chapters:

Back-End CORBA Server	page 3
Middle-Tier CORBA Server	page 47
Java CORBA Client	page 63

CHAPTER 1

Back-End CORBA Server

This chapter discusses the design and implementation of the back-end CORBA server. Starting from a high-level design, the object interfaces are defined in the OMG interface definition language (IDL) and then implemented in Java.

In this chapter

This chapter discusses the following topics:

Design of the Back-End Server	page 4
IDL for the Back-End Server	page 7
Architecture	page 13
Designing the POA Hierarchy	page 14
Implementing the Account Interfaces	page 17
Lifecycle of Account Objects	page 32
Implementing the AccountMgr Interface	page 36
Publishing the AccountMgr Object Reference	page 43

Design of the Back-End Server

Purpose of the back-end server

The purpose of the back-end server is to provide the basic business objects for the bank application—in this demonstration, `Account` objects. The back-end server has the following general characteristics:

- Provides close integration with persistent storage—the CORBA back-end server consists of a wrapper around a database that stores the business data.
- Provides an implementation of `Account` CORBA objects—the account data thus becomes accessible to other distributed applications.
- Ignores presentation requirements—the back-end server is not concerned with the way in which clients access and use the `Account` objects. This is left to other parts of the distributed application.

Object-oriented design and CORBA

CORBA fits in well with object-oriented design methodologies. For example, a formal design specified in UML (Unified Modelling Language) can be used as the basis for defining the interfaces for CORBA objects.

The use of distributed technology does have an impact on the formal design, however. For example, for a class that will be implemented as a CORBA type, it is advisable to modify the design to minimize the number of remote invocations that are required to use the class.

CORBA object types

Figure 1 shows the inheritance hierarchy for the object types implemented in the back-end server.

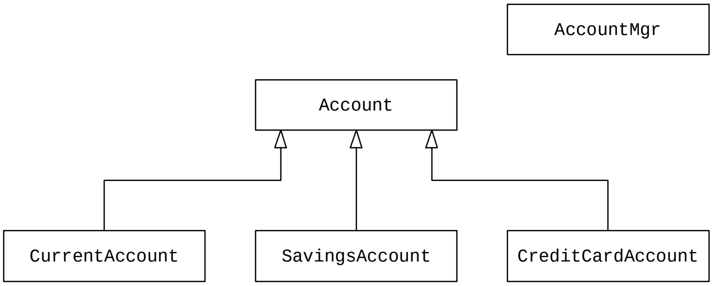


Figure 1: *Inheritance Hierarchy for Account Types*

AccountMgr Type

A single object of `AccountMgr` type is created to manage and provide access to the `Account` objects. Methods defined on the `AccountMgr` type follow the pattern for a *factory/finder* type. Because constructor methods cannot be exposed to remote clients, a factory object such as `AccountMgr` is needed in order to:

- Create new `Account` objects.
- Find existing `Account` objects—two alternative search methods are supported:
    - lookup by account number, and
    - listing all accounts of a particular type.

Account Type

The `Account` class is an abstract base class for the other account types. A number of attributes are defined on the `Account` class:

- Account number.
- Owner details (name and address).
- A list of recent transactions.

Methods are also defined on the `Account` class, as follows:

- Deposit and withdraw cash.
- Transfer money in or out of the account.

CurrentAccount Type

The `CurrentAccount` type inherits from `Account`. The following attribute is added:

- `overdraftlimit`—a readonly attribrute that returns the current overdraft limit.

The following method is added:

- `approveNewOverdraft()`—request approval for a new overdraft limit. The method returns `TRUE` if the new limit is approved.

CreditCardAccount Type

The `CreditCardAccount` type inherits from `Account`. The following attributes are added:

- Credit limit.
- Interest rate on overdue payments.

The following methods are added:

- Authorize an amount of money to be spent.
- Make a purchase, based on an authorization code.
- Calculate the interest due on late payments.

SavingsAccount Type

The `SavingsAccount` type inherits from `Account`, adding no new attributes or methods.

IDL for the Back-End Server

OMG interface definition language

The OMG interface definition language (IDL) is a purely declarative language, with a syntax similar to C++ and Java, that is used to define the interfaces for CORBA objects. The most important entities that can be defined in IDL are *IDL interfaces*, which are analogous to C++ abstract classes or Java interfaces.

Language neutrality of IDL

The advantage of OMG IDL is that it enables you to define distributed interfaces in a language-neutral manner.

A server developer can use IDL to define the service provided to clients, irrespective of the language or platform used on the server side. Conversely, a client programmer can use IDL as a blueprint for accessing the service, irrespective of the language or platform used on the client side.

The IDL compiler

To access the definitions expressed in IDL, it is necessary to compile the IDL into a target language such as C++ or Java. This is accomplished using the *IDL compiler*, which takes an IDL file as input and generates stub files and skeleton files as output.

Orbix provides the IDL compiler as a command line tool, `idl`.

Account IDL

The code listing in Example 1 shows the main IDL file used by the back-end server, `idl/Account.idl`. This IDL file defines all of the CORBA interfaces implemented by the back-end server.

Example 1: *The Account IDL File*

```
   // IDL
1  #ifndef ACCOUNT_IDL
   #define ACCOUNT_IDL

   // Exceptions raised in this file

2  module bankobjects {
3      typedef long accountNum;
4      typedef sequence<accountNum> accountNumList;
```

Example 1: *The Account IDL File*

```
5       exception INSUFFICIENT_FUNDS {};
        exception CANNOT_CLOSE_ACCOUNT {};
        exception ACCOUNT_DOESNT_EXIST {};
        exception FAILED_TO_AUTHORIZE {};

6       struct address {
            string address_1;
            string address_2;
            string address_3;
        };

        // Stucture to hold information on what a customer
        // is doing with the bank
        struct BankTransaction {
            short id;
            string date;
            string record_type;
            string value;
        };

7       typedef sequence<BankTransaction> AccountTransactions;
8       interface Account;

9       interface AccountMgr  {
10          Account openAccount ( in accountNum accountNumber)
                raises (ACCOUNT_DOESNT_EXIST);
            Account newAccount (in string accountType);
            void closeAccount (in accountNum accountNumber )
                raises (CANNOT_CLOSE_ACCOUNT);

            accountNumList getCurrentAccountList ();
            accountNumList getCreditCardList ();
        };

        interface Account {
11          readonly attribute accountNum accountnumber;
            readonly attribute address addr;
            readonly attribute string accountType;

12          attribute string firstname;
            attribute string lastname;

            readonly attribute float accountBalance;
```

Example 1: *The Account IDL File*

```
        readonly attribute AccountTransactions
            recentTransactions;

        // Update methods
        boolean makeLodgement (in float amount );
        boolean withdrawFunds (in float amount)
          raises (INSUFFICIENT_FUNDS);
        boolean updateAddress (in address newAddress);

        void transferFundsIn (in float amount );
        void transferFundsOut (in float amount )
          raises (INSUFFICIENT_FUNDS);

        // Admin stuff
        void sendStatement ();
    };

13  interface CurrentAccount : Account {
        readonly attribute float overdraftLimit;

        // Account maintenace routines
        boolean approveNewOverdraft (in float amount);
    };

    interface SavingsAccount : Account {
    };

    typedef short authorizationCode;

    interface CreditCardAccount : Account {
        attribute float limit;
        attribute float interest_rate;

        // Calculate how much interest is owed on this account
        float calculateInterest ();

        // Basic operations on a credit card
        authorizationCode authoriseAmount (in float amount)
          raises (FAILED_TO_AUTHORIZE);
        boolean makePurchase (in string vendor, in float amount,
                          in authorizationCode auth_code);
    };

}; // Module
#endif //ACCOUNT_IDL
```

The preceding code can be explained as follows:

1. An IDL file can contain preprocessor macros, similar to the C and C++ languages. The start of a macro is signalled by a # character at the beginning of a line.

   In this example, the `#ifndef`, `#define`, and `#endif` preprocessor macros guard against multiple inclusion of this file into other IDL files.

2. The definitions in this file are enclosed within the `bankobjects` module. An IDL module is a scooping mechanism for IDL (conceptually similar to a namespace in C++ or a package in Java).

   All of the entities defined in the scope of the `bankobjects` module gain `bankobjects::` as a prefix. For example, `bankobjects::Account` is the fully scoped identifier for the `Account` interface.

3. The `typedef` construction is grammatically similar to `typedef` in C and C++. In this example, `accountNum` becomes a synonym for the IDL `long` type (32-bit signed integer).

4. This line defines a *sequence type*, `accountNumList`, defined as an unbounded sequence of integers, `accountNum`. A sequence is similar to a one-dimensional array except that its length can be arbitrary.

   For example, the IDL-to-Java mapping specifies that the IDL sequence type, `accountNumList`, maps to a Java array, `accountNum[]`, where the size of the Java array can be chosen arbitrarily.

5. This line and the following lines define some IDL *user exception* types. The syntax for declaring an IDL user exception is similar to the syntax of a C++ `struct`, except that the `struct` keyword is replaced by the `exception` keyword. The exception definitions shown here have an empty body, `{}`, because there is no data associated with these exceptions.

6. The syntax for declaring an IDL `struct` is similar to the syntax of a C++ `struct`. The closest Java analogy is a class that declares only member variables.

   For example, the `address` struct type contains three strings corresponding to the three fields of an address, `address_1`, `address_2`, and `address_3`.

7. The `typedef` declares an unbounded sequence, `AccountTransactions`, that holds a list of `BankTransaction` structs. A sequence should always be declared using a `typedef` construction.
8. This is an example of a forward declaration of an interface, `Account`. This enables the `Account` type to be referenced before it is defined. The actual definition of the `Account` interface appears further on.
9. This line introduces the definition of an IDL interface, `AccountMgr`. Interfaces are the most important sort of definition in IDL. An IDL interface defines the attributes and operations for CORBA objects of a particular type.
10. This line shows an example of an *IDL operation*, `openAccount()`. The syntax for declaring an IDL operation is similar to the definition of a member function in C++ or a method in Java. A `raises()` clause introduces the list of user exceptions that can be thrown by this operation.
11. Declaring a readonly attribute in an interface specifies that the interface implementation will include an accessor method that enables you to retrieve the attribute value.

    For example, when the `accountnumber` readonly attribute is mapped to Java, the following method appears in the `Account` implementation class:

    ```
    // Java
        ...
        // In the scope of the Account implementation class:
        int accountnumber() {
            // return the value of the account number
            ...
        };
    ```

12. Declaring a plain attribute in an interface specifies that the interface implementation will include both an accessor and a modifier method that enables you both to retrieve and to update the attribute’s value.

For example, when the firstname attribute is mapped to Java, the following pair of methods appear in the Account implementation class:

```
// Java
    ...
    // In the scope of the Account implementation class:
    string firstname() {
        // return the firstname string
        ...
    };
    void firstname(string s) {
        // update the firstname string value
        ...
    };
```

13. The CurrentAccount interface inherits from Account. IDL inheritance is indicated using : (colon). Multiple inheritance is supported in IDL.

Architecture

Overview

After defining the application IDL, a range of architectural choices remain open for the implementation of the back-end server. The following aspects of the implementation architecture can be decided at this point:

- Programming language.
- Code generation.
- Persistence mechanism.
- Services.

Programming language

Because of the OMG IDL's language neutrality, you can choose between a range of programming languages. COBOL, PL/I, Java, C, and C++ mappings have all been standardized by the OMG.

Code generation

Orbix provides a CORBA Code Generation Toolkit (CCGT) for developing CORBA applications in C++ and Java. The CCGT takes an IDL file as input and generates an outline C++ or Java application based on the IDL.

Code generation can be particularly beneficial in the context of large-scale projects using a lot of IDL. Customization of the CCGT is possible (and recommended) if you have some expertise in TCL (Tool Command Language) programming.

Persistence mechanism

You can use any standard persistence mechanism, such as a commercial database, file-based storage, or serialized objects. CORBA does not constrain your choice in any way, but it does provide an extra option: the CORBA Persistent State Service (PSS), which is a persistence layer that is closely integrated with CORBA technology.

Services

Orbix provides a range of integrated services, including the following:

- Security with SSL/TLS.
- Transaction support with OTS-Lite or full OTS.
- Session management, using the session management plug-in.

Designing the POA Hierarchy

Role of the POA

The role of the Portable Object Adapter (POA) is to manage a collection of CORBA objects in a specific way. There can be more than one POA in an application, with each POA instance configured to manage different collections of CORBA objects in different ways.

The main responsibilities of the POA are the following:

- *Activating CORBA objects*—A CORBA object cannot receive CORBA invocations until it is activated. The POA then becomes responsible for routing invocations to the CORBA object.
- *Managing the lifecycle of CORBA objects*—a POA instance can be configured to manage the object lifecycle in one of several different ways:
    - Some POA configurations are designed to manage CORBA objects that are created and activated once.
    - Other POA configurations are designed to load and unload CORBA objects dynamically, in response to demand. See also "Lifecycle of Account Objects" on page 32.
- *Defining the threading policy*—a POA instance can be configured to be either single or multi-threaded:
    - In a single-threaded POA, a CORBA object is guaranteed to receive invocations sequentially.
    - In a multi-threaded POA, a CORBA object can receive invocations concurrently.

Activation

Activation is a crucial step that makes a CORBA object accessible to remote clients.

Activation affects a CORBA object as follows:

- *Associates the CORBA object with a particular POA instance*—the POA is then responsible for routing invocations to the object implementation.
- *Gives an identity to the CORBA object*—by associating an *object ID* with the object. An object ID is an array of bytes that, together with the associated POA name, uniquely identifies the CORBA object.

POA hierarchy for the back-end server

Figure 2 shows the POA hierarchy used for the back-end server. The root POA (which is present at the root of every POA hierarchy) has two children:

- A POA for managing `AccountMgr` objects (named `AccountManager`),
- A POA for managing `Account` objects (named `BankObjects`).

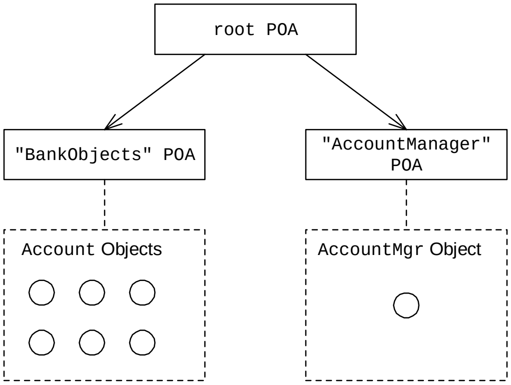


Figure 2: *POA Hierarchy for the Back-End Server*

POA policies

A POA instance is configured by setting its *POA policies*, which can only be set at the time the POA instance is created.

For example, the following policy types are often customized when a POA is created:

- The LifespanPolicy—object life spans are either bounded by a single run of the application (TRANSIENT), or they are unbounded and valid for many runs of the application (PERSISTENT).
- The IdAssignmentPolicy—object IDs can be assigned explicitly by the developer (USER_ID), or generated automatically by the ORB (SYSTEM_ID).
- The ThreadPolicy—can be either single threaded (SINGLE_THREAD_MODEL), or multi-threaded (ORB_CTRL_MODEL).

AccountManager POA

The AccountManager POA is created to manage the AccountMgr object (only one object of AccountMgr type is ever created). The AccountMgr object is created and activated when the application starts up and remains active for as long as the application is running.

Because the lifecycle of the AccountMgr object is very simple, there are no special requirements on the AccountManager POA which, therefore, uses mostly default POA policies.

BankObjects POA

The BankObjects POA is created to manage Account objects. The number of Account objects is potentially very large and it is not practical to store all of the objects in memory at the same time. The back-end server adopts the strategy of loading Account objects into memory only when they are needed (that is, in response to bankserver::AccountMgr::openAccount() operation invocations).

Because of the special requirements for managing the lifecycle of Account objects, the BankObjects POA is specially configured to use a *servant locator*. See “Lifecycle of Account Objects” on page 32 for details.

Implementing the Account Interfaces

Overview

One the server developer's main tasks is to implement the back-end IDL interfaces. This section describes the general approach to implementing the Account interfaces for the back-end server (`Account`, `CurrentAccount`, and `CreditCardAccount`), focusing mainly on the CORBA aspects.

In this section

This section contains the following subsections:

Implementing Interfaces Using the Delegation Approach	page 18
Implementation of the Account Interface	page 21
Persistence Mechanism for Account Objects	page 30

Implementing Interfaces Using the Delegation Approach

Overview

There are two alternative approaches to implementing an IDL interface in Java:

- *The delegation (or TIE) approach*—as described in this subsection.
- *The inheritance approach*—as described in "Implementing Interfaces Using the Inheritance Approach" on page 37.

In Java, the delegation approach predominates because it gets around the Java limitations on multiple inheritance. By contrast, the inheritance approach runs into difficulties as soon as the IDL interface to be implemented inherits from just one other IDL interface.

The delegation approach

With the delegation approach, a single CORBA object is implemented using two Java objects: a tie object, of *InterfaceName*`POATie` type, and a delegate object, conventionally of *InterfaceName*`Delegate` type. Figure 3 shows the relationship between a tie object and its delegate.

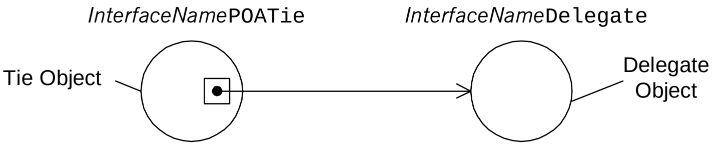


Figure 3: *Relationship Between a Tie Object and its Delegate*

Together, the tie object and its delegate cooperate to provide the implementation of the IDL attributes and operations, as follows:

- The delegate object has the code that implements the IDL attributes and operations.

  The *InterfaceName*`Delegate` class is written by the application developer.
- The tie object caches a reference to the delegate object and uses the cached reference to forward method invocations to the delegate. The tie object is a *servant* (in Java, it inherits from the `org.omg.PortableServer.Servant` interface).

The *InterfaceName*`POATie` class is generated automatically by the IDL compiler.

Servants

A servant is an object that provides the implementation code for an IDL interface. It is incorrect, however, to regard a servant as a CORBA object. A CORBA object is composed of a servant and an identity (object ID), a composition created by *activating* the CORBA object. A servant on its own has no identity.

In the delegation approach the tie object is effectively the servant object, because it inherits from the servant base class. However, there is a sense in which both the tie object and the delegate object together constitute the servant because it is the combination of these two objects that provides the implementation code.

Instantiating a TIE servant

A tie servant for the `CurrentAccount` type is instantiated as follows:

```
// Java
package bankobjects;
...
    // Step 1: Create the delegate object.
    CurrentAccountDelegate deleg = new CurrentAccountDelegate();

    // Step 2: Create the TIE object.
    org.omg.PortableServer.Servant tie_servant
        = new CurrentAccountPOATie(deleg);
```

Classes and interfaces needed for the delegation approach

Figure 4 shows some of the Java classes and interfaces needed for the delegation approach.

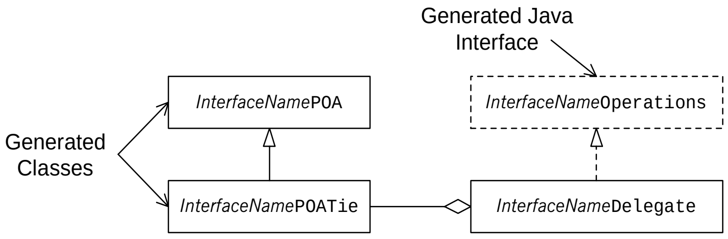


Figure 4: *Classes and Interfaces Needed for the Delegation Approach*

The *InterfaceName*`Delegate` class must implement the *InterfaceName*`Operations` Java interface. This ensures that all of the *InterfaceName* operations and attributes are actually implemented by the delegate class.

The *InterfaceName*`POATie` class inherits from the *InterfaceName*`POA` class, which ensures that it is the correct type of servant for the *InterfaceName* IDL interface.

Implementing the delegate class

There are two possible starting points for implementing the delegate class:

- *Use the CORBA Code Generation Toolkit*—the code generation toolkit can create the outline of a working application based on an IDL file.

  For example, by generating code from the `Account.idl` file using the `java_poa_genie.tcl` code generation genie you can obtain an initial version of the `Account` delegate class (you will need to specify the `-tie` option to the genie). See the *CORBA Programmer's Guide* for details of this approach.
- *Use a stub file as a template for the delegate class*—one of the steps involved in building a CORBA application is to compile your IDL using the *IDL compiler*. This produces *stub files* in your target programming language (for example C++ or Java).

  Some of the stub files have a form that is similar to the form required for a delegate class. These stub files can be copied and modified appropriately to provide the initial versions of your delegate classes.

  Note: This approach (stub file as a template) is recommended only for advanced CORBA developers.

Implementation of the Account Interface

Overview

The Account IDL interface is implemented using the delegation approach (see “Implementing Interfaces Using the Delegation Approach” on page 18). The main task for the developer is to implement the delegate class, AccountDelegate, which has corresponding methods for all of the Account operations and attributes.

Java inheritance hierarchy

Usually, the most convenient way to implement an IDL inheritance hierarchy (see Figure 1 on page 5) is to implement a Java inheritance hierarchy amongst the delegate classes that parallels the IDL one. Figure 5 shows the resulting Java inheritance hierarchy for the Account types.

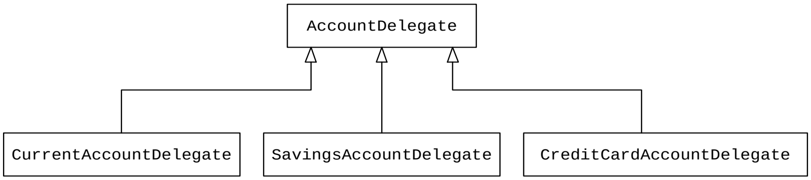


Figure 5: *Java Inheritance Hierarchy for Account Types*

The AccountDelegate class is the base class for all of the other account types. It is never instantiated directly, but it provides common code and state for the subclasses.

The AccountDelegate class

Example 2 shows an outline of the code for the AccountDelegate class.

Example 2: *The AccountDelegate Class (Sheet 1 of 7)*

```
  // Java
1 package bankobjects;

  import org.omg.CORBA.*;
  import org.omg.CORBA.portable.*;
  import org.omg.PortableServer.POA.*;
  import org.omg.PortableServer.*;
  import org.omg.PortableServer.ServantLocatorPackage.*;
  import org.omg.PortableServer.POAPackage.*;
```

Example 2: *The AccountDelegate Class (Sheet 2 of 7)*

```
  import java.io.*;
  import java.util.*;
  import java.text.*;

  import org.omg.CosNaming.*;
  import org.omg.CosNaming.NamingContextPackage.*;

2 public class AccountDelegate
    implements AccountOperations, Serializable
  {

    // Private data
    private int m_accountNumber;
    private address m_address;

    private String m_lastname;
    private String m_firstname;

    // Transaction Data, hold the last 50 transactions
    protected BankTransaction[] m_translist = new
     BankTransaction[50];
    protected int next_trans_location = 0;
    protected short next_id_num = 100;
    protected String m_accountType;

    // Balance Information
    protected float m_balance;
    public String m_filename;

3   public AccountDelegate ()
    {
      m_translist[next_trans_location] = new BankTransaction();

      m_translist[next_trans_location].id = 0;
          // get the date
      Calendar cal = Calendar.getInstance();
      DateFormat df = DateFormat.getInstance ();

      m_translist[next_trans_location].date =
     df.format(cal.getTime());
      m_translist[next_trans_location].record_type = "Opened";
      m_translist[next_trans_location].value = "0.00";

      next_trans_location++;
```

Example 2: *The AccountDelegate Class (Sheet 2 of 7)*

```
import java.io.*;
import java.util.*;
import java.text.*;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

2 public class AccountDelegate
  implements AccountOperations, Serializable
{

  // Private data
  private int m_accountNumber;
  private address m_address;

  private String m_lastname;
  private String m_firstname;

  // Transaction Data, hold the last 50 transactions
  protected BankTransaction[] m_translist = new
   BankTransaction[50];
  protected int next_trans_location = 0;
  protected short next_id_num = 100;
  protected String m_accountType;

  // Balance Information
  protected float m_balance;
  public String m_filename;

3 public AccountDelegate ()
  {
    m_translist[next_trans_location] = new BankTransaction();

    m_translist[next_trans_location].id = 0;
        // get the date
    Calendar cal = Calendar.getInstance();
    DateFormat df = DateFormat.getInstance ();

    m_translist[next_trans_location].date =
   df.format(cal.getTime());
    m_translist[next_trans_location].record_type = "Opened";
    m_translist[next_trans_location].value = "0.00";

    next_trans_location++;
```

Example 2: *The AccountDelegate Class (Sheet 3 of 7)*

```
    }

    // Attributes that are defined on the IDL Interface

4   public int accountnumber ()
    {
      return m_accountNumber;
    }

    public String accountType ()
    {
      return m_accountType;
    }

    public address addr ()
    {
      return m_address;
    }

    public String lastname ()
    {
      return m_lastname;
    }

    public void lastname (String lname )
    {
      m_lastname = lname;
      return;
    }

    public String firstname ()
    {
      return m_firstname;
    }

    public void firstname (String fname)
    {
      m_firstname = fname;
      return;
    }


    public BankTransaction[] recentTransactions ()
    {
      if ( m_translist == null) {
```

Example 2: *The AccountDelegate Class (Sheet 4 of 7)*

```
      System.err.println ("m_translist is null!!!");

    } // end of if ()

    int tlen =0;
    while ( m_translist[tlen] != null) {
      tlen++;
    }

    BankTransaction[] tt = new BankTransaction [tlen];
    for ( int j = 0; j < tlen; j++) {

      tt[j] = m_translist[j];
    } // end of for ()

    return tt;
  }

  public float accountBalance ()
  {
    return m_balance;
  }

  // Operations that are defined on the IDL Interface

5 public boolean makeLodgement (float amnt)
  {
    m_balance += amnt;

    /* 2 decimal places */
    java.text.DecimalFormat df2
      = new java.text.DecimalFormat("#######0.00");
    String g = df2.format(amnt);

    this.addTransaction ("Lodgement", g);

    return true;
  }


  public void transferFundsIn (float amnt)
  {

    System.out.println ("in transferFundsIn with " + amnt);
```

Example 2: *The AccountDelegate Class (Sheet 5 of 7)*

```
    m_balance += amnt;

    /* 2 decimal places */
    java.text.DecimalFormat df2
      = new java.text.DecimalFormat("#######0.00");
    String g = df2.format(amnt);

    this.addTransaction ("Transfer In", g);

    return;
  }

  public boolean withdrawFunds (float amnt)
    throws INSUFFICIENT_FUNDS
  {
    if ( m_balance < amnt) {
      throw new INSUFFICIENT_FUNDS ();
    } // end of if ()

    m_balance -= amnt;

    /* 2 decimal places */
    java.text.DecimalFormat df2
      = new java.text.DecimalFormat("#######0.00");
    String g = df2.format(amnt);

    this.addTransaction ("Withdrawal", g);

    return true;
  }

  public void transferFundsOut (float amnt)
    throws INSUFFICIENT_FUNDS
  {
    if ( m_balance < amnt) {
      throw new INSUFFICIENT_FUNDS ();
    } // end of if ()

    m_balance -= amnt;

    /* 2 decimal places */
    java.text.DecimalFormat df2
      = new java.text.DecimalFormat("#######0.00");
    String g = df2.format(amnt);
```

Example 2: *The AccountDelegate Class (Sheet 6 of 7)*

```
  this.addTransaction ("Transfer Out", g);

  return;
}

public boolean updateAddress (address addr)
{
  m_address = addr;
  return true;
}

public void sendStatement ( )
{
  System.out.println ("Sending a statement....");
  return;
}

// Routines just needed by this class and its sub-classes, not
 exposed via
// IDL, so they cannot be used by any CORBA clients.
6 public void setAccountNumber (int accountNum)
{
  m_accountNumber = accountNum;
}

protected void addTransaction (String type, String value )
{
  // Make sure that we don't exceed more than 50
 transactions...
  if ( next_trans_location == 50 )
  {
    next_trans_location = 0;
  }

  if ( m_translist[next_trans_location] == null ) {
    m_translist[next_trans_location] = new BankTransaction ();
  }

  m_translist[next_trans_location].id = next_id_num;
  next_id_num++;

  // get the date
  Calendar cal = Calendar.getInstance();
  DateFormat df = DateFormat.getInstance ();
```

Example 2: *The AccountDelegate Class (Sheet 7 of 7)*

```
     m_translist[next_trans_location].date =
    df.format(cal.getTime());
     m_translist[next_trans_location].record_type = type;
     m_translist[next_trans_location].value = value;

     next_trans_location++;

     return;
   }
 }
```

The preceding code can be explained as follows:

1. The AccountDelegate class is placed in the bankobjects Java package, which corresponds to the bankobjects IDL module. There is no requirement, however, to place your implementation classes in the same package as the other CORBA classes. You could place the AccountDelegate class in a completely different package if you prefer.
2. The AccountDelegate class implements the following Java interfaces:
   - bankobjects.AccountOperations—*required*. The AccountOperations Java interface declares methods for all the attributes and operations in the Account IDL interface.
   - java.io.Serializable—*optional*. Inheriting from the Serializable class makes it possible to persist an Account object using the Java serialization technique. See "Persistence Mechanism for Account Objects" on page 30.
3. The AccountDelegate() constructor is never called directly because the AccountDelegate class is intended to be used as a base class only (the subclass constructors call this constructor). You can define as many constructors as you like for the AccountDelegate class, but none of them will be accessible to remote CORBA clients.
4. From this line onward, plain attributes and readonly attributes are implemented: two overloaded Java methods for each plain attribute (get and set), and one Java method for each readonly attribute (get).
5. From this line onward, each of the IDL operations are implemented.

6. The methods from this line onward are designed for internal use by this class and its subclasses. They are not defined in IDL and are not accessible to remote CORBA clients.

Persistence Mechanism for Account Objects

Overview

The back-end server uses the standard Java serialization mechanism to make Account objects persistent.

Making the delegate classes serializable

To make a delegate class serializable, have it inherit from the `java.io.Serializable` interface either directly or indirectly. For example, the `AccountDelegate` class is declared as follows:

```
// Java
public class AccountDelegate
  implements AccountOperations, Serializable
{
    ...
}
```

Writing a serializable class to disk

A serializable object can be written to persistent storage (for example, a file on disk) by invoking `writeObject()` on a Java output stream. For example, the `AccountServantLocatorImpl` class defines the following method for writing `Account` objects to disk:

```
// Java
  ...
  void writeObject (String filename, java.lang.Object obj )
    throws IOException
  {
    ObjectOutputStream out_str = null;
    try {
      out_str = new ObjectOutputStream (
                        new FileOutputStream (filename)
                    );
    }
    catch (IOException e) {
         // Handle exception (not shown) ...
    }

    try {
      out_str.writeObject (obj);
    }
    catch (IOException e ) {
         // Handle exception (not shown) ...
    } finally {
        // Flush and close the output stream (not shown) ...
    }
    return;
  }
```

Lifecycle of Account Objects

An Account lifecycle

Figure 6 shows the lifecycle of an `Account` object as time evolves from left to right across the diagram.

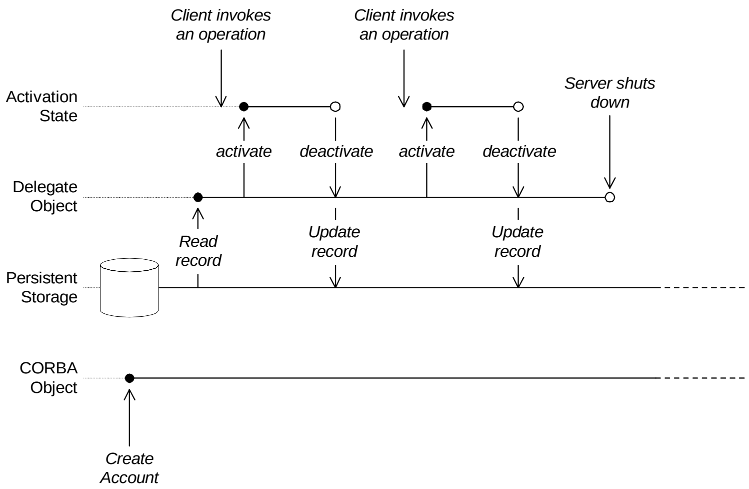


Figure 6: *Lifecycle of an Account Object*

Figure 6 distinguishes between the different aspects of the `Account` object, as follows:

- *CORBA object*—is created by making a record of the account state in persistent storage. The CORBA object endures as long as the corresponding account record exists in persistent storage.
- *Delegate object*—is the Java object that provides the implementation of the `Account` object. It is created by reading the account state from persistent storage (whenever `AccountMgr::openAccount()` is called) and it exists until the server shuts down.
- *Activation state*—is managed by the `ServantLocator`. The `Account` object is activated when a client invokes an operation on the `Account` and deactivated directly afterwards.

Role of the AccountMgr and ServantLocator objects

The AccountMgr object and the ServantLocator object (implemented as AccountServantLocatorImpl) are together responsible for managing the lifecycle of the Account objects.

The main responsibilities of the AccountMgr object are as follows:

- Create an account—the AccountMgr creates a delegate object for the new account and stores the account state in a new persistent record.
- Open an account—the AccountMgr reads the state of the specified account from persistent storage and creates a delegate object for it.

The main responsibilities of the ServantLocator object are, as follows:

- Keep a hash table with references to all of the existing delegate objects.
- Whenever a client makes an invocation on a particular Account object, activate the Account object for the duration of the invocation.
- At the end of an invocation, deactivate the Account object and update the state of the account in persistent storage.

Creating an Account object

Figure 7 shows how the AccountMgr object and the ServantLocator object are involved in the creation of an Account object.

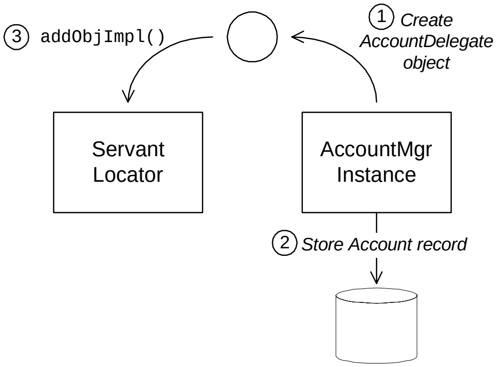


Figure 7: *Creating an Account Object*

In response to an invocation of newAccount(), a new Account is created as follows:

1. An AccountDelegate object (or one of its subclasses) is created and initialized with the data provided in the arguments to newAccount().
2. The new account state is stored in persistent storage (the AccountDelegate object is serialized to disk).
3. The AccountMgr calls addObjImpl() on the servant locator to store the AccountDelegate object in the servant locator's hash table.

Opening an Account object

Figure 8 shows how the AccountMgr object and the ServantLocator object are involved in the opening of an Account object.

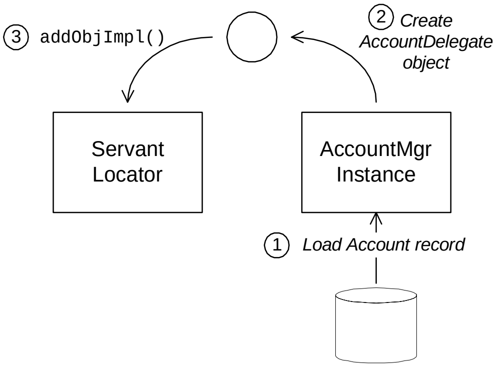


Figure 8: *Opening an Account Object*

In response to an invocation of openAccount(), an Account is loaded into memory as follows:

1. The AccountMgr reads the state of the specified account from persistent storage.
2. An AccountDelegate object is created from the account state (deserialized from disk).
3. The AccountMgr calls addObjImpl() on the servant locator to store the AccountDelegate object in the servant locator's hash table.

Updating an Account object

Figure 9 shows how the `ServantLocator` object updates the state of an account.

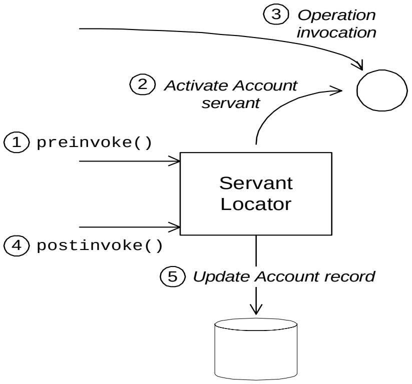


Figure 9: *Updating an Account Object*

Whenever an operation invocation is made on a particular `Account` object, the servant locator reacts as follows:

1. Just prior to the operation invocation, the `BankObjects` POA automatically calls the servant locator’s `preinvoke()` method.
2. The `preinvoke()` method searches for the specified `AccountDelegate` object in the servant locator’s hash table and then activates the `Account` CORBA object.
3. The operation is invoked on the `Account` object.
4. Just after the operation invocation, the `BankObjects` POA automatically calls the servant locator’s `postinvoke()` method.
5. The `postinvoke()` method updates the account state in persistent storage and then deactivates the `Account` object.

Implementing the AccountMgr Interface

Overview

The AccountMgr object is responsible for managing the lifecycle of Account objects (creating and finding). This section describes how the AccountMgr interface is implemented in Java, focusing mainly on the CORBA aspects of the implementation.

In this section

This section contains the following subsections:

Implementing Interfaces Using the Inheritance Approach	page 37
Implementation of the AccountMgr Interface	page 39

Implementing Interfaces Using the Inheritance Approach

Overview

This section discusses how to implement IDL interfaces using the *inheritance approach* (the alternative, delegation approach, is discussed in "Implementing Interfaces Using the Delegation Approach" on page 18).

The inheritance approach is convenient to use, as long as the IDL interface you are implementing does *not* inherit from any other IDL interface (the term *inheritance approach* refers to the use of Java inheritance, not IDL inheritance).

The inheritance approach

In the inheritance approach, the servant for an IDL interface, *InterfaceName*, is represented by a single object, conventionally of *InterfaceName*`Impl` type. The key feature of the inheritance approach is that the implementation class, *InterfaceName*`Impl`, inherits directly from the generated class, *InterfaceName*`POA`.

For example, the `AccountMgrImpl` class is declared as follows:

```
// Java
public class AccountMgrImpl
  extends AccountMgrPOA
  implements Serializable
{
    ...
}
```

Instantiating a servant in the inheritance approach

In the inheritance approach, a servant is instantiated in a single step. For example, the `AccountMgrImpl` servant is instantiated as follows:

```
// Java
package bankobjects;
...
    // Step 1: Create the AccountMgrImpl servant object.
    org.omg.PortableServer.Servant serv  = new AccountMgrImpl(
        ... /* Reference to the servant locator instance */,
        ... /* Reference to the "BankObjects" POA instance */
    );
```

Classes and interfaces needed for the inheritance approach

Figure 10 shows some of the Java classes and interfaces needed for the inheritance approach.

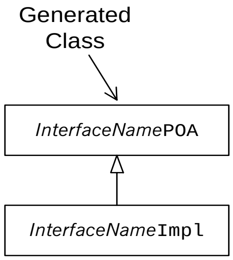


Figure 10: *Classes Needed for the Inheritance Approach*

The *InterfaceName*`Impl` class must extend the *InterfaceName*`POA` Java class. This ensures that *InterfaceName*`Impl` class can be identified as the servant class that implements the *InterfaceName* IDL interface.

Implementing using the inheritance approach

There are two possible starting points for the implementation class:

- *Use the CORBA Code Generation Toolkit*—the code generation toolkit creates an outline of a working application based on an IDL file.

  For example, by generating code from the `Account.idl` file using the `java_poa_genie.tcl` code generation genie you can obtain an initial version of the `AccountImpl` class. See the *CORBA Programmer's Guide* for details of this approach.
- *Use a stub file as a template for the inheritance class*—some of the stub files have a form that is similar to the form required for an inheritance class. These stub files can be copied and modified appropriately to provide the initial versions of your implementation classes.

Note: This approach (stub file as a template) is recommended only for advanced CORBA developers.

Implementation of the AccountMgr Interface

Overview

The AccountMgr IDL interface exposes operations for managing all of the different Account types. In particular, an AccountMgr object is used mainly for creating new accounts or accessing existing accounts.

The implementation of AccountMgr illustrates the inheritance approach—see "Implementing Interfaces Using the Inheritance Approach" on page 37.

Outline of AccountMgr implementation

Example 3 shows an outline of the AccountMgrImpl class, which implements the AccountMgr IDL interface.

Example 3: *Outline of the AccountMgrImpl Class (Sheet 1 of 2)*

```
    // Java
    package bankobjects;

    import org.omg.CORBA.*;
    ...
1   public class AccountMgrImpl
      extends AccountMgrPOA
      implements Serializable
    {
      ...
2     AccountMgrImpl (AccountServantLocatorImpl obj, POA poa)
      {
        m_locatorObj = obj;
        m_poa = poa;
      }

      // Methods that are defined on the IDL interface
3     public Account openAccount ( int accountNumber)
        throws ACCOUNT_DOESNT_EXIST
      {
          ...
      }

      public Account newAccount (String accountType )
      {
          ...
      }

      public void closeAccount (int accountNumber )
```

Example 3: *Outline of the AccountMgrImpl Class (Sheet 2 of 2)*

```
    throws CANNOT_CLOSE_ACCOUNT
  {
      ...
  }

  public int[] getCurrentAccountList()
  {
       ...
  }

  public int[] getCreditCardList()
  {
       ...
  }

4 // Non-IDL, private declarations and methods (not shown)
  ...
}
```

The preceding code can be explained as follows:

1. The `AccountMgrImpl` class extends the following Java class:
   - `bankobjects.AccountMgrPOA`—*required*. The `AccountMgrPOA` Java class declares methods for all the attributes and operations in the `AccountMgr` IDL interface.

   The `AccountMgrImpl` class implements the following Java interface:
   - `java.io.Serializable`—*optional*. Inheriting from the `Serializable` class makes it possible to persist an `AccountMgr` object using the Java serialization technique.
2. The `AccountMgrImpl` constructor caches references to the servant locator and to the `BankObjects` POA instance.

   Note: The cached `BankObjects` POA instance is used to activate `Account` objects, *not* the `AccountMgrImpl` object itself.
3. From this line onwards, all of the `AccountMgr` operations and attributes are defined. If you use the CORBA Code Generation Toolkit to generate the `AccountMgrImpl` class, the method signatures are generated for you and you must only fill in the method bodies.
4. You can also define additional methods, for internal use.

Algorithm for the newAccount() method

To create a new account, the `AccountMgr::newAccount()` method proceeds as follows:

Stage	Description
1	An account number is generated for the new account. The account number is then used as the *object ID* for the corresponding CORBA object.
2	The account delegate object is created and initialized (for example, a `CurrentAccountDelegate` or a `CreditCardAccountDelegate` object, depending on the specified account type).
3	The account state is saved to persistent storage (that is, the delegate object is serialized to disk).
4	The account delegate object is registered with the servant locator. This implies that the delegate object is stored in the servant locator’s hash table and indexed by the account number.
5	An *object reference* is generated for the return value of `newAccount()`. An object reference is an object that encapsulates a CORBA object’s location and gives remote clients access to the CORBA object.

Algorithm for the openAccount() method

To open an existing account, the `AccountMgr::openAccount()` method proceeds as follows:

Stage	Description
1	The method checks whether the account is already in memory by searching the servant locator's hash table.
2	If the account is not found in the hash table, the method searches persistent storage (for example, the file system) to find a record for this account.
3	When the account record is found, the method loads it into memory and creates an account delegate object (that is, deserializes the delegate object from disk).
4	The method registers the account delegate object with the servant locator.
5	The method creates an object reference for the account, which is then passed back to the caller.

Publishing the AccountMgr Object Reference

Object reference

An *object reference* is an object that encapsulates the location and other properties of a CORBA object. It encapsulates all of the information that a CORBA client needs to find and use a CORBA object.

Creating an object reference

The following code listing shows one of the ways to create an object reference for an `AccountMgr` object, using the `PortableServer::POA::create_reference_with_id()` operation.

```
// Java
byte [] oid = "AccountMgrImpl_obj".getBytes();

org.omg.CORBA.Object tmp_ref =
    accountMgrPOA.create_reference_with_id (
        oid,                      // Object ID
        AccountMgrHelper.id()     // Repository (or type) ID
    );
```

In general, the following items are needed to create an object reference:

- A reference to a POA instance—a CORBA object must be associated with a POA. In this example, `accountMgrPOA` references the `AccountManager` POA instance.
- An object ID—together, the POA name and object ID identify the CORBA object uniquely. In this example, the object ID is the string, `AccountMgrImpl_obj`, converted to an array of bytes.
- A repository ID—identifies the object's type. In this example, the value returned by `AccountMgrHelper.id()` is the string, `IDL:bankobjects/AccountMgr:1.0`.

CORBA Naming Service

The back-end server makes the `AccountMgr` object accessible to CORBA clients by publishing the `AccountMgr` object reference to the *CORBA Naming Service*. The CORBA Naming Service is a basic service that stores *name, object reference* associations.

Figure 11 shows how the back-end server publishes object references to the naming service, thereby making them available to CORBA clients.

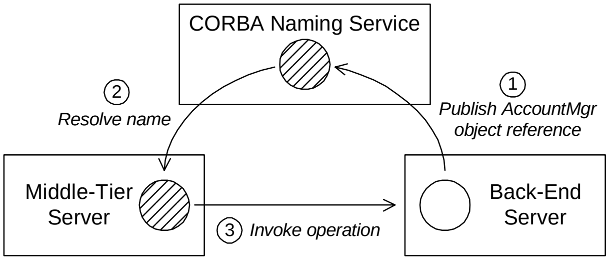


Figure 11: *Publishing an Object Reference in the CORBA Naming Service*

Figure 11 shows the following stages of publishing an object reference:

Stage	Description
1	The back-end server publishes the `AccountMgr` object reference under the name, `BankObjects_AccountMgr`.
2	A client looks up the name, `BankObjects_AccountMgr`, in the naming service and receives the `AccountMgr` object reference in return.
3	The client can now use the `AccountMgr` object reference to make remote invocations on the `AccountMgr` CORBA object.

Example

The back-end server defines a `publish_reference()` method in the `bankobjects.server` class which is used to publish object references to the naming service. Example 4 shows the code for the `publish_reference()` method.

Example 4: *The publish_reference() Method*

```
   // Java
1  import org.omg.CosNaming.*;
   import org.omg.CosNaming.NamingContextPackage.*

     ...
```

Example 4: *The publish_reference() Method*

```
   // In the scope of the ’bankobjects.server’ class
   //
2  static void publish_reference(
                   org.omg.CORBA.Object ref,
                   String refName
               )
   {
     org.omg.CORBA.Object objref = null;
3    NameComponent[] tmpName = new NameComponent[1];

     try
     {
4      objref = orb.resolve_initial_references("NameService");
5      rootContextExt = NamingContextExtHelper.narrow(objref);

6      tmpName[0] = new NameComponent(refName, "");
7      rootContextExt.rebind(tmpName, ref);
     }
     catch (CannotProceed ex) { ... }
     // Catch all relevant exceptions (not shown) ...
     ...
   }
```

The preceding code can be explained as follows:

1. The naming service definitions are contained in the scope of `org.omg.CosNaming`. The `org.omg.CosNaming.NamingContextPackage` subscope contains the definitions of naming service exceptions.
2. The `publish_reference()` method takes two arguments: the object reference to be published, `ref`, and the name under which the object reference will be published, `refName`.
3. Technically, a name in the naming service is an IDL sequence of name components (of `org.omg.CosNaming.NameComponent[]` type in Java). For simplicity, this example creates an array with just a single name component, `tmpName`.
4. An initial reference to the naming service is obtained from the ORB by calling `resolve_initial_references()` with the string argument, `NameService`. This is the standard way of connecting to the naming service.

5. The reference returned from `resolve_initial_references()`, of `org.omg.CORBA.Object` type, is cast to the type, `org.omg.CosNaming.NamingContextExt`. The `NamingContextExt` object, `rootContextExt`, provides access to the naming service functionality.
6. The name component array, `tmpName`, is initialized using the `refName` string.
7. Invoking `rebind()` on the root naming context creates a *name, object reference* association between `tmpName` and `ref` in the naming service.

CHAPTER 2

Middle-Tier CORBA Server

This chapter discusses the design and implementation of the middle-tier CORBA server. Starting from a high-level design, the business session interfaces are defined in the OMG interface definition language (IDL) and then implemented in Java.

In this chapter

This chapter discusses the following topics:

Design of the Middle-Tier Server	page 48
IDL for the Middle-Tier Server	page 51
Designing the POA Hierarchy	page 55
Resolving the AccountMgr Object Reference	page 57
Implementing the BusinessSession Interfaces	page 60

Design of the Middle-Tier Server

Purpose of the middle-tier server

The purpose of the middle-tier server is to mediate between the back-end server and a variety of different client types. The middle-tier server provides support for session management and imposes constraints on what clients can and cannot do.

CORBA object types

Figure 12 shows the inheritance hierarchy for the object types implemented in the middle-tier server.

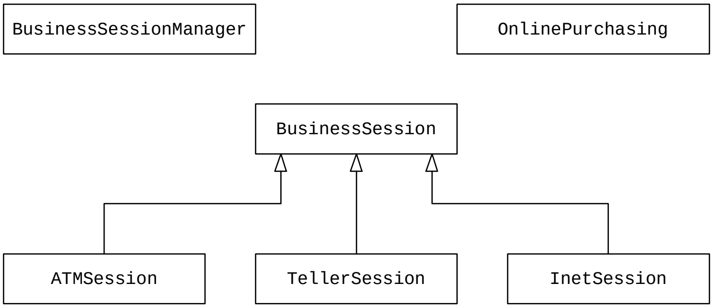


Figure 12: *Inheritance Hierarchy for BusinessSession Types*

BusinessSessionManager type

A single object of `BusinessSessionManager` type is provided to open and close client sessions. Because client sessions are represented here by `BusinessSession` objects, the `BusinessSessionManager` acts as a *factory* for `BusinessSession` objects. The following operations are provided:

- `openSession()`—open a new client session of the specified type and return a reference to a `BusinessSession` object.
- `closeSession()`—close the specified client session, releasing all of the associated resources.

BuesinessSession type

The BusinessSession type is an abstract base class for the other session types. One attribute is defined on the BusinessSession class:

- Session ID.

Some methods are also defined on the BusinessSession class, as follows:

- Resolve account—finds a specified account and associates it with the current session.
- Get information on the account currently associated with the session.
- Get a list of accounts of a particular type.

ATMSession type

The ATMSession type inherits from BusinessSession and adds methods to support the following functionality:

- Validate the customer's PIN against the currently active account.
- Check the daily limit on withdrawal amounts.
- Give the ATM authorization to dispense the cash.
- Receive confirmation that cash was dispensed.

InetSession type

The InetSession type inherits from BusinessSession and adds no methods or attributes.

TellerSession type

The TellerSession type inherits from BusinessSession and adds methods to support the following functionality:

- Create a new account and associate the account with the current client session.
- Access an existing account and associate the account with the current client session.
- Deposit and withdraw cash.
- Transfer money in or out of the account.
- Check the balance on the account.

OnlinePurchasing type

The `OnlinePurchasing` interface is designed to support retailers, or *merchants*, who sell goods over the Internet. Registered merchants are allowed to debit credit cards, transferring money from a customer's credit card account into the merchant's own account. In this way, merchants can sell goods over the Internet which are paid for by credit card.

The following operations are provided:

- `registerMerchant()`—the merchant uses this operation to log on to the online purchasing system, receiving a merchant ID in return.
- `makePurchase()`—this operation is called when a customer purchases an item from the merchant. An amount of money is debited from the customer's account (identified by the credit card details) and credited to the merchant's account (identified by the merchant ID).
- `listMerchants()`—returns a list of all of the merchants that are currently registered.
- `lookupMerchant()`—returns the account details for a particular merchant ID.

IDL for the Middle-Tier Server

BusinessSessionManager IDL

The code listing in Example 5 shows the IDL for the middle-tier server, `idl/BusinessSessionManager.idl`.

Example 5: *The BusinessSessionManager.idl File (Sheet 1 of 4)*

```
   // IDL
   #ifndef BUSINESSSESSIONMANAGER_IDL
   #define BUSINESSSESSIONMANAGER_IDL

1  #include "Account.idl"

   module fnbba {
       // Exceptions
       exception AUTHORIZE_FAILED {} ;
       exception NO_RESOURCES {};
       exception ACCOUNT_DOESNT_EXIST {};
       exception NO_OPEN_SESSION {};
       exception NO_SUCH_ACCOUNT {};
       exception NO_SUCH_MERCHANT {};
       exception INSUFFICIENT_FUNDS {};

       // Structures
       struct SessionInfo_s {
           string username;
           string password;
           string session_type;
           string client_id;
       };

2      enum transtype {
         LODGEMENT,
         WITHDRAWAL,
         TRANSFER_IN,
         TRANSFER_OUT,
         ACCOUNT_OPENED,
         PURCHASE,
         OTHER
       };

       // A structure to associate a transaction and a description
       struct transaction_s {
```

Example 5: *The BusinessSessionManager.idl File (Sheet 2 of 4)*

```
      string date;
      string description;
      transtype transactionType;
      string value;
    };

    // A sequence of transaction details
    typedef sequence<transaction_s> transList;

    struct AccountInfo_s
    {
       string lname;
       string fname;
       string accType;
       string addr1;
       string addr2;
       string addr3;
       float limit;

       transList transactions;
    };

    // Typedefs
    typedef SessionInfo_s SessionInfo;
    typedef AccountInfo_s AccountInfo;

    interface BusinessSession {
        readonly attribute short session_id;

3       bankobjects::Account resolveAccount (in accountNum acct)
            raises (NO_SUCH_ACCOUNT);
        AccountInfo_s getAccountInfo ();
        accountNumList getAccountList (in string accountType);
    };

    interface BusinessSessionManager {
4       BusinessSession openSession (inout SessionInfo usi)
            raises( NO_RESOURCES );
        void closeSession (in BusinessSession bs);
    };

    interface ATMSession : BusinessSession {
        void validateCard (in short pin )
            raises( AUTHORIZE_FAILED );
        void checkLimits (
```

Example 5: *The BusinessSessionManager.idl File (Sheet 3 of 4)*

```
            out short dailyLimit,
            out short alreadyWithdrawn
        );
        boolean okToDispense (in short amount);
        void dispensedCash (in short amount);
    };

    interface InetSession : BusinessSession {
    };

    interface TellerSession: BusinessSession {
        accountNum newAccount (in AccountInfo accDetails);
        void openAccount (in accountNum acctNum)
          raises (NO_SUCH_ACCOUNT);
        boolean lodgeFunds (in float amnt);
        boolean withdrawFunds (in float amnt)
          raises (INSUFFICIENT_FUNDS);
        boolean transferFunds (in float amnt, in accountNum acct)
          raises (NO_SUCH_ACCOUNT, INSUFFICIENT_FUNDS);

        float accBalance ();
    };

    typedef string MerchantIdentifier;

    struct Merchant {
        bankobjects::accountNum acct;
        MerchantIdentifier merchantID;
    };

    typedef sequence<Merchant> Merchants;

    interface OnlinePurchasing {
        MerchantIdentifier registerMerchant(
            in bankobjects::accountNum acct
        ) raises (NO_SUCH_ACCOUNT);
        string makePurchase(
            in MerchantIdentifier merchantID,
            in string cardNum, in string expiryDate,
            in string securityCode,
            in float amount
        ) raises (NO_SUCH_MERCHANT, INSUFFICIENT_FUNDS);
        Merchants listMerchants();
        bankobjects::Account lookupMerchant(
            in MerchantIdentifier merchantID
```

Example 5: *The BusinessSessionManager.idl File (Sheet 4 of 4)*

```
        )
        raises (NO_SUCH_MERCHANT);
    };
};  // Module  fnbba

#endif //BUSINESSSESSIONMANAGER_IDL
```

The preceding code can be explained as follows:

1. The #include directive brings in all of the definitions from `Account.idl`. See Example 1 on page 7.
2. The IDL `enum` type is similar to the C and C++ `enum` type, except that you cannot assign integer values to the `enum` labels. In Java, an IDL `enum` maps to a Java class with constant members defined for each label.

   For example, in Java the `fnbba::transtype::LODGEMENT` IDL value is mapped to an `fnbba.transtype.LODGEMENT` constant (of `fnbba.transtype` type), and an `fnbba.transtype._LODGEMENT` constant (of `int` type).
3. Because the `Account` type is defined outside the scope of the `fnbba` module, it is necessary to use the fully-scoped name here, `bankobjects::Account`.
4. The `SessionInfo` parameter of `openSession()` is declared to be an `inout` parameter. During an operation invocation, the `inout` parameter travels in both directions: from client to server, and back from server to client. It is possible for the server to modify the `inout` parameter before sending it back to the client.

Designing the POA Hierarchy

Overview

This section describes the POA hierarchy for the middle-tier server and how it affects the life cycle of the various CORBA objects. For more details about the POA, see “Designing the POA Hierarchy” on page 14 and the *CORBA Programmer’s Guide*.

POA hierarchy for the back-end server

Figure 13 shows the POA hierarchy used for the middle-tier server. The root POA has two children:

- A POA for managing `BusinessSessionManager` objects (named `BusinessSessionManager`), and
- A POA for managing `BusinessSession` objects (named `BusinessSession`).

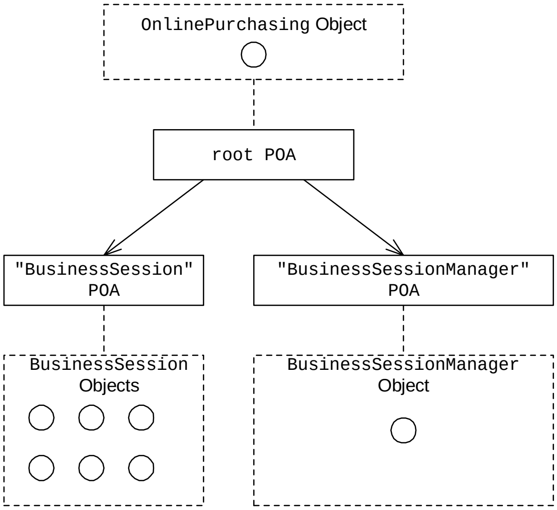


Figure 13: *POA Hierarchy for the Middle-Tier Server*

BusinessSessionManager POA

The `BusinessSessionManager` POA is created to manage the `BusinessSessionManager` object (only one object of `BusinessSessionManager` type is ever created). The `BusinessSessionManager` object is created and activated when the application starts up and remains active for as long as the application is running.

Because the life cycle of the `BusinessSessionManager` object is fairly simple, the associated POA has straightforward policies. Some of the policies that are explicitly set on the `BusinessSessionManager` POA are the following:

- The `LifespanPolicy`—is defined to be `PERSISTENT`.
- The `ThreadPolicy`—is defined to be single threaded, `SINGLE_THREAD_MODEL`.

BusinessSession POA

The `BusinessSession` POA is created to manage `BusinessSession` objects. The POA is created with the default POA policies.

In a highly-scalable system, you would probably require a more sophisticated way of managing the session life cycle than the approach used in this demonstration. For example, you might want to impose a time-out on client sessions, so that a session is automatically deleted if it remains idle for a specified period of time. This sort of functionality is supported by the CORBA Session Management Plug-In. See the *CORBA Session Management Guide* for details.

Resolving the AccountMgr Object Reference

Overview

The middle-tier server initially gains access to the back-end server by retrieving an `AccountMgr` object reference from the naming service. The `AccountMgr` object in the back-end is thus the initial point of contact for the middle-tier.

CORBA Naming Service

The middle-tier server *resolves* the name of the `AccountMgr` object reference previously published to the CORBA Naming Service by the back-end server (see "Publishing the AccountMgr Object Reference" on page 43). The return value of the resolve operation is an `AccountMgr` object reference.

Figure 14 shows how the middle-tier server resolves a published `AccountMgr` object reference.

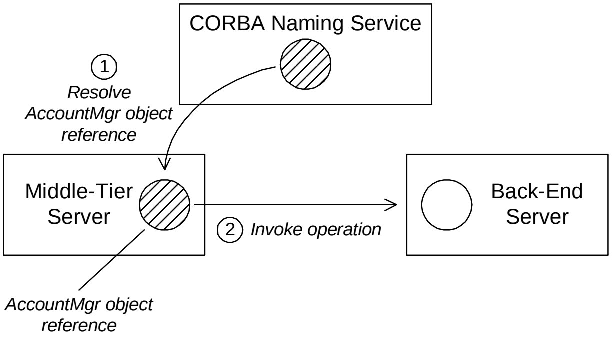


Figure 14: *Resolving the AccountMgr Object Reference*

Example

The middle-tier server defines a `resolveObject()` method in the `fnbba.server` class which is used to resolve object references from the naming service. Example 6 shows the code for the `resolveObject()` method.

Example 6: *The resolveObject() Method*

```
    // Java
1     public static org.omg.CORBA.Object resolveObject(
                        String refName
                    )
        throws Exception
      {
        org.omg.CORBA.Object tmpObj1 = null;
2       NameComponent[] tmpName = new NameComponent[1];
        try
        {
3         objref = orb.resolve_initial_references("NameService");
4         rootContextExt = NamingContextExtHelper.narrow(objref);
5         tmpName[0] = new NameComponent(refName, "");
6         tmpObj1 = rootContextExt.resolve(tmpName);
        }
        catch (CannotProceed ex) { ... }
        // Catch all the different exceptions (not shown) ...
        ...
        return tmpObj1;
      }
```

The preceding code can be explained as follows:

1. The `resolveObject()` method takes a string name, `refName`, as an argument and returns the corresponding object reference that it finds in the naming service.
2. A name in the naming service is an IDL sequence of name components (of `org.omg.CosNaming.NameComponent[]` type in Java). For simplicity, this example creates an array with just a single name component, `tmpName`.
3. An initial reference to the naming service is obtained from the ORB by calling `resolve_initial_references()` with the string argument, `NameService`. This is the standard way of connecting to the naming service.

4. The reference returned from `resolve_initial_references()`, of `org.omg.CORBA.Object` type, is cast to the type, `org.omg.CosNaming.NamingContextExt`. The `NamingContextExt` object, `rootContextExt`, provides access to the naming service functionality.
5. The name component array, `tmpName`, is initialized using the `refName` string.
6. This line invokes `resolve()` on the root naming context, thereby looking up the name, `tmpName`, in the naming service to get an object reference, `tmpObj1`, in return.

Implementing the BusinessSession Interfaces

Overview

The BusinessSession interfaces are all implemented using the delegate (or TIE) approach—see "Implementing Interfaces Using the Delegation Approach" on page 18. The session interfaces are implemented as follows:

- BusinessSession—implemented by BusinessSessionDelegate.
- ATMSession—implemented by ATMSessionDelegate.
- TellerSession—implemented by TellerSessionDelegate.

Inheritance hierarchy for the implementation classes

Figure 15 shows the inheritance hierarchy for the delegate objects that implement the various session types.

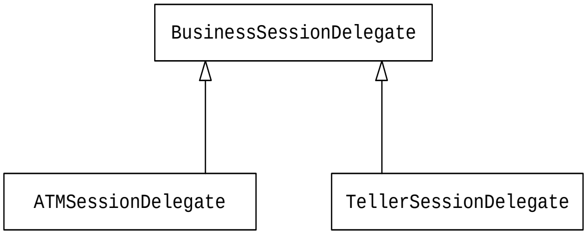


Figure 15: *Java Inheritance Hierarchy for the Delegate Objects*

The BusinessSessionDelegate Class

Example 7 shows an outline of the BusinessSessionDelegate class, which implements the BusinessSession IDL interface.

Example 7: *Outline of the BusinessSessionDelegate Class*

```
// Java
package fnbba;

1 class BusinessSessionDelegate
    implements BusinessSessionOperations
  {
    protected bankobjects.AccountMgr myMgr = null;
    protected bankobjects.Account myAccount = null;
```

Example 7: *Outline of the BusinessSessionDelegate Class*

```
2   public BusinessSessionDelegate ( )
    {
      // Initialize the myMgr member variable by resolving
      // the 'AccountMgr' object in the naming service.
      ...
    }

3   // IDL Operation Implementations
    public short session_id ()
    {
      ...
    }

    public bankobjects.Account resolveAccount (int accountNum)
      throws NO_SUCH_ACCOUNT
    {
      ...
    }

    public float accBalance ()
    {
      ...
    }

    public AccountInfo_s getAccountInfo ()
    {
      ...
    }

    public int [] getAccountList (String accountType)
    {
      ...
    }
  }
```

The preceding code can be explained as follows:

1. The `BusinessSessionDelegate` class implements the following Java interface:
   - `fnbba.BusinessSessionOperations`—*required*. The `BusinessSessionOperations` Java interface declares methods for all the attributes and operations in the `BusinessSession` IDL interface.

2. The constructor uses the naming service to get an object reference for the back-end server's `AccountMgr` object, which is then cached in the `myMgr` member variable.
3. From this line onward, each of the IDL operations are implemented.

CHAPTER 3

Java CORBA Client

The Java CORBA client is implemented as a graphical user interface (GUI). This chapter presents the design of the CORBA client and describes how the client accesses the business logic in the middle-tier using CORBA remote operation invocations.

In this chapter

This chapter discusses the following topics:

Design of the CORBA Client	page 64
Using Forte for Java and NetBeans	page 72
Resolving the BusinessSessionManager Object Reference	page 77
Implementation of the Java CORBA Client	page 81
Implementation of the Open Account Dialog	page 83

Design of the CORBA Client

Purpose of the client

The CORBA client is implemented as a graphical user interface (GUI) that provides the banking services you would normally expect from a human bank teller. For example, the CORBA client might used by a bank teller when dealing with customers at the counter.

The following banking services are supported by the CORBA client:

- Creating a new account.
- Accessing an existing account to check the balance and view recent transactions.
- Performing a variety of transactions: lodging funds, withdrawing money, and transferring funds from one account to another.

Organization of screens

The client GUI is organized as a set of screens that support different functions:

- Main screen—this is the initial screen for the client application.

From the main screen, you can access a set of dialog windows:

- Open account dialog.
- New account dialog.
- Lodge funds dialog.
- Withdraw funds dialog.
- Transfer funds dialog.

Main screen

The main screen displays the details for the currently open account (only one account can be open at a time). The **Account** menu on the main screen provides access to each of the dialog windows, as shown in Figure 16.

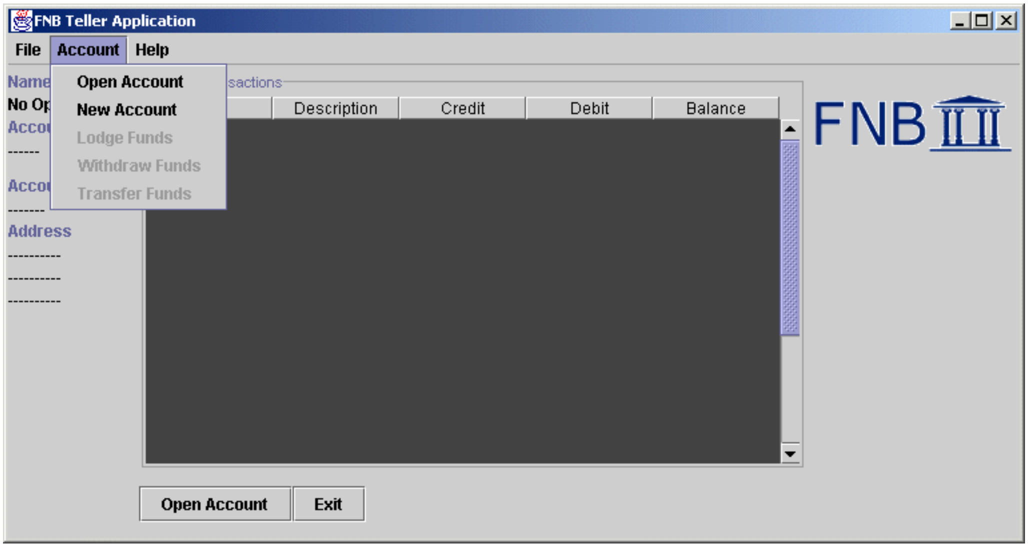


Figure 16: *The Main Screen of the Java CORBA Client*

In this section

This section describes each of the dialog windows:

The Open Account Dialog	page 66
The New Account Dialog	page 67
The Lodge Funds Dialog	page 69
The Withdraw Funds Dialog	page 70
The Transfer Funds Dialog	page 71

The Open Account Dialog

Dialog window

The teller uses the open account dialog window, Figure 17, to set the currently active account. The details of this account are then displayed in the main screen.

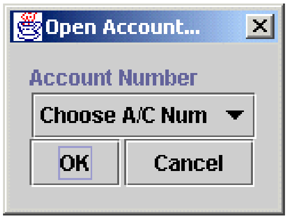


Figure 17: *The Open Account Dialog Window*

Data required to initialize the dialog

The following data is required to initialize the open account dialog:

- A list of account numbers for all the accounts stored in the back-end server—the list is displayed when the user clicks on the **Choose A/C Num** drop-down menu.

Data returned by the dialog

The data returned by the open account dialog depends on the event that closes the dialog window:

- Click on **OK**—the selected account number is returned.
- Click on **Cancel**—no data is returned.

Associated files

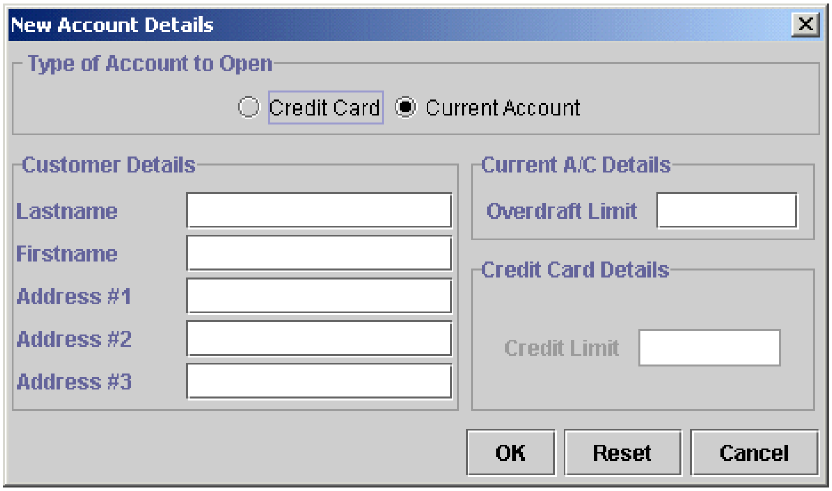
The following files are associated with the open account dialog implementation:

```
gui/openAccount.java
gui/openAccount.form
```

The New Account Dialog

Dialog window

The teller uses the new account dialog window, Figure 18, to create a new account. The details of this account are then displayed in the main screen.



New Account Details

Type of Account to Open

○ Credit Card ● Current Account

Customer Details

Lastname

Firstname

Address #1

Address #2

Address #3

Current A/C Details

Overdraft Limit

Credit Card Details

Credit Limit

Figure 18: *The New Account Dialog Window*

Data required to initialize the dialog

No data is required to initialize the new account dialog.

Data returned by the dialog

The data returned by the new account dialog depends on the event that closes the dialog window:

- Click on **OK**, with **Current Account** selected—the following data is returned:
    - The **Lastname** and **Firstname** of the new account owner.
    - The account owner's address, in the **Address #1**, **Address #2**, and **Address #3** fields.
    - The amount of the **Overdraft Limit** on the current account.
- Click on **OK**, with **Credit Card** selected—the following data is returned:
    - The **Lastname** and **Firstname** of the new account owner.
    - The account owner's address, in the **Address #1**, **Address #2**, and **Address #3** fields.
    - The amount of the **Credit Limit** on the credit card.
- Click on **Cancel**—no data is returned.

Associated files

The following files are associated with the new account dialog implementation:

```
gui/newAccount.java
gui/newAccount.form
```

The Lodge Funds Dialog

Dialog window

The teller uses the lodge funds dialog window, Figure 19, to lodge an amount of money into the currently active account.

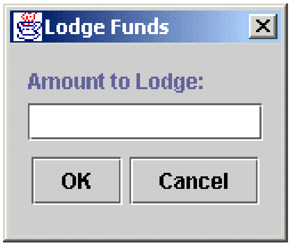


Figure 19: *The Lodge Funds Dialog Window*

Data required to initialize the dialog

No data is required to initialize the lodge funds dialog.

Data returned by the dialog

The data returned by the lodge funds dialog depends on the event that closes the dialog window:

- Click on **OK**—the lodgement amount is returned.
- Click on **Cancel**—no data is returned.

Associated files

The following files are associated with the lodge funds dialog implementation:

```
gui/lodgeFunds.java
gui/lodgeFunds.form
```

The Withdraw Funds Dialog

Dialog window

The teller uses the withdraw funds dialog window, Figure 20, to withdraw cash from the currently active account.

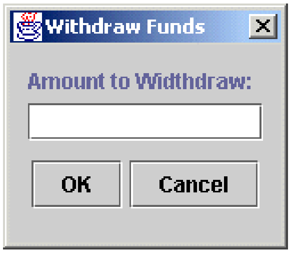


Figure 20: *The Withdraw Funds Dialog Window*

Data required to initialize the dialog

No data is required to initialize the withdraw funds dialog.

Data returned by the dialog

The data returned by the withdraw funds dialog depends on the event that closes the dialog window:

- Click on **OK**—the withdrawal amount is returned.
- Click on **Cancel**—no data is returned.

Associated files

The following files are associated with the withdraw funds dialog implementation:

```
gui/withdrawFunds.java
gui/withdrawFunds.form
```

The Transfer Funds Dialog

Dialog window

The teller uses the transfer funds dialog window, Figure 21, to transfer money from the currently active account to another account.

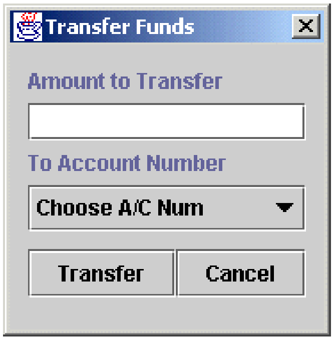


Figure 21: *The Transfer Funds Dialog Window*

Data required to initialize the dialog

The following data is required to initialize the transfer funds dialog:

- A list of account numbers for all the accounts stored in the back-end server—the list is displayed when the user clicks on the **Choose A/C Num** drop-down menu.

Data returned by the dialog

The data returned by the transfer funds dialog depends on the event that closes the dialog window:

- Click on **Transfer**—the transfer amount and the selected account number are returned.
- Click on **Cancel**—no data is returned.

Associated files

The following files are associated with the transfer funds dialog implementation:

```
gui/transferFunds.java
gui/transferFunds.form
```

Using Forte for Java and NetBeans

Overview

The graphical elements of the Java CORBA client are implemented using Sun's Forte for Java.

NetBeans

NetBeans is an open source integrated development environment (IDE) for building client-side and server-side applications. Because the NetBeans IDE is based on an extensible, modular framework, third parties can also provide customized distributions of NetBeans based on the *NetBeans Tools Platform*. Hence, the following varieties of NetBeans-based products are available:

- *NetBeans IDE*—the original open source IDE, which can be downloaded directly from the NetBeans web site, http://www.netbeans.org.
- *Third-party IDEs, based on the NetBeans Tools Platform*—other organizations and vendors can add their own modules to the NetBeans core and then release enhanced versions of the IDE. Sun's Forte for Java is an example of such a third-party IDE.

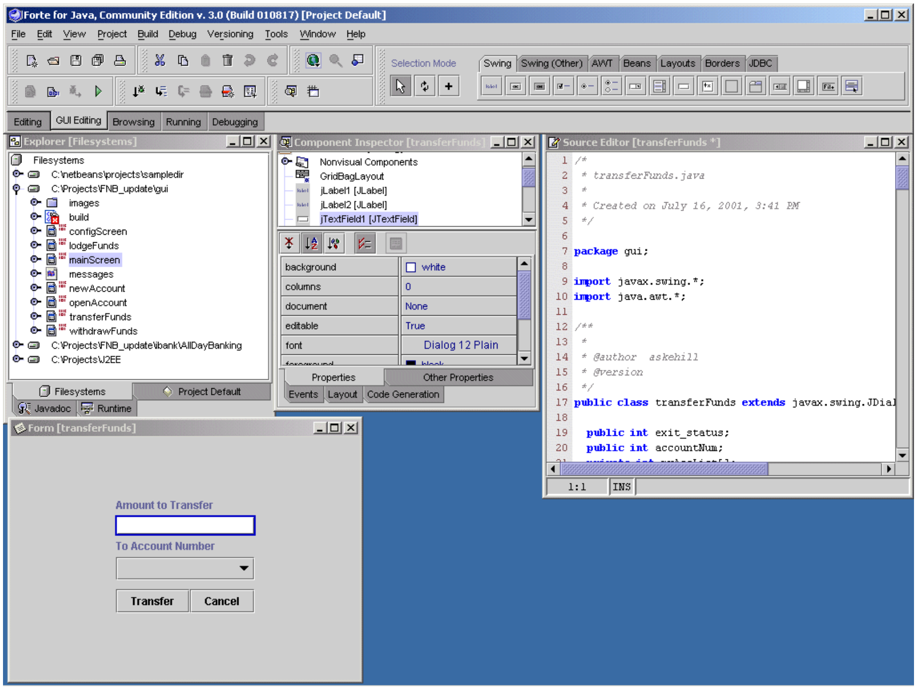
Forte for Java

Forte for Java is Sun's extensible, integrated development environment (IDE) for Java Technology developers. It is based on the NetBeans Tools Platform and is integrated with the Sun Open Net Environment (ONE).

Opening a Java source file using Forte for Java

If you have Forte for Java installed, you can use it to view the CORBA client source files. Start up the Forte for Java IDE, and then use the **File|Open** menu option to open one of the Java source files.

For example, Figure 22 shows the screen layout of the Forte for Java IDE after opening the `transferFunds.java` file.


Forte for Java, Community Edition v. 3.0 (Build 010817) [Project Default]
File Edit View Project Build Debug Versioning Tools Window Help
Selection Mode
Swing Swing (Other) AWT Beans Layouts Borders JDBC
Editing GUI Editing Browsing Running Debugging
Explorer [Filesystems]
Filesystems
C:\netbeans\projects\sampledir
C:\Projects\FNB_update\gui
images
build
configScreen
lodgeFunds
mainScreen
messages
newAccount
openAccount
transferFunds
withdrawFunds
C:\Projects\FNB_update\ibank\AllDayBanking
C:\Projects\J2EE
Filesystems Project Default
Javadoc Runtime
Component Inspector [transferFunds]
Nonvisual Components
GridBagLayout
jLabel1 [JLabel]
jLabel2 [JLabel]
jTextField1 [JTextField]
Properties Other Properties
Events Layout Code Generation
Source Editor [transferFunds *]
1:1 INS
Form [transferFunds]
Amount to Transfer
To Account Number
Transfer Cancel

background	□ white
columns	0
document	None
editable	True
font	Dialog 12 Plain

```
/*
 * transferFunds.java
 *
 * Created on July 16, 2001, 3:41 PM
 */

package gui;

import javax.swing.*;
import java.awt.*;

/**
 *
 * @author  askehill
 * @version
 */
public class transferFunds extends javax.swing.JDial

  public int exit_status;
  public int accountNum;
```

Figure 22: *Editing the Transfer Funds Dialog within the Forte for Java IDE*

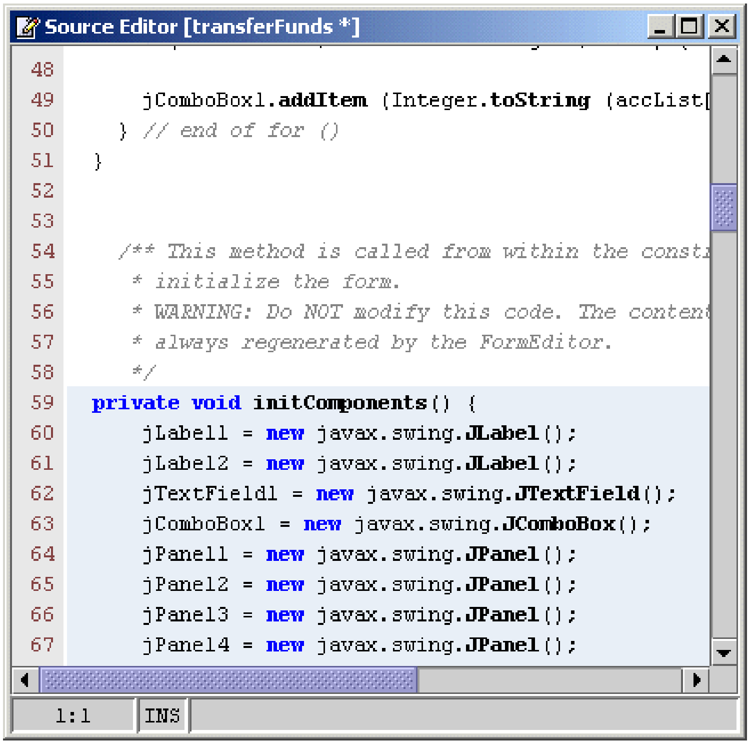
Forte for Java screen layout

As soon as you open the source file for a GUI form, the Forte for Java editor automatically switches to the **GUI Editing** view, as in Figure 22. In this view, the following windows are visible:

- The main Forte for Java window (top).
- The **Explorer** window (midway up the left-hand side)—provides a view of the file system.
- The **Form** window (bottom left)—shows the layout of the form that is currently being edited.
- The **Source Editor** window (midway up the right-hand side)—shows the Java source code for the form.
- The **Component Inspector** window (center)—shows the properties of the component currently selected in the **Form** window. The selected component is highlighted by a blue-colored border.

Forte for Java generated code

Figure 23 shows part of the code listing from the `transferFunds.java` file, as viewed in the Forte for Java **Source Editor** window.



```

        jComboBox1.addItem (Integer.toString (accList[
      } // end of for ()
    }


      /** This method is called from within the const
       * initialize the form.
       * WARNING: Do NOT modify this code. The content
       * always regenerated by the FormEditor.
       */
    private void initComponents() {
        jLabel1 = new javax.swing.JLabel();
        jLabel2 = new javax.swing.JLabel();
        jTextField1 = new javax.swing.JTextField();
        jComboBox1 = new javax.swing.JComboBox();
        jPanel1 = new javax.swing.JPanel();
        jPanel2 = new javax.swing.JPanel();
        jPanel3 = new javax.swing.JPanel();
        jPanel4 = new javax.swing.JPanel();
```

Figure 23: *Viewing the transferFunds.java file in the Source Editor*

The source editor window makes it easy to distinguish between the generated code, on a shaded background, and the hand-written code, on a white background.

In particular, the `initComponents()` method, shown in Figure 23, is responsible for initializing the layout of the window and is wholly generated by the IDE.

References

For more details about the NetBeans IDE, see the following page:

http://www.netbeans.org/intro.html

For more details about the Forte for Java IDE, see the following page:

http://www.sun.com/forte/ffj/

Resolving the BusinessSessionManager Object Reference

Overview

The Java CORBA client gains access to the middle-tier server by retrieving a `BusinessSessionManager` object reference from the naming service. The client can then open a business session by calling the `openSession()` operation on the `BusinessSessionManager` object.

The middle-tier server provides all of the services needed by the client; the client does not contact the back-end directly.

CORBA Naming Service

Figure 24 shows how the CORBA client resolves the `BusinessSessionManager` object reference from the naming service.

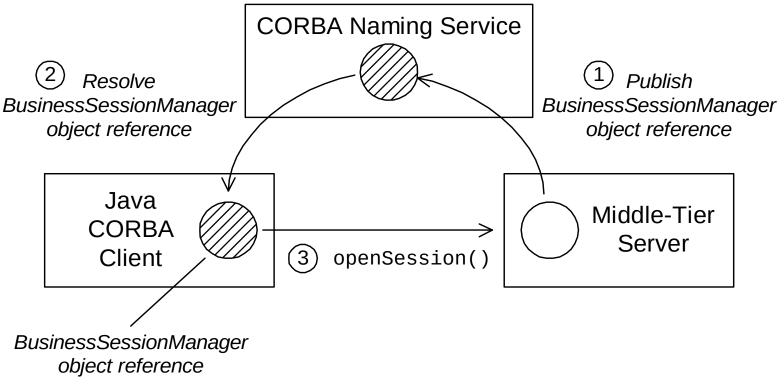

CORBA Naming Service
2 Resolve BusinessSessionManager object reference
1 Publish BusinessSessionManager object reference
Java CORBA Client
3 openSession()
Middle-Tier Server
BusinessSessionManager object reference

Figure 24: *Establishing a Connection to the Middle-Tier Server*

The connection between the Java CORBA client and the middle-tier is established as follows:

Stage	Description
1	The middle-tier server publishes the `BusinessSessionManager` object reference under the name, `FNBBA_BusinessSessionManager`.
2	The Java CORBA client looks up the name, `FNBBA_BusinessSessionManager`, in the naming service and receives the `BusinessSessionManager` object reference in return.
3	The client can now invoke the `openSession()` operation on the `BusinessSessionManager` object reference to open a new business session.

Example

The Java CORBA client defines a `resolveServer()` method (in the `gui.mainScreen` class) which resolves the middle-tier `BusinessSessionManager` object reference. Example 8 shows the code for the `resolveServer()` method.

Example 8: *The resolveServer() Method*

```
// Java
...
    // In the 'mainScreen' Class
    //
    public static void resolveServer (String[] args)
    {
      org.omg.CORBA.Object objref = null;

      try {
        global_orb = ORB.init(args, null);
1       objref = global_orb.resolve_initial_references (
                     "NameService"
                 );
2       rootContextExt = NamingContextExtHelper.narrow (objref);
      }
      catch ( ... ) { ... }
      // Handle all exceptions (not shown)...
```

Example 8: *The resolveServer() Method*

```
        org.omg.CORBA.Object tmpObj1 = null;
3       NameComponent[] tmpName = new NameComponent[1];
        try {
4         tmpName[0] = new NameComponent(
                                 "FNBBA_BusinessSessionManager", ""
                             );
5         tmpObj1 = rootContextExt.resolve(tmpName);
6         sessionMgr = fnbba.BusinessSessionManagerHelper.narrow (
                             tmpObj1
                         );
        }
        catch ( ... ) { ... }
        // Handle all exceptions (not shown)...

        return;
      }
```

The preceding code can be explained as follows:

1. An initial reference to the naming service is obtained from the ORB by calling `resolve_initial_references()` with the string argument, `NameService`. This is the standard way of connecting to the naming service.
2. The reference returned from `resolve_initial_references()`, of `org.omg.CORBA.Object` type, is cast to the type, `org.omg.CosNaming.NamingContextExt`. The `NamingContextExt` object, `rootContextExt`, provides access to the naming service functionality.
3. Create a name, `tmpName`, with just a single name component (of `org.omg.CosNaming.NameComponent[1]` type).
4. The name component array, `tmpName`, is initialized with the string, `FNBBA_BusinessSessionManager`.
5. This line invokes `resolve()` on the root naming context, thereby looking up the name, `tmpName`, in the naming service to get an object reference, `tmpObj1`, in return.

6. The reference returned from `resolve()` is cast to the type, `fnbba.BusinessSessionManager`, using a `narrow()` method. The `narrow()` method defined on `BusinessSessionManagerHelper` provides a type-safe way of down-casting the returned object reference to the `BusinessSessionManager` type.

Implementation of the Java CORBA Client

Overview

The client is implemented using six classes: one for the main screen, `gui.mainScreen`, and one for each of the five dialog windows. Since most of the CORBA code is contained in `mainScreen.java`, this section focuses on the implementation of the `mainScreen` class.

Organization of the client code

Figure 25 illustrates the relationship between the main screen and a dialog window. Each of the dialog windows is treated as a black box that returns information from a user.

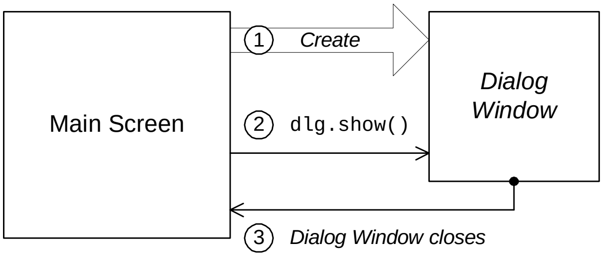


Figure 25: *Relationship Between the Main Screen and a Dialog Window*

The general pattern of interaction between the `mainScreen` class and a dialog window is as follows:

Stage	Description
1	The `mainScreen` class creates a dialog object, `dlg`, and passes initial data to the dialog.
2	The `mainScreen` class passes control to the dialog object by calling `dlg.show()`.
3	When the dialog window closes, the `mainScreen` class extracts the information from the dialog that was set by the user.

Implementation of the mainScreen class

Because the `mainScreen` class is created using the Forte for Java IDE, there are chunks of generated code in the listing that are not meant to be edited by the developer. In particular, you can ignore the `initComponents()` method.

The following methods of the `mainScreen` class are of interest here:

- `main()`—the entry point for the client application.
- `resolveServer()`—bootstraps a connection to the middle-tier server by retrieving a `BusinessSessionManager` object reference from the naming service (see "Resolving the BusinessSessionManager Object Reference" on page 77). Called from `main()`.
- `openAccountActionPerformed()`—launches the **Open Account** dialog window and opens an account.
- `newAccountActionPerformed()`—launches the **New Account** dialog window and creates a new account.
- `lodgeFundsActionPerformed()`—launches the **Lodge Funds** dialog window and lodges an amount into the currently active account.
- `withdrawFundsActionPerformed()`—launches the **Withdraw Funds** dialog window and withdraws an amount from the currently active account.
- `transferFundsActionPerformed()`—launches the **Transfer Funds** dialog window and transfers an amount from the currently active account to a specified account.

Implementation of the Open Account Dialog

Overview

To illustrate how the dialog screens work, this section describes how the `mainScreen` class interacts with the **Open Account** dialog. The `mainScreen` class uses the data from the dialog to open a session with an account object in the back-end.

From the main screen there are two ways of initiating the open account dialog:

- Select **Account|Open Account**—this calls the `openAccountActionPerformed()` method.
- Click the **Open Account** button—this calls the `openAccountButtonActionPerformed()` method.

This section describes the `openAccountActionPerformed()` method. The `openAccountButtonActionPerformed()` method has an essentially identical implementation.

Code for openAccountActionPerformed()

Example 9 shows the Java code for the `openAccountActionPerformed()` method.

Example 9: *The openAccountActionPerformed() Method (Sheet 1 of 3)*

```
   // Java
     ...
1    private void
      openAccountActionPerformed(java.awt.event.ActionEvent evt)
     {//GEN-FIRST:event_openAccountActionPerformed
2      genericOpenAccount (evt);

       lodgeFunds.setEnabled(true);
       withdrawFunds.setEnabled (true);
       transferFunds.setEnabled(true);
     }//GEN-LAST:event_openAccountActionPerformed
     ...

3    private void genericOpenAccount (
                      java.awt.event.ActionEvent evt
     )
     {
```

Example 9: *The openAccountActionPerformed() Method (Sheet 2 of 3)*

```
    // Get a session back from the FNB Core
4   fnbba.SessionInfo_s sessionInfo = new fnbba.SessionInfo_s ();

    // Hardcoded  values for now...
    sessionInfo.username = new String ("Adrian");
    sessionInfo.password = new String ("pass001");
    sessionInfo.session_type = new String ("Teller");
    sessionInfo.client_id = new String ("Teller-012");

    fnbba.SessionInfo_sHolder sesHold
      = new fnbba.SessionInfo_sHolder(sessionInfo);
    fnbba.BusinessSession sess = null;
    try {
5     sess = sessionMgr.openSession ( sesHold );
    }
    catch ( ... ) { ... }
    // Handle all exceptions (not shown)...

    fnbba.TellerSession tSession
      = fnbba.TellerSessionHelper.narrow (sess);

6   int currentAccList [] = tSession.getAccountList ("Current");
    int creditcardList [] = tSession.getAccountList (
                                "Credit Card"
                            );
    int accList[] = new int[currentAccList.length +
                            creditcardList.length];
    System.arraycopy(
        currentAccList, 0, accList, 0, currentAccList.length
    );
    System.arraycopy(creditcardList, 0, accList,
   currentAccList.length, creditcardList.length);

7   openAccount dlg = new openAccount (this, true);
    dlg.setAccList (accList);
    dlg.show();

    // Check to see if the user didn't cancel the operation
8   if ( dlg.exit_status == 1 )
    {
       return;
    }

    presentAccountNumber = dlg.accountNum;
    try {
```

Example 9: *The openAccountActionPerformed() Method (Sheet 3 of 3)*

```
9       tSession.openAccount (dlg.accountNum);
      }
      catch ( ... ) { ... }
      // Handle all exceptions (not shown)...

      fnbba.AccountInfo_s accInfo = null;
      try {
10      accInfo = tSession.getAccountInfo ();
      }
      catch ( ... ) { ... }
      // Handle all exceptions (not shown)...

      // OK, let's get the information back from the dialog box
11    nameText.setText(accInfo.lname + ", " + accInfo.fname);
      accTypeText.setText(accInfo.accType);

      addrText1.setText(accInfo.addr1);
      addrText2.setText (accInfo.addr2);
      addrText3.setText (accInfo.addr3);
      accountNumText.setText(
          String.valueOf(presentAccountNumber)
      );

      accBalance = tSession.accBalance ();
      refreshTransList (accInfo.transactions);
    }
```

The preceding code can be explained as follows:

1. The `openAccountActionPerformed()` method is called when the user selects the **Account|Open Account** menu option from the main screen.
2. Most of the work of the `openAccountActionPerformed()` method is delegated to the `genericOpenAccount()` method.
3. The `genericOpenAccount()` method is called by both the `openAccountActionPerformed()` and the `openAccountButtonActionPerformed()` methods.
4. This line and the following lines initialize an `fnbba.SessionInfo_s` object with default session login details. For the IDL definition of the `SessionInfo_s` struct, see “IDL for the Middle-Tier Server” on page 51.

5. The openSession() operation is invoked on the remote BusinessSessionManager object, with the SessionInfo_s struct being passed as an inout argument.
6. The getAccounts() operation is invoked on the session object reference, tsession, to get a list of all Current accounts and Credit Card accounts.
7. The **Open Account** dialog window is created, dlg, and the combined list of accounts is passed to the dialog as initial data.

   The call to dlg.show() passes control to the dialog window.
8. The dialog exit_status is checked to see if the user clicked **Cancel**.
9. Otherwise the user must have clicked **OK**, in which case the account is opened with the user-selected account number, dlg.accountNum.
10. The details for the currently active account, accInfo, are retrieved from the business session, tsession.
11. The account information extracted from accInfo is displayed in the main screen.

Part II

J2EE Internet Banking

In this part

This part contains the following chapters:

J2EE AllDayBanking Application	page 89
Accessing the CORBA Back-End	page 109
EJB Middle-Tier	page 123
J2EE Presentation Layer	page 149

CHAPTER 4

J2EE AllDayBanking Application

This chapter gives an overview of the J2EE AllDayBanking application and of the tools and utilities that are provided for building, packaging, and deploying J2EE applications.

In this chapter

This chapter discusses the following topics:

Architecture of the J2EE Application	page 90
Overview of the J2EE Development Cycle	page 92
Source Code Organization (EARSCO)	page 94
Building and Packaging the J2EE Application	page 98

Architecture of the J2EE Application

Overview

Figure 26 shows the architecture of the J2EE AllDayBanking application. Both the presentation layer and the middle tier of the application are implemented using J2EE technology, while the back-end is implemented using CORBA technology.

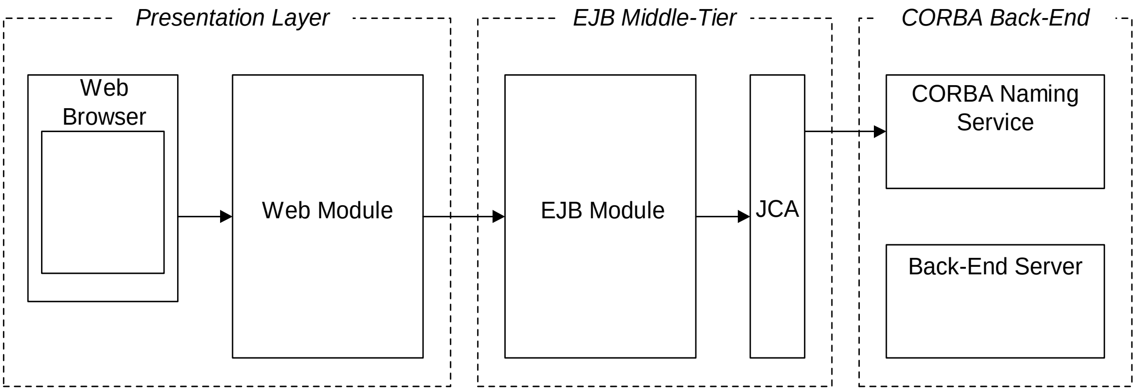


Figure 26: *Architecture of the J2EE AllDayBanking Application*

CORBA back-end

The CORBA back-end server provides access to the persistent account data stored in the back-end database—see "Back-End CORBA Server" on page 3 for details. A link to the back-end server can be established by retrieving an `AccountMgr` object reference from the CORBA Naming Service.

Communication with the CORBA back-end uses the OMG's Internet inter-ORB protocol (IIOP).

JCA layer

The Java Connector Architecture (JCA) layer is used to bootstrap connections between the EJB middle tier and the CORBA backend.The JCA is a Java standard that describes how to integrate J2EE applications with external third-party resources.

EJB middle-tier

The middle tier is based on the J2EE Enterprise Java Beans (EJB) technology. This layer implements the application business logic using a collection of enterprise beans. See “EJB Middle-Tier” on page 123 for details of the bean implementations.

Presentation layer

The J2EE presentation layer is designed to be integrated with a Web server. It consists of two parts:

- *HTML pages and Java Server Pages (JSP)*—the content that is served up to Web clients by the Orbix Application Server (the application server is also a Web server).
- *Worker beans*—are helper classes that cooperate with JSP pages to simplify the presentation logic.

Web client

The Web client is an ordinary Web browser, such as Internet Explorer or Netscape.

After the J2EE application has been deployed on the Orbix Application Server, a Web client can access the J2EE AllDayBanking application by going to the following URL:

http://*AppServerHost*:8080/AllDayBanking

Assuming you are running the application on the JBoss platform, where *AppServerHost* is the host on which the J2EE application server is running.

Note: You could also run the application on another implementation of the J2EE platform—for example, WebSphere or WebLogic.

Overview of the J2EE Development Cycle

Development cycle

Figure 27 shows an overview of the J2EE development cycle. Orbix provides a comprehensive set of utilities for simplifying each stage of the cycle.

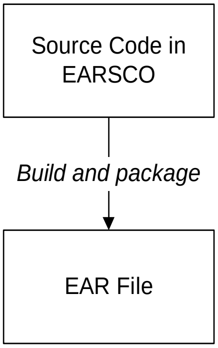


Figure 27: *The J2EE Development Cycle*

Source code organization

Historically, the Orbix E2A ASP 6.0 product (and earlier versions) used a specific directory structure, EARSCO, to store the source files from a J2EE application. Orbix no longer provides special tools to manage this directory structure, but this directory structure is still used for the FNB AllDayBanking demonstration.

The source code organization is described in "Source Code Organization (EARSCO)" on page 94.

Building and packaging

The rules for building and packaging the AllDayBanking J2EE application are encapsulated in the `ibank/build.xml` ant build file. The output from the build step is an Enterprise Application Archive (EAR) file—for example, `AllDayBanking.ear`—which contains a deployable J2EE application.

The building and packaging of the AllDayBanking application is described in "Building and Packaging the J2EE Application" on page 98.

Configuring the container

The details of container configuration are proprietary. Hence, different J2EE application servers would have different configuration requirements for their EJB containers and Web containers.

For example, the JBoss J2EE application server configures an EJB container using a `jboss.xml` file (located in `ibank/AllDayBanking/src/WebStuff.jar/etc`), which has a proprietary format. It is only at this point that the proprietary details of the application server come into play. Abstract security and persistence properties are mapped onto specific security mechanisms and database details.

For more details about the `jboss.xml` file, see "jboss.xml file" on page 133.

Source Code Organization (EARSCO)

EAR files

An Enterprise Application Archive (EAR) file is a compressed archive (in standard zip file format) containing all of the EJB, Web, and client components that constitute a single J2EE application. The purpose of the EAR file format is to simplify deployment of J2EE applications by bundling all of the required files into a single archive.

The contents of an EAR file have a standard directory layout, the details of which are described in "Directory Structure in an EAR File" on page 100.

Source code organization

Although the J2EE standard defines a standard layout for storing all of your compiled code and configuration files within an EAR file, there is *no* equivalent layout defined by J2EE for organizing your source code files.

EARSCO

Historically, the Enterprise Application Archive Source Code Organization (EARSCO) was used to organize J2EE source code for the Orbix E2A ASP product. IONA's J2EE application server is no longer part of the Orbix product, but the same EARSCO directory structure is still used to hold the source code for the AllDayBanking demonstration.

The `ibank/build.xml` ant build file is designed to be compatible with the EARSCO directory structure, enabling you to build and package the AllDayBanking demonstration in a single step—see "Building and Packaging the J2EE Application" on page 98.

EARSCO overview

Figure 28 gives a general overview of the EARSCO.

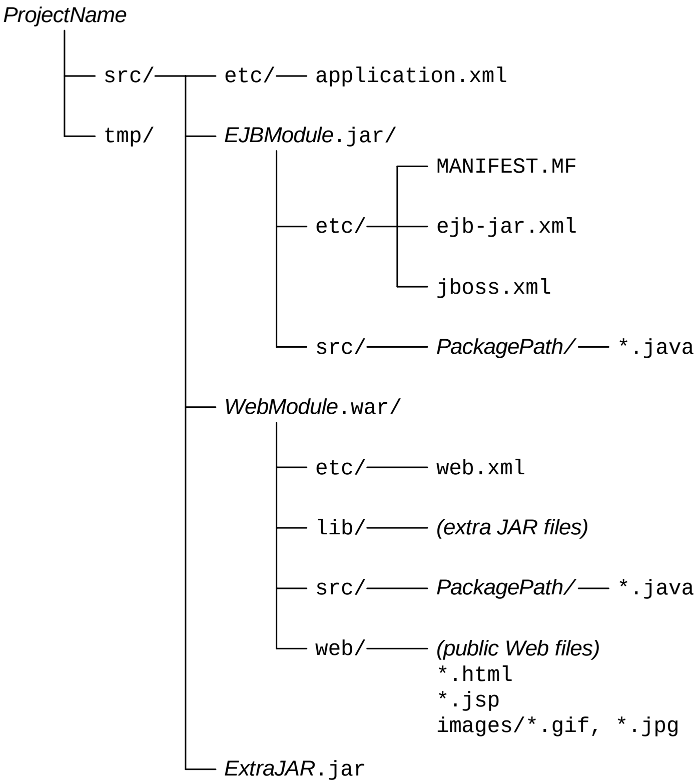


Figure 28: *The Enterprise Application Archive Source Code Organization*

application.xml file

The `application.xml` file is a standard J2EE configuration file that specifies which modules are in the J2EE application—see also "Directory Structure in an EAR File" on page 100.

EJB modules

An EJB module is a collection of enterprise Java beans that cooperate to provide a certain unit of functionality.

For every EJB module, *EJBModule*, a *ProjectName*`/src/`*EJBModule*`.jar` directory contains the following standard elements:

- *EJBModule*`.jar/etc/ejb-jar.xml`—the `ejb-jar.xml` file is a standard J2EE file that specifies the basic configuration for the enterprise beans in the EJB module.
- *EJBModule*`.jar/etc/MANIFEST.MF`—the `MANIFEST.MF` is an optional file that can be used to specify additional meta-information for the EJB module—see "MANIFEST.MF file" on page 104. For example, you can use the `MANIFEST.MF` file to specify a class path for the EJB module—see "Accessing the Stub JARs from EJB" on page 121.
- *EJBModule*`.jar/src/`—the `src/` subdirectory is the root of all the Java source code for the enterprise beans in the EJB module.

And, if you are deploying the application to JBoss, one additional non-standard element:

- *EJBModule*`.jar/etc/jboss.xml`—the `jboss.xml` file is a non-standard file that JBoss uses to map abstract EJB references to concrete resources in the EJB container.

Web modules

A Web module contains all of the files that are needed for the presentation layer of a J2EE application. This typically includes HTML files, Java server pages, and ordinary Java beans.

For every Web module, *WebModule*, a *ProjectName*`/src/`*WebModule*`.jar` directory contains the following standard elements:

- *WebModule*`.jar/etc/web.xml`—the `web.xml` file is a standard J2EE file that specifies the basic configuration of the Web module.
- *WebModule*`.jar/etc/MANIFEST.MF`—the `MANIFEST.MF` is an optional file that can be used to specify additional meta-information for the Web module. See "MANIFEST.MF file" on page 104.
- *WebModule*`.jar/lib/`—the `lib/` subdirectory can hold JAR files used by the Web module. The JAR files in this directory are automatically made accessible to the Web module without needing to be added to the class path.

- *WebModule*.jar/src/—the src/ subdirectory is the root of all the Java source code in the Web module.
- *WebModule*.jar/web/—the web/ subdirectory contains all of the Web module's *public files* (that is, files that can be downloaded through a Web server). This directory typically contains HTML files, JSP files, and graphics files (*.gif, *.jpg and so on).

Extra JAR files

You can place extra JAR files directly into the *ProjectName*/src directory. To make the extra JAR files accessible to an EJB module, use the Java extension mechanism—see "Accessing the Stub JARs from EJB" on page 121.

tmp directory

The *ProjectName*/tmp directory is used to hold intermediate files created in the course of building and packaging the J2EE application.

Building and Packaging the J2EE Application

The ant build file

Complete rules for building and packaging the AllDayBanking demonstration are encapsulated in the relevant ant build file, `ibank/build.xml`. Hence, you can build and package the J2EE demonstration by entering the following at a command prompt:

```
cd FNBHome/ibank/
itant build
```

The `itant` utility is a wrapper for the standard `ant` build utility from apache. By default, the `itant` utility reads the build rules from a file called `build.xml` in the current directory. For more details, see:

http://jakarta.apache.org/ant/

What happens when you build the application?

When you invoke `itant build` in the `ibank` directory, the ant utility builds and packages the J2EE application, performing the following tasks:

1. Compiles the J2EE application code.
2. Places all of the intermediate build files into the `ibank/AllDayBanking/tmp` directory.
3. Packages the compiled J2EE application into an EAR file, `ibank/AllDayBanking/AllDayBanking.ear`.

Files generated

The `itant build` (or `ant build`) command generates the following files under the `ibank/AllDayBanking` directory:

- Files under the `tmp/` directory—intermediate build files.
- *ProjectName*`.ear`—the complete J2EE application packaged as an Enterprise Application Archive.

The Enterprise Application Archive file format

The EAR file is basically a zip file, except that the file suffix is `.ear`. It's contents can be viewed using the Java `jar` utility or any other standard zip file utility. The directories and files in the EAR file conform to a standard layout, which is described in this section.

In this section

This section contains the following subsections:

Directory Structure in an EAR File	page 100
Directory Structure in an EJB Module JAR File	page 102
Directory Structure in a Web Module WAR File	page 105

Directory Structure in an EAR File

Overview

Figure 29 shows the standard directory structure and layout of an EAR file.

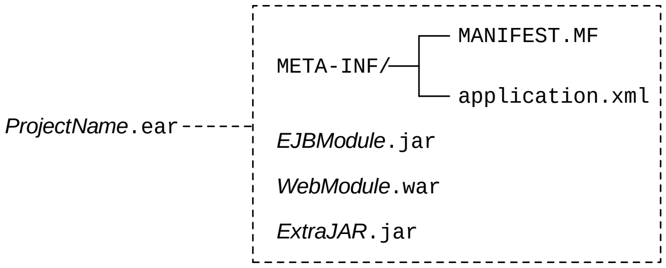


Figure 29: *Standard Layout of an EAR File*

META-INF/ directory

The `META-INF/` directory can contain the following files:

- `META-INF/application.xml`—a standard J2EE configuration file that specifies which modules are in the J2EE application.
- `META-INF/MANIFEST.MF`—an optional file that can be used to specify additional meta-information for the EAR. See "MANIFEST.MF file" on page 104.

application.xml file

The application.xml file is a standard XML file that specifies the modules to include in a J2EE application. For example, the AllDayBanking demonstration defines the following application.xml file:

```
<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD
   J2EE Application 1.2//EN'
   'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>

<application>
  <display-name>AllDayBanking</display-name>
  <module>
    <ejb>WebStuff.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>WebStuff.war</web-uri>
      <context-root>AllDayBanking</context-root>
    </web>
  </module>
</application>
```

EJB module JAR files

Each EJB module is packaged in a JAR file—see “Directory Structure in an EJB Module JAR File” on page 102.

Web module JAR files

Each Web module is packaged in a JAR file—see “Directory Structure in a Web Module WAR File” on page 105.

Extra JAR files

Extra JAR files are JAR files that are referenced by the J2EE application but are not modules in their own right. Some extra configuration is required to make them accessible to an EJB module—see “Accessing the Stub JARs from EJB” on page 121 for details.

Directory Structure in an EJB Module JAR File

Overview

Figure 30 shows the standard directory structure and layout of an EJB module JAR file including an additional, proprietary, `jboss.xml` file.

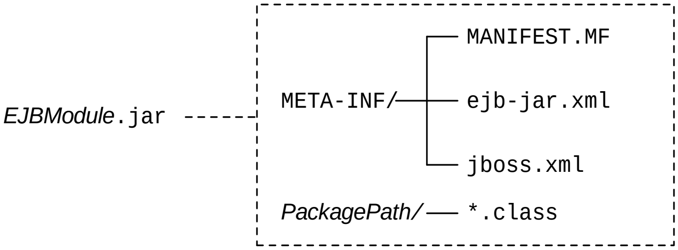


Figure 30: *Layout of an EJB Module JAR File*

META-INF/ directory

The `META-INF/` directory can contain the following standard files:

- `META-INF/ejb-jar.xml`—the EJB deployment descriptor for this EJB module.
- `META-INF/MANIFEST.MF`—an optional file that can be used to specify additional meta-information for the JAR. See "MANIFEST.MF file" on page 104. For example, `MANIFEST.MF` can be used to extend the CLASSPATH used by the EJB module—see "Accessing the Stub JARs from EJB" on page 121.

And the following non-standard file for JBoss deployments:

- `META-INF/jboss.xml`—a file that maps abstract EJB references to concrete container resources.

ejb-jar.xml file

The purpose of the EJB deployment descriptor, `ejb-jar.xml`, is to describe the enterprise beans in the EJB module to the application container. Example 10 shows the partial contents of the `ejb-jar.xml` file from the WebStuff EJB module in the AllDayBanking application.

Example 10: *Part of the ejb-jar.xml File from the EJB Module in the AllDayBanking Application*

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
   Enterprise JavaBeans 1.1//EN'
   'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  ...
  <display-name>EJB Modules</display-name>
  <enterprise-beans>
    <session>
      <display-name>InetAccount</display-name>
      <ejb-name>InetAccount</ejb-name>
      ...
    </session>
    <entity>
      <display-name>User</display-name>
      <ejb-name>User</ejb-name>
      ...
    </entity>
  </enterprise-beans>

  <assembly-descriptor>
    ...
  </assembly-descriptor>
</ejb-jar>
```

In Example 10, two types of element are nested directly within the `<ejb-jar>` element, as follows:

`<enterprise-beans>`

This element contains a basic description of every session and entity bean in the EJB module, using nested `<session>` and `<entity>` elements.

`<assembly-descriptor>`

This optional element describes how the beans are used in conjunction with standard J2EE services. For example, the assembly descriptor can be used to assign security roles to beans, and to describe transactional behavior.

MANIFEST.MF file

A MANIFEST.MF file is a standard component of a JAR file. Historically, it was introduced to support packaging options for Java applets (such as, for example, the addition of a digital signature). Manifest files are now used for J2EE archives as well, where they can store various kinds of meta-information about an archive.

For a tutorial introduction to manifest files, see the following URL:

http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html

For a detailed specification of the manifest file format, see:

http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html

Note: When editing a MANIFEST.MF file, be sure to include a carriage return at the end of the file.

jboss.xml file

The jboss.xml file is a proprietary file that needs to be included in the *EJBModule*.jar file, only if you deploy the EJB module to a JBoss J2EE application server.

For more details, see "jboss.xml file" on page 133.

Class files

The EJB module JAR file also contains the module's class files. The class files are arranged within the standard directory structure produced by the Java compiler, javac.

Directory Structure in a Web Module WAR File

Overview

Figure 31 shows the standard directory structure and layout of a Web module WAR file.

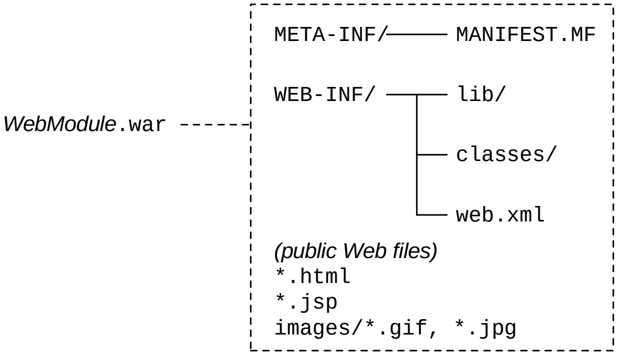


Figure 31: *Standard Layout of a Web Module WAR File*

META-INF/ directory

The `META-INF/` directory can contain the following file:

- `META-INF/MANIFEST.MF`—an optional file that can be used to specify additional meta-information for the WAR. See “MANIFEST.MF file” on page 104.

WEB-INF/ directory

The `WEB-INF/` directory contains a Web archive’s *private* files and directories. That is, when the Web archive is deployed, the files and directories under the `WEB-INF/` directory cannot be accessed directly by Web clients.

The `WEB-INF/` directory can contain the following files and directories:

- `WEB-INF/web.xml`
- `WEB-INF/lib/`
- `WEB-INF/classes/`

WEB-INF/web.xml file

The `WEB-INF/web.xml` file is a *Web deployment descriptor*, a standard J2EE file that specifies the basic configuration of the Web module.

Example 11 shows an extract from the AllDayBanking Web deployment descriptor, `web.xml`. In this example, the Web deployment descriptor is used primarily to specify references to enterprise beans.

Example 11: *Extract from the AllDayBanking Web Deployment Descriptor*

```
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web
   Application 2.2//EN'
   'http://java.sun.com/j2ee/dtds/web-app_2.2.dtd'>

<web-app>
  <display-name>Web Modules</display-name>
  <session-config>
    <session-timeout>5</session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>/index.html</welcome-file>
  </welcome-file-list>
  ...
  <ejb-ref>
    <ejb-ref-name>alldaybanking/InetAccount</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>alldaybanking.session.InetAccountHome</home>
    <remote>alldaybanking.session.InetAccount</remote>
    <ejb-link>InetAccount</ejb-link>
  </ejb-ref>
  ...
</web-app>
```

WEB-INF/lib/ directory

The `WEB-INF/lib/` subdirectory can store JAR files used by the Web module. The JAR files in this directory are automatically accessible to the Web module without needing to be added to the class path.

WEB-INF/classes/ directory

The `WEB-INF/classes/` subdirectory contains the compiled Java code for the Web module.

Public files and directories

All of the files and directories not stored under the special `WEB-INF` directory are *public*. After the Web archive is deployed, public files and directories can be accessed directly by Web clients.

Public files typically include the following:

- HTML files.
- JSP files.
- Image files and other multimedia files—it is a common convention to store image files in an `images` subdirectory.

References

For an example of how a Web archive is used in practice, see "J2EE Presentation Layer" on page 149.

CHAPTER 5

Accessing the CORBA Back-End

The AllDayBanking EJB middle-tier functions both as a CORBA client and as an EJB server. This chapter discusses how to configure and package the EJB application so that it can gain access to the CORBA back-end.

In this chapter

This chapter discusses the following topics:

Overview of the EJB to CORBA Link	page 110
Using Orbix Connect and JBoss	page 112
Creating the IDL Stub JAR File	page 119
Accessing the Stub JARs from EJB	page 121

Overview of the EJB to CORBA Link

Overview

Figure 32 shows an overview of the link between the EJB middle-tier and the CORBA back-end server. In this architecture, the `InetAccount` session bean acts as a CORBA client of the back-end server. The EJB middle-tier, therefore, uses a mixture of J2EE and CORBA technologies.

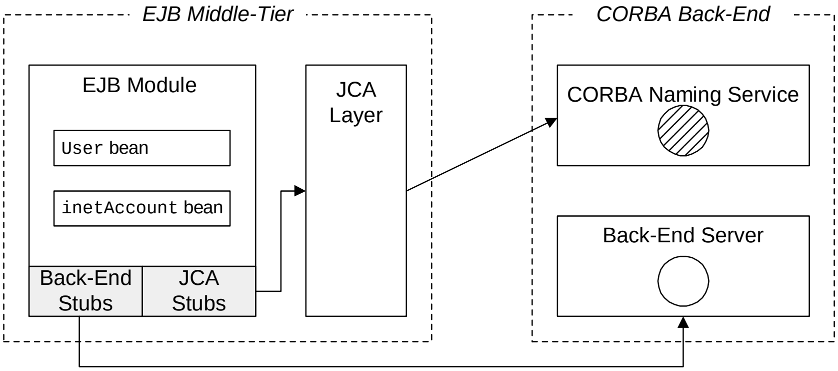


Figure 32: *The EJB Middle-Tier Accesses the CORBA Back-End*

CORBA back-end

The back-end registers a `bankobjects::AccountMgr` object with the CORBA naming service. This makes the `AccountMgr` object accessible to applications that can use the IIOP protocol.

JCA layer

The *Java Connectivity Architecture* (JCA) layer acts as a bridge between the EJB middle tier and the CORBA back-end. The main purpose of the JCA layer is to bootstrap connections between J2EE and CORBA. JCA provides a simplified programming interface, which J2EE applications use for looking up CORBA objects in the CORBA naming service.

EJB module

The EJB module uses the JCA programming interface to gain access to CORBA objects in the CORBA back-end. With the help of the JCA layer, an EJB bean can obtain a CORBA object reference using just a few lines of code.

To gain access to the CORBA back-end, the EJB module needs some additional JAR libraries, as follows:

- JCA stub code.
- Back-end stub code.

JCA stub code

The JCA stub code provides access to the JCA programming interface. The EJB middle tier uses the JCA API to lookup CORBA object references.

Note: The JCA stub JAR is *not* part of the Orbix product. You can get the JCA stub from a JCA implementation—for example, Orbix Connect.

Back-end stub code

The IDL stub code enables the EJB middle-tier to invoke operations on the CORBA objects in the back-end server. The application IDL stub code (that is, the stub code derived from the `Account.idl` and `BusinessSessionManager.idl` files) must be explicitly included in the EJB module.

Using Orbix Connect and JBoss

Overview

The AllDayBanking demonstration uses the Orbix product only for the back-end. The middle-tier and the presentation layer require third-party J2EE application server software in order to run. Hence, to complete the AllDayBanking demonstration, you should install the following additional products:

- Orbix Connect.
- JBoss.

Orbix Connect

Orbix Connect (http://www.iona.com/products/orbix_connect.htm) is IONA's implementation of the J2EE Connector Architecture (JCA). The purpose of JCA is to provide a standardized way for J2EE applications to link to external resources. In particular, Orbix Connect provides a way of linking J2EE applications to CORBA servers.

JBoss

JBoss (www.jboss.org) is an open source, J2EE-based application server. The JBoss application server is free software, distributed under the Lesser Gnu Public Licence (LPGL). You can download a free copy of the JBoss application server from the following URL:

http://www.jboss.org/downloads

Orbix Connect and JBoss scenario

Figure 33 shows an example of a specific scenario where the EJB middle tier (JBoss) connects to the CORBA back-end (Orbix), using a JCA connector layer (Orbix Connect).

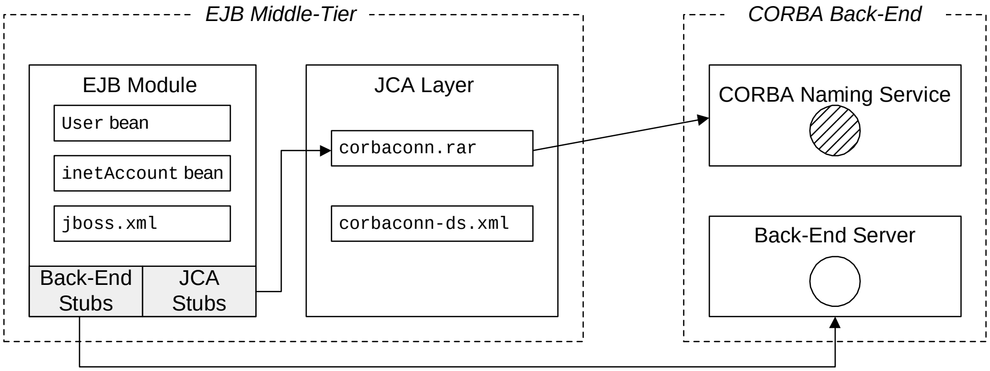


Figure 33: *EJB to CORBA Connectivity Using Orbix Connect and JBoss*

Orbix Connect JCA layer

The Orbix Connect JCA layer is used to bootstrap connections between the EJB middle-tier and the CORBA back-end. To perform this bootstrapping role, the JCA layer relies on the CORBA naming service in the back-end. The JCA layer provides a simple API that enables J2EE applications to retrieve CORBA object references from the CORBA naming service.

The Orbix Connect JCA layer consists of the following files:

- corbaconn.rar file
- corbaconn-ds.xml file

corbaconn.rar file

The `corbaconn.rar` file is the Resource Adapter aRchive (RAR) file for the Orbix Connect product. The RAR file contains all of the code and configuration details required for a client ORB, as well as the code that implements the JCA programming interface.

To deploy the Orbix Connect RAR file, you can copy the following file:

OrbixConnectHome`/lib/corbaconn.rar`

to the JBoss deploy directory (*OrbixConnectHome* is the directory where Orbix Connect is installed).

corbaconn-ds.xml file

The corbaconn-ds.xml file contains the configuration settings that initialize the Orbix Connect JCA adapter. In this example, the main purpose of the corbaconn-ds.xml file is to provide the JCA adapter with the location of the CORBA naming service. The corbaconn-ds.xml file must be copied into the JBoss deploy directory.

The file naming convention, *AdapterName*-ds.xml, and the format of the *-ds.xml files are specific to the JBoss J2EE application server. JBoss uses *-ds.xml data source files to configure adapters to third-party resources. When a JBoss J2EE application server starts up, it reads all of the *-ds.xml files in the deployment directory and imports the configuration data from these files, making the data available through the Java Naming and Directory Interface (JNDI).

In this example, the Orbix Connect JCA configuration data is made available through the following JNDI name:

```
java:/CORBAConnector
```

Configuration Based on an Orbix Configuration Domain

The FNB demonstration features two alternatives for the corbaconn-ds.xml file. The first alternative (configuration based on an Orbix configuration domain) is shown in Example 12.

Example 12: *JCA Configuration Based on an Orbix Configuration Domain*

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>CORBAConnector</jndi-name>
    <adapter-display-name>CORBAConnector</adapter-display-name>
    <config-property name="ORBConfig" type="java.lang.String">
      file://@IT_CONFIG_DOMAINS_DIR@/@IT_DOMAIN_NAME@.cfg
    </config-property>
  </no-tx-connection-factory>
</connection-factories>
```

The JCA configuration shown in Example 12 assumes, as a prerequisite, that the IT_CONFIG_DOMAINS_DIR and the IT_DOMAIN_NAME variables are set in your system environment—that is, you must have initialized an Orbix configuration domain. When you run itant jboss_deploy, the

@IT_CONFIG_DOMAINS_DIR@ and @IT_DOMAIN_NAME@ macros from Example 12 are substituted with literal values and the corbaconn-ds.xml file is copied into the JBoss deployment directory.

The <jndi-name> tag specifies that the configuration data is stored under the java:/CORBAConnector JNDI name.

Configuration Based on a corbaloc URL

The second alternative (configuration based on a corbaloc URL) is shown in Example 13.

Example 13: *JCA Configuration Based on a corbaloc URL*

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>CORBAConnector</jndi-name>
    <adapter-display-name>CORBAConnector</adapter-display-name>
    <config-property name="NameServiceReference"
   type="java.lang.String">
      corbaloc:iiop:1.2@localhost:3075/NameService
    </config-property>
  </no-tx-connection-factory>
</connection-factories>
```

The JCA configuration shown in Example 13 can be used, if an Orbix configuration domain is not available. In fact, this configuration could be used to integrate Orbix Connect with *any* back-end ORB that supports IIOP. It might be necessary to edit the corbaloc: URL, however. See the *Orbix Connect User's Guide* for more details.

The <jndi-name> tag specifies that the configuration data is stored under the java:/CORBAConnector JNDI name.

JBoss EJB module

The JBoss EJB module requires the following files in order to integrate with the JCA layer and the CORBA back-end:

- Stub JARs.
- jboss.xml file.
- ejb-jar.xml.

Stub JARs

It is necessary to bundle some stub JAR files with the EJB module, as follows:

- `api.jar`—contains the public API for the Orbix Connect JCA adapter (copied from *OrbixConnectHome*`/lib/corbaconn/api/1.0`).
- `idlstubs.jar`—contains the IDL stubs for the CORBA back-end.

For full details about how to include these stub JARs in a J2EE application, see "Accessing the Stub JARs from EJB" on page 121.

jboss.xml file

The `jboss.xml` file is used, in addition to the standard `ejb-jar.xml` file, to configure the JBoss EJB container. Some special XML tags must be added to the `jboss.xml` file to make the JCA adapter available to an EJB bean.

For example, the `InetAccount` session bean is configured by the `jboss.xml` file shown in Example 14:

Example 14: *jboss.xml Configuration File*

```
<?xml version="1.0" encoding="UTF-8"?>
...
<jboss>
   <enterprise-beans>
      <session>
         <ejb-name>InetAccount</ejb-name>
           <resource-ref>
             <res-ref-name>eis/CorbaConn</res-ref-name>
<res-type>com.iona.j2ee.resourceadapter.CorbaConnection</res-typ
   e>
             <jndi-name>java:/CORBAConnector</jndi-name>
           </resource-ref>

      </session>
      ...
   </enterprise-beans>
</jboss>
```

The configuration shown in Example 14 on page 116 specifies that the Orbix Connect JCA adapter can be accessed by resolving the `java:comp/env/eis/CorbaConn` JNDI name. For more details about the `jboss.xml` file, see "jboss.xml file" on page 133.

ejb-jar.xml

The JCA connector must also be declared as a resource within the `ejb-jar.xml` file. For the AllDayBanking application, the JCA connector must be declared as a resource for the `InetAccount` and the `ValidateCreditCard` session beans. Example 15 shows how the JCA resource is declared for the `InetAccount` bean.

Example 15: *ejb-jar.xml Configuration File*

```
<?xml version="1.0" encoding="UTF-8"?>
...
<ejb-jar>
  ...
  <enterprise-beans>
      ...
    <session>
      <display-name>InetAccount</display-name>
      <ejb-name>InetAccount</ejb-name>
      <home>alldaybanking.session.InetAccountHome</home>
      <remote>alldaybanking.session.InetAccount</remote>

   <ejb-class>alldaybanking.session.InetAccountBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <resource-ref>
        <res-ref-name>eis/CorbaConn</res-ref-name>
<res-type>com.iona.j2ee.resourceadapter.CorbaConnection</res-typ
   e>
        <res-auth>Container</res-auth>
      </resource-ref>
    </session>
    ...
  </enterprise-beans>

  <assembly-descriptor>
    ...
  </assembly-descriptor>
</ejb-jar>
```

Establishing an EJB to CORBA link

Using the configuration settings shown in this section, an EJB bean (such as `InetAccount`) can bootstrap a connection to the CORBA back-end using the API provided by the JCA layer.

Your EJB code can obtain a reference to the JCA adapter by resolving the the `java:comp/env/eis/CorbaConn` JNDI name. With the help of the JCA adapter, it takes just a few lines of code to establish a link to the CORBA server.

For a complete code example, see "Implementation of ejbCreate()" on page 129.

Creating the IDL Stub JAR File

Overview

This section provides an overview of the steps required to create the IDL stub JAR file, `idlstubs.jar`.

Note: There is no need to perform these steps for the AllDayBanking application, however, because the `idlstubs.jar` file is already provided in the `AllDayBanking/lib/` directory.

Steps to create the idlstubs.jar file

From the `AllDayBanking` directory, you can recreate the `idlstubs.jar` file with the following steps:

Step	Action
1	Compile the IDL files. Invoke the CORBA IDL compiler, `idl`, as follows: `idl -jbase=-OAllDayBanking/classes/idl_java -jpoa=-OAllDayBanking/classes/idl_java AllDayBanking/idl/BusinessSessionManager.idl AllDayBanking/idl/Account.idl` The generated output includes both client stub code (generated by the `-jbase` option) and server skeleton code (generated by the `-jpoa` option). The output is put into the `AllDayBanking/classes/idl_java` directory.
2	Compile the Java code. Use the Java compiler, `javac`, to compile all of the source files in the `AllDayBanking/classes/idl_java` directory and place the output file in the `AllDayBanking/classes/idl_classes` directory. While compiling, make sure that you use the correct `CLASSPATH` for your Orbix configuration domain. For a particular domain, *DomainName*, the `CLASSPATH` is normally initialized when you run the *DomainName*`_env.bat` (Windows) or *DomainName*`_env.sh` (UNIX) script.

Step	Action

3	Create the JAR file.
	Use the standard Java utility, `jar`, to package the compiled stub code into a JAR file, `idlstubs.jar`, as follows:
	`jar cf AllDayBanking/lib/idlstubs.jar`
	`   AllDayBanking/classes/idl_classes`

Accessing the Stub JARs from EJB

Overview

To make a stub JAR file (for example, `idlstubs.jar` and `api.jar`) accessible to an EJB module, you must:

- Include the stub JAR file in the application EAR file, and
- Use the Java extension mechanism to add the stub JAR to the EJB module's class path.

Including the stub JAR files

The IDL stub JAR file, `idlstubs.jar`, and the JCA stub JAR file, `api.jar`, must be included somewhere in the application EAR file. For example, the top-level directory inside the `AllDayBanking.ear` file contains the following files and directories:

```
META-INF/
api.jar
idlstubs.jar
WebStuff.jar
WebStuff.war
```

For example, to add the `idlstubs.jar` file to the application EAR file, put `idlstubs.jar` into the `ibank/AllDayBanking/src/` directory of the FNB directory structure (see Figure 28 on page 95) and run the `itant build` command from the `ibank` directory to regenerate the EAR file.

The Java extension mechanism

The Java extension mechanism allows you to reference additional packages from within a JAR file. In the context of an EAR file, it enables you to extend the classpath of a specific EJB module to access another JAR file in the Enterprise Archive.

Within the FNB directory structure, you should edit the `MANIFEST.MF` file in the `ibank/AllDayBanking/WebStuff.jar/etc/` directory and add a `Class-Path:` entry of the following form:

```
Class-Path: PathToExtraPackage1.jar PathToExtraPackage2.jar ...
```

For example, the `ibank/AllDayBanking/WebStuff.jar/etc/MANIFEST.MF` file contains the following text:

```
Class-Path: idlstubs.jar api.jar
```

Reference

For further details on the Java extension mechanism, see:

http://java.sun.com/j2se/1.3/docs/guide/extensions/index.html

CHAPTER 6

EJB Middle-Tier

The EJB middle-tier implements the business logic for the AllDayBanking application. This chapter describes the implementation and configuration of a session bean and an entity bean from the AllDayBanking EJB middle-tier.

In this chapter

This chapter discusses the following topics:

The InetAccount Session Bean	page 124
The User Entity Bean	page 135

The InetAccount Session Bean

Overview

The purpose of the `InetAccount` session bean is to provide clients with temporary access to an `Account` object in the CORBA back-end.

Because the `InetAccount` session bean is effectively a wrapper for an `Account` CORBA object, the methods defined on the `InetAccount` bean offer similar functionality to the `Account` IDL interface.

One of the differences between `InetAccount` and `Account` is that the `InetAccount` bean defines a `resolveAccount()` method to associate an `InetAccount` bean with a particular `Account` object. After the association is established, subsequent method calls on `InetAccount` are delegated to that `Account` object. The association can be switched to a different `Account` object, however, by making a subsequent call to `resolveAccount()`.

In this section

This section contains the following subsections:

Anatomy of a Session Bean	page 125
EJB Session Bean Life Cycle Methods	page 128
Session Bean Configuration	page 131

Anatomy of a Session Bean

What is a session bean?

A *session bean* is a remotely accessible bean that exists in the J2EE Application Server for as long as a client session is active. When the client has finished using the EJB application, the session bean can be discarded.

You can think of a session bean as a kind of client proxy. The session bean is an object in the EJB middle-tier that does work on behalf of a particular client.

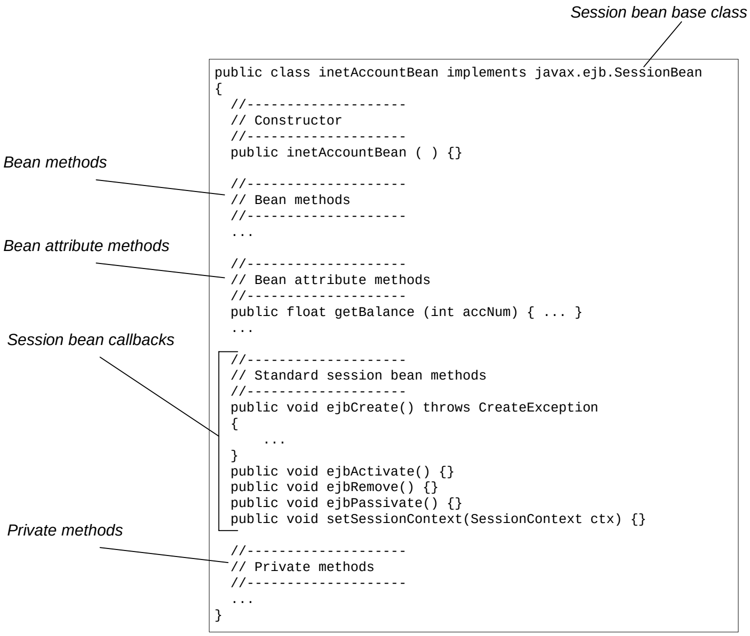
Parts of a session bean

Three elements are needed to implement the `InetAccount` session bean, as follows:

- `InetAccount`—the *remote interface* of the `InetAccount` session bean. This Java interface declares the methods that are made available to remote clients.
- `InetAccountHome`—the *home interface* of the `InetAccount` session bean. This Java interface declares methods for creating `InetAccount` session beans.
- `InetAccountBean`—the bean class provides the implementation of the `InetAccount` session bean.

Structure of the InetAccountBean class

The `InetAccountBean` class is the most important part of the `InetAccount` session bean because it provides the actual implementation of the bean. Figure 34 gives an overview of the structure of the `InetAccountBean` class.



```
public class inetAccountBean implements javax.ejb.SessionBean
{
  //--------------------
  // Constructor
  //--------------------
  public inetAccountBean ( ) {}

  //--------------------
  // Bean methods
  //--------------------
  ...

  //--------------------
  // Bean attribute methods
  //--------------------
  public float getBalance (int accNum) { ... }
  ...

  //--------------------
  // Standard session bean methods
  //--------------------
  public void ejbCreate() throws CreateException
  {
      ...
  }
  public void ejbActivate() {}
  public void ejbRemove() {}
  public void ejbPassivate() {}
  public void setSessionContext(SessionContext ctx) {}

  //--------------------
  // Private methods
  //--------------------
  ...
}
```

Figure 34: *Structure of the InetAccountBean Session Bean Class*

Session bean base class

A session bean class always extends the following standard base class:

`javax.ejb.SessionBean`

Bean methods

All of the methods declared in the remote interface, `InetAccount`, are also defined in the `InetAccountBean` class. The method signatures in the `InetAccountBean` class are the same as in the `InetAccount` remote interface except that the `throws java.rmi.RemoteException` clause is omitted.

Bean attribute methods

Bean methods that conform to either of the following patterns are treated specially:

```
Type getAttributeName();
void setAttributeName(Type x);
```

where *AttributeName* is an attribute of *Type* type. The JavaBeans specification mandates that these methods are recognized as accessor and modifier methods for bean attributes. Various tools and utilities can then use Java reflection to identify the bean attributes automatically.

Session bean callbacks

The following public methods are standard session bean methods that must be defined on every session bean:

```
// Java
public void ejbCreate() throws javax.ejb.CreateException { }
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(javax.ejb.SessionContext ctx) {}
```

See “EJB Session Bean Life Cycle Methods” on page 128.

Private methods

Additional, private methods can be defined for bean-internal use.

EJB Session Bean Life Cycle Methods

Overview

The EJB session bean life cycle methods are called by the EJB container to notify a session bean instance when specific life cycle events occur. The session bean class (for example, `InetAccountBean`) must provide an implementation for each of the life cycle methods, although the implementation of these methods is often trivial or even empty.

ejbCreate() method

The `ejbCreate()` method has the following signature:

```
public void ejbCreate() throws javax.ejb.CreateException
```

Called just after the session bean instance is created, in response to a client calling `create()` on the bean's home interface.

ejbRemove() method

The `ejbRemove()` method has the following signature:

```
public void ejbRemove()
```

Called just before a session bean instance is permanently destroyed, in response to a client calling `remove()` on the bean's remote interface or on the bean's home interface.

ejbPassivate() method

The `ejbPassivate()` method has the following signature:

```
public void ejbPassivate()
```

Called just before the container *passivates* the bean by storing the bean data (typically serializing the bean) and removing the bean instance from memory. The container passivates a bean in order to conserve memory and other resources. The container is prepared, however, to reactivate the bean automatically as soon as it is needed again.

ejbActivate() method

The `ejbActivate()` method has the following signature:

```
public void ejbActivate()
```

Called just after the container has reactivated a bean that was previously passivated.

Implementation of ejbCreate()

Example 16 shows the implementation of the `ejbCreate()` method for the `InetAccountBean` class. The `ejbCreate()` method is called by the J2EE container just after an `InetAccountBean` object is created. This is where you can do any once-off initialization for the new `InetAccountBean` object.

Example 16: *The InetAccountBean.ejbCreate() Method*

```
  // Java
  import bankobjects.AccountMgr;
  import com.iona.corbaconn.CorbaConnectionFactory;
  ...
  public class InetAccountBean implements javax.ejb.SessionBean {
1   private static String EIS_JNDI_NAME =
     "java:comp/env/eis/CorbaConn";
    private CorbaConnectionFactory corbaFact = null;
    private bankobjects.AccountMgr myMgr = null;
    private bankobjects.Account myAccount = null;
    ...
    public void ejbCreate() throws CreateException {
      try {
2       javax.naming.Context ctx =
            new javax.naming.InitialContext();
3       corbaFact = (CorbaConnectionFactory)
                                     ctx.lookup(EIS_JNDI_NAME);
      } catch (javax.naming.NamingException ne) {
        System.err.println(
          "Trouble finding CORBA JCA Connector in JNDI"
        );
        ne.printStackTrace();
      }

      System.out.println("BEAN>In ejbcreate....");

      if (myMgr == null) {
        try {
4         myMgr = (AccountMgr) corbaFact.getConnection(
                                 AccountMgr.class,
                                "Mainframe/BankObjects_AccountMgr"
                               );
        } catch (ResourceException re) {
          System.err.println("Failure location CORBA Object " +
     re);
        }
      }
    }
```

The preceding code can be explained as follows:

1. The `java:comp/env/eis/CorbaConn` URL is a Java Naming and Directory Interface (JNDI) name. This name can be analyzed as follows:

    i. The first part of the name, `java:comp/env`, is a standard prefix used to access J2EE environment variables.

    ii. The second part of the name, `eis/CorbaConn`, is mapped to a connection factory resource by an XML configuration file (in the case of JBoss, this file is `jboss.xml`).

    See "jboss.xml file" on page 133 and "Using Orbix Connect and JBoss" on page 112 for more details.

2. The `javax.naming.InitialContext()` static method creates a new JNDI context, which accesses the default JNDI service provided by the J2EE application container. This is the standard way of accessing JNDI from within an EJB bean.

3. A reference to a `com.iona.corbaconn.CorbaConnectionFactory` object is obtained by looking up the `java:comp/env/eis/CorbaConn` URL in the JNDI service. The CORBA connection factory object is used to get references to remote CORBA objects.

4. The `getConnection()` method is invoked on the CORBA connection factory to obtain a reference to the `bankobjects.AccountMgr` CORBA object. The `getConnection()` method takes the following arguments:

    - *ClassName*`.class`—the type of object reference.
    - CORBA name in string format—the string provided here is resolved in the CORBA naming service relative to the root naming context.

    The value returned by `getConnection()` must be cast to the appropriate type, that is `bankobjects.AccountMgr`.

    The `bankobjects.AccountMgr` instance, `myMgr`, provides direct access to the back-end CORBA server.

Reference

For more details about JNDI, and how it is used within J2EE, see:

- http://java.sun.com/developer/technicalArticles/Programming/jndi/index.html

Session Bean Configuration

Overview

A session bean has two layers of configuration.

The first layer is configured by the following file:

`ejb-jar.xml`	The EJB deployment descriptor.

The second layer is configured by a proprietary container configuration file, which is specific to the particular J2EE deployment platform you are using:

`jboss.xml`	The JBoss container configuration.

ejb-jar.xml file

The EJB deployment descriptor, `ejb-jar.xml`, is a standard J2EE file that conforms to the EJB 1.1 Document Type Definition (DTD). The purpose of this file is to describe the enterprise beans in an EJB module to the EJB container.

For example, the XML code in Example 17 is an incomplete extract from the AllDayBanking deployment descriptor that shows the configuration of the `InetAccount` session bean:

Example 17: *ejb-jar.xml Extract Showing InetAccount Bean Configuration*

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
   Enterprise JavaBeans 1.1//EN'
   'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <display-name>EJB Modules</display-name>
  <enterprise-beans>
    ...
    <session>
      <display-name>InetAccount</display-name>
      <ejb-name>InetAccount</ejb-name>
      <home>alldaybanking.session.InetAccountHome</home>
      <remote>alldaybanking.session.InetAccount</remote>
    <ejb-class>alldaybanking.session.InetAccountBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
    ...
  </enterprise-beans>
```

Example 17: *ejb-jar.xml Extract Showing InetAccount Bean Configuration*

```
  <assembly-descriptor>
    <container-transaction>
      ...
      <method>
        <ejb-name>InetAccount</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>

</ejb-jar>
```

In Example 17, the following elements contain detailed information about the `InetAccount` session bean:

`<session>`

This element provides a basic description of the `InetAccount` session bean. For example, the `<ejb-name>` element gives the name of the session bean; the `<home>`, `<remote>`, and `<ejb-class>` elements identify, respectively, the home, remote, and bean implementation classes.

`<container-transaction>`

This element specifies the transaction properties for all the beans and bean methods in the EJB module. The configuration in Example 17 specifies that every method in `InetAccount` has the `Required` transaction attribute. The `Required` transaction attribute implies that the methods can be called either by a transactional or by a non-transactional client. In the case of a non-transactional client, the container creates the transactional context for the call and automatically commits the transaction at the end of the method call.

jboss.xml file

The JBoss container configuration, `jboss.xml`, is an IONA proprietary file. The purpose of this file is to map abstract bean properties onto specific container resources and services.

For example, the XML code in Example 18 is an extract from the `AllDayBanking` container configuration that shows the configuration of the `InetAccount` session bean and the `validateCreditCard` session bean:

Example 18: *jboss.xml Configuration File*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC
   "-//JBoss//DTD JBOSS 3.0//EN"
   "http://www.jboss.org/j2ee/dtd/jboss_3_0.dtd">

<jboss>
   <enterprise-beans>
      <session>
         <ejb-name>InetAccount</ejb-name>
           <resource-ref>
             <res-ref-name>eis/CorbaConn</res-ref-name>

   <res-type>com.iona.j2ee.resourceadapter.CorbaConnection</res-
   type>
             <jndi-name>java:/CORBAConnector</jndi-name>
           </resource-ref>
      </session>

      <session>
         <ejb-name>ValidateCreditCard</ejb-name>
           <resource-ref>
             <res-ref-name>eis/CorbaConn</res-ref-name>

   <res-type>com.iona.j2ee.resourceadapter.CorbaConnection</res-
   type>
             <jndi-name>java:/CORBAConnector</jndi-name>
           </resource-ref>
      </session>
   </enterprise-beans>
</jboss>
```

The `<resource-ref>` tag contains the following sub-tags:

`<res-ref-name>`

Specifies the name of a J2EE environment variable that is made accessible through JNDI. For example, the `eis/CorbaConn` resource reference name can be accessed using the `java:comp/env/eis/CorbaConn` JNDI name.

`<res-type>`

Specifies the type of object stored under the `java:comp/env/eis/CorbaConn` JNDI name. The type specified here, `com.iona.j2ee.resourceadapter.CorbaConnection`, is implemented by the Orbix Connect RAR, `orbixconn.rar`.

`<jndi-name>`

This JNDI name, `java:/CORBAConnector`, refers to an entry in a JBoss datasource file. The JBoss datasource file is used to store configuration properties for the CORBA connector. For more details, see "corbaconn-ds.xml file" on page 114.

The User Entity Bean

Overview

The purpose of the User entity bean is to store a user's registration details persistently. These registration details are provided when the user registers to use the AllDayBanking Internet application for the first time—see "The New User Registration Web Form" on page 167.

Persistence of the User entity beans is provided by the J2EE Application Server (working together with a specified database resource), using the *container-managed persistence* mechanism.

In this section

This section contains the following subsections:

Anatomy of an Entity Bean	page 136
EJB Entity Bean Life Cycle Methods	page 139
Entity Bean Configuration	page 142
Container-Managed Persistence in JBoss	page 145

Anatomy of an Entity Bean

What is an entity bean?

An *entity bean* is a remotely accessible bean whose state is stored persistently. The entity bean continues to exist across multiple runs of the J2EE Application Server until it is explicitly destroyed.

You can think of an entity bean as the object-oriented representation of a database record (and, typically, that is exactly how it is stored).

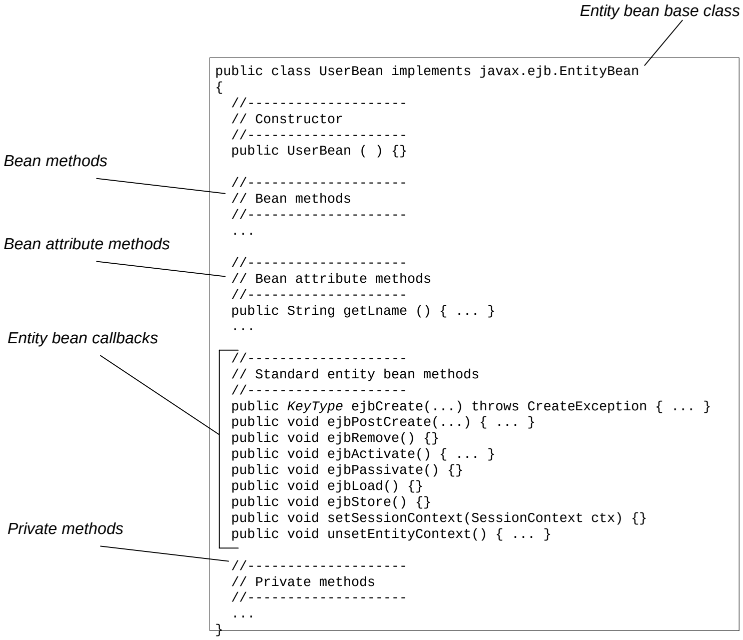
Parts of an entity bean

Three elements are needed to implement the `User` entity bean, as follows:

- `User`—the *remote interface* of the `User` entity bean. This Java interface declares the methods that are made available to remote clients.
- `UserHome`—the *home interface* of the `User` entity bean. This Java interface declares methods for creating and finding `User` entity beans.
- `UserBean`—the bean class provides the implementation of the `User` entity bean.

Structure of the UserBean class

The `UserBean` class is the most important part of the `User` entity bean because it provides the implementation of the bean. Figure 35 gives an overview of the structure of the `UserBean` class.



```
public class UserBean implements javax.ejb.EntityBean
{
  //--------------------
  // Constructor
  //--------------------
  public UserBean ( ) {}

  //--------------------
  // Bean methods
  //--------------------
  ...

  //--------------------
  // Bean attribute methods
  //--------------------
  public String getLname () { ... }
  ...
  //--------------------
  // Standard entity bean methods
  //--------------------
  public KeyType ejbCreate(...) throws CreateException { ... }
  public void ejbPostCreate(...) { ... }
  public void ejbRemove() {}
  public void ejbActivate() { ... }
  public void ejbPassivate() {}
  public void ejbLoad() {}
  public void ejbStore() {}
  public void setSessionContext(SessionContext ctx) {}
  public void unsetEntityContext() { ... }

  //--------------------
  // Private methods
  //--------------------
  ...
}
```

Figure 35: *Structure of the UserBean Entity Bean Class*

User bean base class

An entity bean class always extends the following standard base class:

`javax.ejb.EntityBean`

Bean methods

All of the methods declared in the remote interface, `User`, are also defined in the `UserBean` class. The method signatures in the `UserBean` class are the same as in the `User` interface except that the `throws java.rmi.RemoteException` clause is omitted.

Bean attribute methods

Bean methods that conform to either of the following patterns are treated specially:

```
Type getAttributeName();
void setAttributeName(Type x);
```

where *AttributeName* is an attribute of *Type* type. The JavaBeans specification mandates that these methods are recognized as accessor and modifier methods for bean attributes. Various tools and utilities can then use Java reflection to identify the bean attributes automatically.

Standard entity bean methods

The following public methods are standard entity bean methods that must be defined on every entity bean:

```
// Java
public PrimaryKeyType ejbCreate(InitialData)
   throws javax.ejb.CreateException
public void ejbPostCreate(InitialData)
public void ejbRemove()
public void ejbPassivate()
public void ejbActivate()
public void ejbLoad()
public void ejbStore()
public void setSessionContext(javax.ejb.SessionContext ctx)
public void unsetEntityContext()
```

Where *PrimaryKeyType* is a type that is used to identify the bean (and, by implication, also identifies an associated record in a database). The *InitialData* is an arbitrary list of parameters that is used to initialize the entity bean.

If the entity bean uses bean-managed persistence, you also have to define one or more finder methods, `ejbFind`*`Suffix`*`()`.

See “EJB Entity Bean Life Cycle Methods” on page 139.

Private methods

Additional, private methods can be defined for bean-internal use.

EJB Entity Bean Life Cycle Methods

Overview

The EJB entity bean life cycle methods are called by the EJB container to notify an entity bean instance when specific life cycle events occur. The entity bean class (for example, `UserBean`) must provide an implementation for each of the life cycle methods.

ejbCreate() methods

There can be several overloaded `ejbCreate()` methods defined on the bean class, one for every `create()` method defined on the home interface. An entity bean `ejbCreate()` method has the following signature:

```
public PrimaryKeyType ejbCreate(InitialData)
   throws javax.ejb.CreateException
```

Called just after the entity bean instance is created, in response to a client calling `create(`*InitialData*`)` on the bean's home interface. The return value from `ejbCreate()`, of *PrimaryKeyType* type, depends on the kind of persistence that is used:

- *Container-Managed Persistence*—returns `null`.
- *Bean-Managed Persistence*—returns the primary key for this bean instance.

ejbPostCreate() methods

For each `ejbCreate()` method, there is a matching `ejbPostCreate()`. An `ejbPostCreate()` method has the following signature:

```
public void ejbPostCreate(InitialData)
   throws javax.ejb.CreateException
```

Called after the entity bean is fully initialized. For example, at this stage both the bean data and the primary key are initialized irrespective of whether container-managed or bean-managed persistence is used.

ejbRemove() method

The `ejbRemove()` method has the following signature:

```
public void ejbRemove()
```

Called just before an entity bean instance is permanently destroyed, in response to a client calling `remove()` on the bean's remote interface or on the bean's home interface.

ejbPassivate() method

The `ejbPassivate()` method has the following signature:

```
public void ejbPassivate()
```

Called just before the container *passivates* the bean by storing the bean data (typically serializing the bean) and removing the bean instance from memory. The container passivates a bean in order to conserve memory and other resources. The container is prepared, however, to reactivate the bean automatically as soon as it is needed again.

ejbActivate() method

The `ejbActivate()` method has the following signature:

```
public void ejbActivate()
```

Called just after the container has reactivated a bean that was previously passivated.

ejbLoad() method

The `ejbLoad()` method has the following signature:

```
public void ejbLoad()
```

Load the entity bean state from the database. Typically, the container calls this method at the start of a transaction to ensure that the state of the bean in memory is synchronized with the state in the database.

ejbStore() method

The `ejbStore()` method has the following signature:

```
public void ejbStore()
```

Store the entity bean state in the database. Typically, the container calls this method at the end of a transaction to update the bean state in the database.

ejbFind() methods

The `ejbFind()` methods need only be defined on the entity bean class if you are using bean-managed persistence. There are no `ejbFind()` methods defined on the `UserBean` entity bean class because the `User` bean is implemented with container-managed persistence.

Implementation of ejbCreate()

Example 19 shows the implementation of the `ejbCreate()` method for the `UserBean` class. The `ejbCreate()` method is called by the J2EE container just after a `UserBean` object is created. In this example, the `ejbCreate()` method simply initializes the member variables of the `UserBean` instance.

The `ejbCreate()` method returns `null` because the `User` bean is implemented with container-managed persistence.

Example 19: *The User.ejbCreate() Method*

```
// Java
  ...
  public Integer ejbCreate(
      String userid, String lname, String fname,
      int accnum, int ccnum, String accpwd, String emailaddr
  )
  throws CreateException
  {
    this.lname = lname;
    this.fname = fname;
    this.userid = userid ;
    this.emailaddr = emailaddr;
    this.accnum = accnum;
    this.ccnum = ccnum;
    this.accpwd = accpwd ;

    return null;
  }
```

Entity Bean Configuration

Overview

An entity bean has two layers of configuration, which correspond to the following XML files:

`ejb-jar.xml`	The EJB deployment descriptor.
`jboss.xml`	The JBoss container configuration.

ejb-jar.xml file

The EJB deployment descriptor, `ejb-jar.xml`, is a standard J2EE file that conforms to the EJB 1.1 Document Type Definition (DTD). The purpose of this file is to describe the enterprise beans in an EJB module to the EJB container.

For example, the XML code in Example 20 is an incomplete extract from the AllDayBanking deployment descriptor that shows the configuration of the `User` entity bean:

Example 20: *ejb-jar.xml Extract Showing User Bean Configuration*

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
   Enterprise JavaBeans 1.1//EN'
   'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <display-name>EJB Modules</display-name>
  <enterprise-beans>
    ...
    <entity>
      <description>
          Entity bean represent a user of the online bank
      </description>
      <display-name>User</display-name>
      <ejb-name>User</ejb-name>
      <home>alldaybanking.entity.UserHome</home>
      <remote>alldaybanking.entity.User</remote>
      <ejb-class>alldaybanking.entity.UserBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>True</reentrant>
      <cmp-field>
        <field-name>userid</field-name>
      </cmp-field>
```

Example 20: *ejb-jar.xml Extract Showing User Bean Configuration*

```
      <cmp-field>
        <field-name>lname</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>fname</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>accnum</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>ccnum</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>accpwd</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>emailaddr</field-name>
      </cmp-field>
      <primkey-field>userid</primkey-field>
    </entity>
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>User</ejb-name>
        <method-name>*</method-name>
      </method>
      ...
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

In Example 20, the following elements contain detailed information about the `User` entity bean:

`<entity>`

This element provides a basic description of the `User` entity bean. For example, the `<ejb-name>` element gives the name of the session bean; the `<home>`, `<remote>`, and `<ejb-class>` elements identify, respectively, the home, remote, and bean implementation classes.

Various other elements nested within the <entity> element are used to configure the User bean for container-managed persistence. The <persistence-type> element has the value Container, which specifies that container-managed persistence is selected. The <cmp-field> elements specify which of member variables in the UserBean class are to be made persistent. One of the UserBean member variables, userid, is designated as the primary key by enclosing it in the <primkey-field> element.

<assembly-descriptor> and <container-transaction>

In Example 20, the <assembly-desciptor> element contains a single nested element, <container-transaction>. The <container-transaction> element specifies that every method in the User bean has the Required transaction attribute. The Required transaction attribute implies that the methods can be called either by a transactional or by a non-transactional client. In the case of a non-transactional client, the container creates the transactional context for the call and automatically commits the transaction at the end of the method call.

jboss.xml file

The JBoss container configuration file, jboss.xml, can be used for the following purposes:

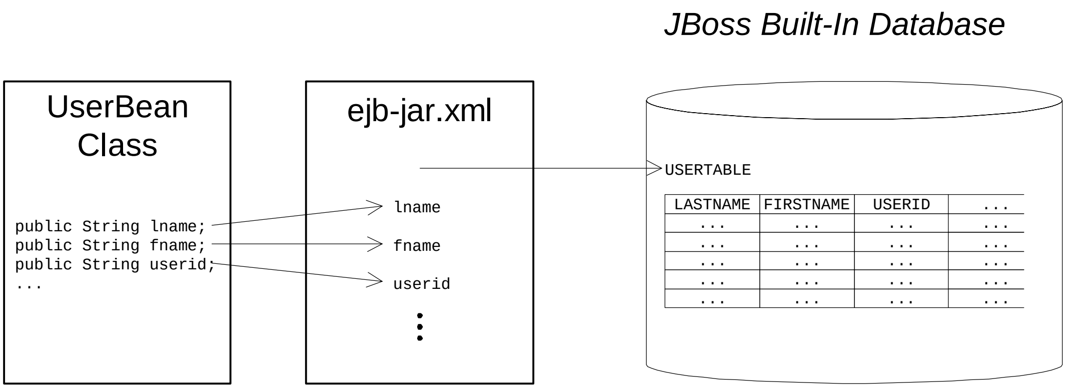
- Declaring references to other EJB beans.
- Declaring resources (for example, if the entity bean needed to access a JCA connector resource).

In the case of the UserBean entity bean, however, no declarations need to be made in the jboss.xml file.

Container-Managed Persistence in JBoss

Overview

Figure 36 gives an overview of container-managed persistence for the `User` entity bean, showing the elements involved in providing the container-managed persistence in JBoss.



```
public String lname;
public String fname;
public String userid;
...
```

LASTNAME	FIRSTNAME	USERID	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...

Figure 36: *Overview of Container-Managed Persistence*

The UserBean class

All of the `UserBean` public member variables are made persistent using container-managed persistence. For example, this includes the `userid`, `lname`, and `fname` member variables.

Container-managed persistence imposes a particular implementation pattern on the entity bean developer. For example, the entity bean is not responsible for reading its state from a database or writing its state to the database. This is looked after automatically by the container. Consequently, the entity bean life cycle methods tend to be rather simple for an entity bean using container-managed persistence—see “EJB Entity Bean Life Cycle Methods” on page 139.

ejb-jar.xml file

The ejb-jar.xml file is responsible for specifying which of the UserBean member variables should be made persistent through container-managed persistence.

In the <entity> element that describes the User entity bean, a sequence of <cmp-field> elements specify the persistent member variables. For example, the following extract from the AllDayBanking ejb-jar.xml file specifies that userid, lname, and fname are persistent variables:

```
...
<ejb-jar>
  ...
  <enterprise-beans>
    ...
    <entity>
      ...
      <cmp-field>
        <field-name>userid</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>lname</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>fname</field-name>
      </cmp-field>
      ...
      <primkey-field>userid</primkey-field>
    </entity>
  </enterprise-beans>
  ...
</ejb-jar>
```

In the preceding extract from ejb-jar.xml, the <primkey-field> element specifies that the userid member variable is the primary key for the User bean.

JBoss built-in database

JBoss has a built-in SQL database, implemented in Java, which it uses for container-managed persistence by default. There is no need to start up the built-in database explicitly; it is launched at the same time as the JBoss Web server.

Default container-managed persistence

JBoss implements a default container-managed persistence, which requires no special configuration by the user. The default container-managed persistence has the following features:

- Persistence is managed by the JBoss JAWS (Just Another Web Storage) package, which implements object-relational mapping to generate database tables automatically from Java classes.
- Container-managed persistence is defined by the `ejb-jar.xml` file. No additional configuration is necessary.
- JAWS automatically creates a table to hold the container-managed persistence data (using the built-in SQL database).
- Table fields are created with default sizes. For example, a string field would automatically be allocated 256 bytes.

Customizing container-managed persistence

You can, optionally, customize container-managed persistence by providing a `jaws.xml` file with the EJB application. For example, the `jaws.xml` file allows you to specify the sizes of table fields and to use databases other than the JBoss built-in database.

For more details, consult the JAWS documentation from JBoss.

CHAPTER 7

J2EE Presentation Layer

The J2EE presentation layer is the front-end of an Internet application. It consists of web pages, Java server pages, worker beans, and miscellaneous supporting files (such as images and style sheets), all packaged within a single Web archive file.

In this chapter

This chapter discusses the following topics:

Overview of the Presentation Layer	page 150
Worker Beans	page 154
Using a JSP to Process a Web Form	page 161
Using a JSP to Access an Enterprise Bean	page 173

Overview of the Presentation Layer

Overview

Figure 37 shows an overview of the presentation layer for the AllDayBanking application. The presentation layer consists of a client, which is a Web browser, and the components on the server side that are directly responsible for generating Web pages. In particular, the J2EE presentation layer usually makes extensive use of Java Server Pages (JSP) technology.

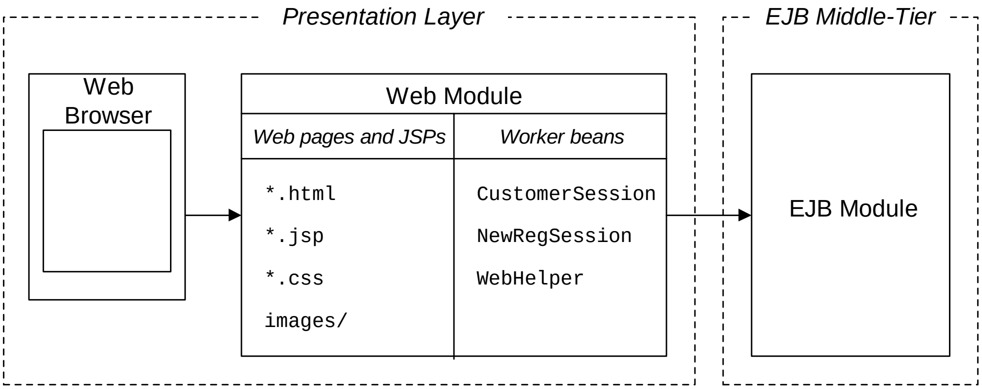


Figure 37: *Overview of the J2EE Presentation Layer for AllDayBanking*

Web module

A Web module contains all of the server-side components needed for the J2EE presentation layer. When a Web module, *WebModule*, is ready for deployment, the files in the module are usually zipped into a Web archive, *WebModule*`.war`—see “Directory Structure in a Web Module WAR File” on page 105. The Web archive itself can also be included in an EAR file—see “Directory Structure in an EAR File” on page 100.

The main components of a Web module are the following:

- Worker beans.
- Web pages and JSPs.

Worker beans

The worker beans in a Web module are ordinary Java beans (*not* enterprise Java beans) that are used in conjunction with JSPs to encapsulate part of the presentation logic.

The following directories are associated with worker beans:

WebModule`.war/src/`	EARSCO directory containing the worker bean source code.
`WEB-INF/classes/`	Directory in a Web archive containing the compiled worker bean code.

Web pages and JSPs

Web pages and JSPs are placed in the public part of a Web archive, which makes them directly accessible to client Web browsers.

The following directories are associated with Web pages and JSPs:

WebModule`.war/web/`	EARSCO directory containing the public Web files and directories.
Web archive top-level directory	The directory tree under the *WebModule*`.war/web/` EARSCO directory is copied to the Web archive's top-level directory.

Web browser

Files that are placed in the public part of a Web archive (that is, anything not under the WEB-INF directory) are directly accessible to client Web browsers.

The URL that clients use to access the public files is determined by the <context-root> element in the application.xml file. For example, the AllDayBanking application.xml file sets the <context-root> as follows:

```
<!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD
   J2EE Application 1.2//EN'
   'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>

<application>
  ...
  <module>
    <web>
      <web-uri>WebStuff.war</web-uri>
      <context-root>AllDayBanking</context-root>
    </web>
  </module>
</application>
```

With this setting, a client would use the following URL to access the index.html located in the Web archive's top-level directory:

http://*HostName*:8080/AllDayBanking/index.html

Where *HostName* is the name of the host where the J2EE application server is running (could be localhost if you run the client Web browser on the same host as the application server) and 8080 is the default IP port on which the JBoss J2EE application server is configured to run.

The J2EE application server also supports the standard Web server convention whereby the index.html can be omitted from the end of the URL. A client Web browser can then use the following shortened URL to access the index.html file:

http://*HostName*:8080/AllDayBanking

The file that is accessed by this shortened URL can be specified explicitly using the `<welcome-file-list>` element in the `web.xml` file. For example, the AllDayBanking `WebStuff.war/web.xml` file sets the `<welcome-file-list>` as follows:

```
...
<web-app>
  ...
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
  ...
</web-app>
```

Worker Beans

Overview

The AllDayBanking application provides the following worker beans, which cooperate with the JSPs to provide the presentation logic:

- `alldaybanking.web.CustomerSession`
- `alldaybanking.web.NewRegSession`
- `alldaybanking.web.WebHelper`

Using worker beans in conjunction with JSPs enables you to write simpler, more maintainable JSPs. Any lengthy bits of presentation logic can be put into a worker bean and then called from a JSP scriptlet. This enables the scriptlets inside a JSP to be kept relatively short and simple.

Bean attributes

Ordinary beans have the following noteworthy feature. Bean methods that conform to either of the following patterns are treated specially:

```
Type getAttributeName();
void setAttributeName(Type x);
```

where *AttributeName* is an attribute of *Type* type. The JavaBeans specification mandates that these methods are recognized as accessor and modifier methods for bean attributes. Various tools and utilities can then use Java reflection to identify the bean attributes automatically.

In this section

This section discusses the following Java classes:

The CustomerSession Bean	page 155
The NewRegSession Bean	page 158
The WebHelper Class	page 160

The CustomerSession Bean

Overview

The purpose of the CustomerSession bean is to provide support for user login over the Internet. The CustomerSession bean stores the user login details, user ID and password, and then validates the user identity by obtaining the user's details from a User entity bean in the EJB middle tier.

At subsequent stages during the user interaction with the AllDayBanking application, a JSP can check with the CustomerSession to confirm that the session remains valid.

Outline of the CustomerSession bean class

Example 21 gives an extract from the CustomerSession bean class, showing the bean attributes and method signatures, without the implementation code. The CustomerSession bean has several attributes, as represented by the methods of the form set*AttributeName*() and get*AttributeName*().

The bean attributes for the user ID and password, as represented by setUserid(), getUserid(), and setAccpwd(), are set automatically by a HTML form—see "The Login Web Form" on page 163 for details.

Example 21: *Extract from the CustomerSession Bean Class*

```
// Java
package alldaybanking.web;
...

public class CustomerSession implements java.io.Serializable {
  private String userid;
  private String accountPassword;
  private boolean isValid = false;
  private float ccamount;

  private alldaybanking.entity.User myUserBean;
  Exception exception;

  // Null constructor as required for a bean
  public CustomerSession () {
  }

  //--------------------
  // Bean attributes
  //--------------------
```

Example 21: *Extract from the CustomerSession Bean Class*

```
  public void   setUserid (String webuserid ) { ... }
  public String getUserid () { ... }

  public void   setAccpwd (String webAccountPassword ) { ... }
  // No getAccpwd(), that would create a bit of a security hole!

  public void   setAmount (float amount) { ... }
  public float  getAmount () { ... }

  public int getAccNum ( ) {
    // Delegate this call to the User entity bean (not shown)
    ...
  }

  public int getCcNum ()
  {
    // Delegate this call to the User entity bean (not shown)
    ...
  }

  //--------------------
  // Other bean methods
  //--------------------
  public boolean validateUser () { ... }

  public void isValidSession() throws SessionOverExceptio
  { ... }

  public void logout ()  { ... }
};
```

Validating the user identity

The main functionality offered by the CustomerSession bean is to validate the user identity, that is to check that the user-supplied ID and password are valid. Example 22 shows the implementation of the validateUser() method, which is responsible for validating the user's identity.

The implementation of validateUser() contacts the EJB middle-tier and searches for a User entity bean that matches the user-supplied ID, userid. The implementation then checks that the user-supplied password, accountPassword, matches the password from the User entity bean.

Example 22: *The validateUser() Method*

```
// Java
public class CustomerSession implements java.io.Serializable {
  ...
  public boolean validateUser () {
    try {
      InitialContext ctx = new InitialContext();
      UserHome uhome = (UserHome) PortableRemoteObject.narrow(
        ctx.lookup("java:comp/env/alldaybanking/User"),
        UserHome.class
      );
      myUserBean = uhome.findByPrimaryKey(userid);
    } catch (Exception ex) {
      exception = ex;
      return false;
    }

    String dbpwd;

    // Retrieve the password from the database
    try {
      dbpwd = myUserBean.getAccpwd ();
    } catch (Exception e) {
      System.out.println ("Exception "  + e );
      return false;
    }

    // Let's just make sure the passowrd is ok by comparing it
    // with what the user has supplied
    if ( accountPassword.equals(dbpwd) ) {
      isValid = true;
      return true;
    } // end of if ()

    return false;
  }
  ...
};
```

The NewRegSession Bean

Overview

The purpose of the NewRegSession bean is to enable new users to register with the AllDayBanking application. The NewRegSession bean receives the user's registration details from a HTML form and then registers the user by creating a new User entity bean in the EJB middle tier.

Outline of the NewRegSession bean class

Example 23 gives an extract from the NewRegSession bean class, showing the bean attributes and method signatures, without the implementation code. The NewRegSession bean has several *attributeName* attributes, as represented by the methods of the form set*AttributeName*() and get*AttributeName*().

All of the NewRegSession bean attributes are set automatically by the New User Registration Web form—see "The New User Registration Web Form" on page 167 for details.

Example 23: *Extract from the NewRegSession Bean Class*

```
// Java
package alldaybanking.web;
...

public class NewRegSession implements Serializable {
  private String lastname;
  private String firstname;
  private String userid;
  private int accountNumber;
  private int creditcardNumber;
  private String passwordOne;
  private String passwordTwo;
  private String emailAddress;

  //--------------------
  // Bean attributes (set by Web form)
  //--------------------
  public void   setFname (String fn) { ... }
  public String getFname ( ) { ... }

  public void   setLname (String ln ) { ... }
  public String getLname () { ... }
```

Example 23: *Extract from the NewRegSession Bean Class*

```
  public void   setUserid (String id) { ... }
  public String getUserid () { ... }

  public void   setAccnum (int accnum) { ... }
  public int    getAccnum () { ... }

  public void   setCcnum (int ccnum) { ... }
  public int    getCcnum () { ... }

  public void   setEmailaddr (String addr) { ... }
  public String getEmailaddr () { ... }

  public void   setAccpwdone (String pwd) { ... }
  public String getAccpwdone ( ) { ... }

  public void   setAccpwdtwo (String pwd) { ... }
  public String getAccpwdtwo () { ... }

  // Null constructor as required for a bean
  public NewRegSession () {
  }

  //--------------------
  // Other bean methods
  //--------------------
  public void addUser () throws UserAlreadyExistsException,
   AccountValidationException
  { ... }
}
```

Adding the user to the database

The `NewRegSession.addUser()` method is responsible for registering a new user by creating a new `User` entity bean in the EJB middle tier to represent the registered user. The implementation of this method is not shown here.

For an example of how a worker bean can contact the EJB middle tier, see the implementation of the `CustomerSession.validateUser()` method in "Validating the user identity" on page 156.

The WebHelper Class

Overview

The WebHelper class declares static methods that return references from beans in the EJB middle tier. This provides JSPs with a quick and easy way of accessing enterprise beans in the EJB middle tier.

Getting a reference to an InetAccount enterprise bean

Example 24 gives the implementation of the WebHelper.getInetAccount() static method, which creates and returns a reference to an InetAccount session bean from the middle tier.

Example 24: *Implementation of the getInetAccount() Method*

```
// Java
package alldaybanking.web;
...

public class WebHelper implements Serializable {

  ...
  public static InetAccount getInetAccount ()
  {
    InetAccount InetAccountObject = null;

    try {
      InitialContext ctx = new InitialContext();
      InetAccountHome vhome
        = (InetAccountHome) PortableRemoteObject.narrow (
          ctx.lookup("java:comp/env/alldaybanking/InetAccount"),
          InetAccountHome.class
        );
      InetAccountObject = vhome.create();
    } catch (Exception ex) {
      exception = ex;
    }
    return InetAccountObject;
  }

}
```

Using a JSP to Process a Web Form

Overview

One of the common uses for a JSP is to process the data from a HTML Web form and generate an appropriate response. This section presents two examples from the AllDayBanking application, the *login* Web form and the *new user registration* Web form, that show how to process a Web forms using JSP.

Overview of Web form processing

Figure 38 shows the typical interaction between a Web form, JSP, and a worker bean as the JSP processes the Web form data.

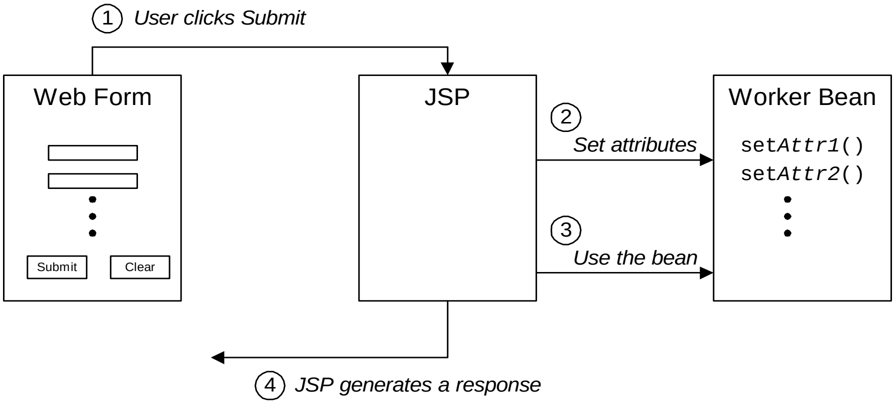


Figure 38: *Processing Web Form Data Using a JSP*

Stages of Web form processing

The stages shown in Figure 38 can be explained as follows:

Stage	Description
1	When a user clicks the **Submit** button on the Web form, the form data is sent to a particular JSP using the HTTP protocol.
2	The JSP uses the `<jsp:useBean>` and `<jsp:setProperty>` tags to send the form data to the worker bean. See "Processing the form action" on page 165 for more details.
3	The JSP uses methods defined on the worker bean to help it process the form data.
4	Based on the results of processing the form data, the JSP generates a response (either generating HTML directly or forwarding to a different page).

In this section

This section describes how the following Web forms are processed:

The Login Web Form	page 163
The New User Registration Web Form	page 167

The Login Web Form

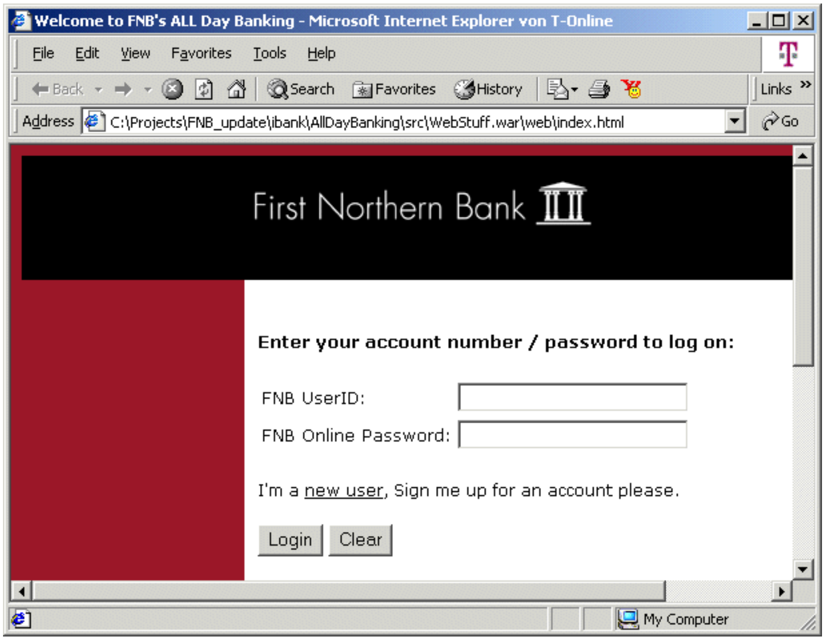
Overview

When a user initially connects to the AllDayBanking application (by linking to `http://`*HostName*`:8080/AllDayBanking/`), the user is presented with a login form. After the user clicks the **Submit** button, the form is processed by the `main.jsp` JSP working in conjunction with the `CustomerSession` worker bean.

The Login page

Figure 39 shows the first page of the AllDayBanking application, which consists of a HTML Web form that prompts the user for the following login data:

- **FNB UserID**
- **FNB Online Password**


Welcome to FNB's ALL Day Banking - Microsoft Internet Explorer von T-Online
File Edit View Favorites Tools Help
Back Search Favorites History Links
Address C:\Projects\FNB_update\ibank\AllDayBanking\src\WebStuff.war\web\index.html
Go
First Northern Bank

Enter your account number / password to log on:

FNB UserID:

FNB Online Password:

I'm a new user, Sign me up for an account please.

Figure 39: *The Login Page of the AllDayBanking Application*

The form HTML source

Example 25, which is an extract from the AllDayBanking `index.html` file, gives the HTML source for the Login Web form depicted in Figure 39 on page 163.

The form defines two input fields, `userid` and `accpwd`, and specifies the form action to be `main.jsp`.

Example 25: *Web Form from the AllDayBanking index.html File*

```
<html>
...
<FORM ACTION="main.jsp" METHOD="post">
<P>
 <FONT SIZE="3">
  <B>Enter your account number / password to log on:</B>
 </FONT>
<P>

<TABLE BORDER="0">
 <TR>
  <TD>FNB UserID:</TD>
  <TD ALIGN="left">
    <INPUT TYPE="text" SIZE="25" NAME="userid">
  </TD>
 </TR>
 <TR>
  <TD>FNB Online Password:</TD>
  <TD ALIGN="left">
    <INPUT TYPE="password" SIZE="25" NAME="accpwd" >
  </TD>
 </TR>
</TABLE>

  <P>I'm a <a href="NewUser.jsp">new user</a>, Sign me up for an
   account please.
  </P> <P>
    <INPUT TYPE="submit" VALUE="Login">
    <INPUT TYPE="reset" VALUE="Clear">
  </P>
</FORM>
...
</html>
```

Processing the form action

When the user clicks **Submit** on the Web form, the form data, `userid` and `accpwd`, is posted to `main.jsp` (the specified action for the form). Example 26 shows the JSP script from the `main.jsp` file, which is responsible for processing the form data.

Example 26: *The AllDayBanking main.jsp File*

```
   <!-- JSP -->

   <%@ page info = "Validating user details..." %>
   <%@ page language = "java" %>

   <%@ page import = "alldaybanking.web.WebHelper" %>
   <%@ page import = "alldaybanking.session.validate" %>

1  <jsp:useBean
       id = "inetSession"
       class = "alldaybanking.web.CustomerSession"
       scope = "session">
   </jsp:useBean>

2  <jsp:setProperty name="inetSession" property="*"/>

   <HTML>
     <HEAD>
       <TITLE>Welcome to FNB's All Day Banking</TITLE>
     </HEAD>

   <%
       // Right, before anything happens, we need to validate that
      this userid
       // password combination is valid
3     if (inetSession.validateUser() == false) {
4         response.sendRedirect ("NotRegistered.html");
       } else { %>
5           <jsp:forward page="Ledger.jsp"/>;
      <%
      }
   %>
   </HTML>
```

The preceding JSP script can be explained as follows:

1. The `<jsp:useBean>` tag establishes a reference to a `CustomerSession` worker bean. The `CustomerSession` bean instance can be accessed throughout this script using the `inetSession` handle.
2. The `<jsp:setProperty>` tag sends all of the form data to the `CustomerSession` bean (identified by its handle, `inetSession`). This tag uses Java reflection to match the `userid` and `accpwd` form properties to the corresponding `setUserid()` and `setAccpwd()` attribute methods defined on `CustomerSession`.
3. The JSP calls `validateUser()` on the `CustomerSession` bean (represented as `inetSession`) to verify that the user ID and password are correct.
4. The `response` object is the `javax.servlet.http.HttpServletResponse` object that is associated with this page. The `response` identifier is implicitly defined for every JSP.
5. The `<jsp:forward>` action enables the HTTP request to be forwarded to another HTML page, JSP, or servlet.

The New User Registration Web Form

Overview

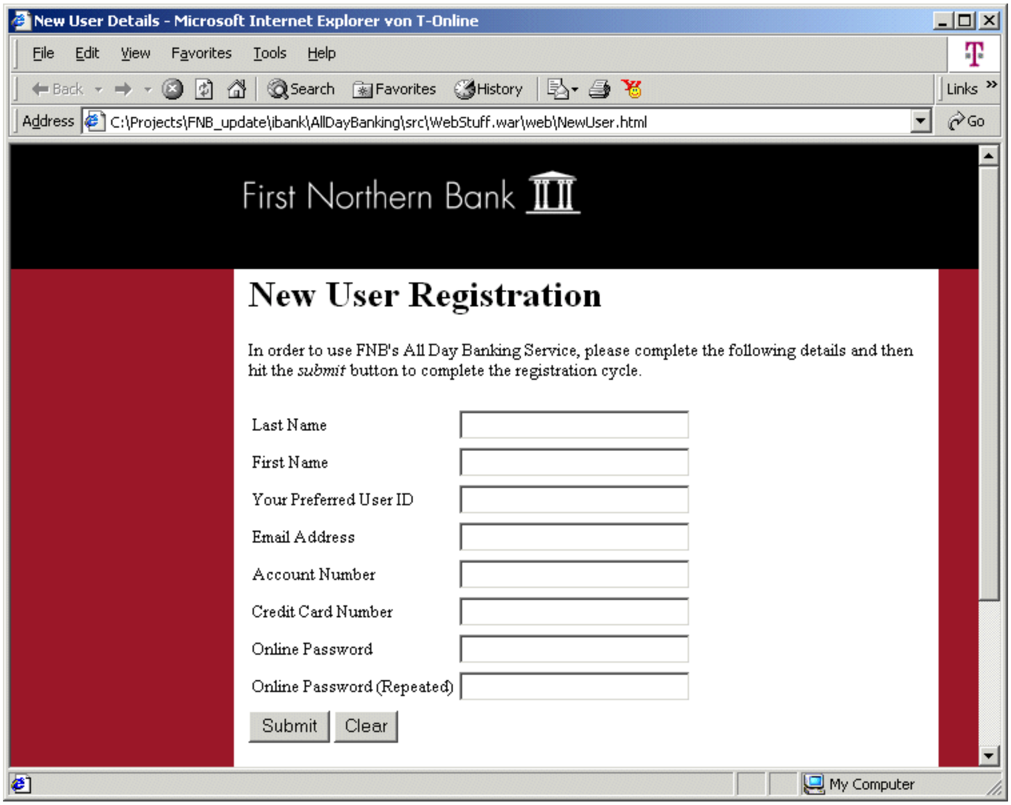
If a user is about to use the AllDayBanking application for the first time, the user can follow the **new user** link on the AllDayBanking home page (see Figure 39 on page 163) to arrive at the **New User Registration** Web form.

After the user fills in the registration details and clicks the **Submit** button, the form is processed by the `register.jsp` JSP working in conjunction with the `NewRegSession` worker bean.

The New User Registration page

Figure 40 shows the **New User Registration** page of the AllDayBanking application, which consists of a HTML Web form that prompts the user for the following registration data:

- **Last Name**
- **First Name**
- **Your Preferred UserID**
- **Email Address**
- **Account Number**
- **Credit Card Number**
- **Online Password**
- **Online Password (Repeated)**



New User Details - Microsoft Internet Explorer von T-Online
File Edit View Favorites Tools Help
Back Search Favorites History
Links
Address C:\Projects\FNB_update\ibank\AllDayBanking\src\WebStuff.war\web\NewUser.html
Go
First Northern Bank
My Computer

New User Registration

In order to use FNB's All Day Banking Service, please complete the following details and then hit the *submit* button to complete the registration cycle.

Last Name	
First Name	
Your Preferred User ID	
Email Address	
Account Number	
Credit Card Number	
Online Password	
Online Password (Repeated)	

Submit Clear

Figure 40: *The New User Registration Page of the AllDayBanking Application*

The form HTML source

Example 27, which is an extract from the AllDayBanking `NewUser.jsp` file, gives the HTML source for the **New User Registration** Web form depicted in Figure 40 on page 168.

The form defines several input fields containing registration data and specifies the form action to be `register.jsp`.

Example 27: *Web Form from the AllDayBanking NewUser.jsp File*

```
<html>
...
<FORM ACTION="register.jsp" METHOD="post">

<h1>New User Registration</h1>
<p>In order to use FNB's All Day Banking Service, please complete
the following details and then hit the <i>submit</i> button to
complete the registration cycle.</p>

<TABLE BORDER="0">
 <TR>
  <TD>Last Name</TD>
  <TD ALIGN="left">
    <INPUT TYPE="text" SIZE="25" NAME="lname">
  </TD>
 </TR>
 <TR>
  <TD>First Name</TD>
  <TD ALIGN="left">
    <INPUT TYPE="text" SIZE="25" NAME="fname" >
  </TD>
 </TR>
 <TR>
  <TD>Your Preferred User ID</TD>
  <TD ALIGN="left">
    <INPUT TYPE="text" SIZE="25" NAME="userid" >
  </TD>
 </TR>
 <TR>
  <TD>Email Address</TD>
  <TD ALIGN="left">
    <INPUT TYPE="text" SIZE="25" NAME="emailaddr" >
  </TD>
 </TR>
 <TR>
  <TD>Account Number</TD>
  <TD ALIGN="left">
```

Example 27: *Web Form from the AllDayBanking NewUser.jsp File*

```
    <INPUT TYPE="text" SIZE="25" NAME="accnum" >
  </TD>
 </TR>
 <TR>
  <TD>Credit Card Number</TD>
  <TD ALIGN="left">
    <INPUT TYPE="text" SIZE="25" NAME="ccnum" >
  </TD>
 </TR>
 <TR>
  <TD>Online Password</TD>
  <TD ALIGN="left">
    <INPUT TYPE="password" SIZE="25" NAME="accpwdone" >
  </TD>
 </TR>
 <TR>
  <TD>Online Password (Repeated)</TD>
  <TD ALIGN="left"> <INPUT TYPE="password" SIZE="25"
   NAME="accpwdtwo" >
  </TD>
 </TR>
</TABLE>

<INPUT TYPE="submit" VALUE="Submit">
<INPUT TYPE="reset" VALUE="Clear">

</FORM>
...
</html>
```

Processing the form action

When the user clicks **Submit** on the Web form, the form data is posted to `register.jsp` (the specified action for the form). Example 28 shows the JSP script from the `register.jsp` file, which is responsible for processing the form data.

Example 28: *The AllDayBanking register.jsp File*

```
   <%@page contentType="text/html"%>
   <html>
   <head><title>New User Registration Details</title></head>
   <body>

1  <jsp:useBean
       id = "regSession"
       class = "alldaybanking.web.NewRegSession"
       scope = "session">
   </jsp:useBean>

2  <jsp:setProperty name="regSession" property="*"/>

   <%

   try {
3    regSession.addUser() ;
     } catch (alldaybanking.web.UserAlreadyExistsException uae) {
     %>
       <H1>Sorry</H1>
       <P>Your account was not created.</P>
       <P>This user ID already exists.</P>
       <P>Please <A href="/AllDayBanking/NewUser.jsp">try
      again</A>.</P>
     <%
      return;
     } catch (alldaybanking.web.AccountValidationException ex) {
     %>
       <H1>Sorry</H1>
       <P>Your account was not created.</P>
       <P><%=ex%></P>
       <P>Please <A href="/AllDayBanking/NewUser.jsp">try
      again</A>.</P>
     <%
      return;
     } %>

     <H1>Welcome</H1>
```

Example 28: *The AllDayBanking register.jsp File*

```
  <P>Your account has been created.</P>
  <P>Please log in at <A href="/AllDayBanking"
   target="_top">AllDayBanking</A>. </P>
</body>
</html>
```

The preceding JSP script can be explained as follows:

1. The `<jsp:useBean>` tag establishes a reference to a `NewRegSession` worker bean. The `NewRegSession` bean instance can be accessed throughout this script using the `regSession` handle.
2. The `<jsp:setProperty>` tag sends all of the form data to the `NewRegSession` bean (identified by its handle, `regSession`). This tag uses Java reflection to match each of the form properties to the corresponding attribute `set` methods defined on `NewRegSession` (see "The NewRegSession Bean" on page 158).
3. The JSP calls `addUser()` on the `NewRegSession` bean to create a new `User` entity bean in the EJB middle tier for this user.

Using a JSP to Access an Enterprise Bean

Overview

In addition to accessing worker beans, a JSP can also access enterprise beans in the EJB middle tier directly. For example, this section describes the AllDayBanking `PayBill.jsp` script which accesses an `InetAccount` session bean.

The PayBill JSP

After a user has logged in and gained access to an account, the AllDayBanking application presents the user with a menu of actions to perform. One of the available actions is to pay a credit card bill out of funds from the user's account. The `PayBill.jsp` script implements the first step of this action.

Accessing the InetAccount enterprise bean

Example 29 shows the JSP script from the `PayBill.jsp` file, which checks the balance remaining in the user's account and presents the user with a simple form to fill in.

Example 29: *The AllDayBanking PayBill.jsp File*

```
<!-- JSP -->
<%@ page language = "java" %>

<%@ page import = "alldaybanking.web.WebHelper" %>
<%@ page import = "alldaybanking.session.validate" %>
<%@ page import = "alldaybanking.session.InetAccount" %>

<jsp:useBean
    id = "inetSession"
    class = "alldaybanking.web.CustomerSession"
    scope = "session">
</jsp:useBean>

<HTML>
  <HEAD>
    <link rel="STYLESHEET" type="text/css" href="layout.css"/>
  </HEAD>

  <%
    try {
       inetSession.isValidSession ();
```

Example 29: *The AllDayBanking PayBill.jsp File*

```
    } catch (alldaybanking.web.SessionOverException ex ) { %>
       <jsp:forward page="SessionExpired.html"/>
    <%}
    java.text.DecimalFormat df2
      = new java.text.DecimalFormat("###,###,##0.00");
1   InetAccount iacc = WebHelper.getInetAccount ();
  %>
<BODY>
  <H3>Credit Card Bill Payment:</H3>
  <P>How much do you want to pay onto your Credit Card? </P>

  <FORM ACTION="ConfirmPay.jsp" METHOD="post">
  <INPUT TYPE="text" SIZE="25" NAME="amount"> </TD>

2 <P>Max. value you can clear is <%=
      df2.format(iacc.getBalance(inetSession.getAccNum()))
    %> </P>

  <INPUT TYPE="submit" VALUE="Pay Bill">
  </FORM>

  </BODY>
</HTML>
```

The preceding JSP script can be explained as follows:

1. The `alldaybanking.web.WebHelper` class defines a static method, `getInetAccount()`, that creates a new `InetAccount` session bean in the EJB middle tier and returns a remote reference, `iacc`. See “The WebHelper Class” on page 160.
2. The `getBalance()` method is invoked on the remote `InetAccount` session bean to obtain the balance on the user’s account.

Part III

COMet and .NET Clients

In this part

This part contains the following chapters:

Visual Basic COMet Client	page 177
C# .NET Client	page 191

CHAPTER 8

Visual Basic COMet Client

The FNB demonstration includes a simulation of an Automated Teller Machine (ATM), which is implemented in Visual Basic. The ATM client is implemented using DCOM automation and access to CORBA servers is provided through COMet (IONA's implementation of a COM/CORBA bridge).

In this chapter

This chapter discusses the following topics:

Overview of the Visual Basic Client	page 178
Implementation of the Visual Basic Client	page 182

Overview of the Visual Basic Client

Overview

Figure 41 shows the architecture of the Visual Basic ATM client application. The Visual Basic client communicates with the CORBA mid-tier server and the CORBA back-end server, using IONA's COMet to bridge between DCOM and CORBA.

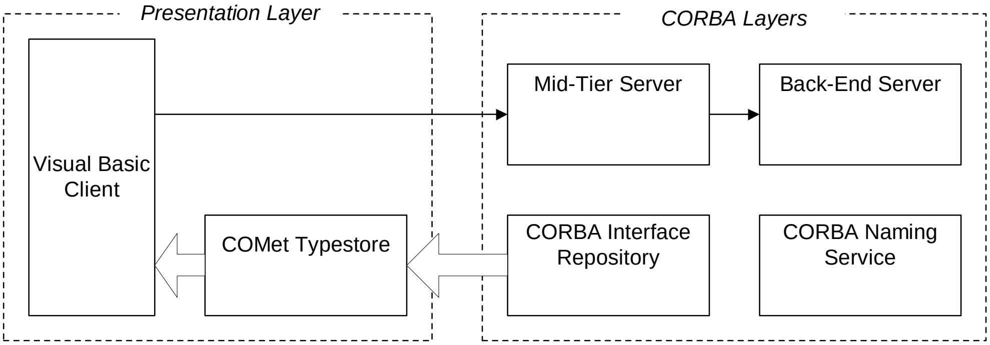


Figure 41: *Architecture of the Visual Basic ATM Client Application*

Visual Basic Client

The ATM demonstration is implemented as a Visual Basic client, which is augmented by the COMet libraries and interfaces. IONA's COMet acts as a bridge between the Visual Basic client and the CORBA servers in the mid-tier and back-end. The Visual Basic automation client accesses the CORBA servers with the help of the type information cached in the COMet typestore.

COMet typestore

The COMet typestore must be populated by the types obtained from the `fnbba` and `bankobjects` OMG IDL modules. The automation client cannot bind to the FNB CORBA servers or use any of the CORBA data types unless the COMet typestore is populated.

For a particular Orbix configuration domain, *Domain*, the files that comprise the COMet typestore are located in the following directory:

OrbixInstallDir`\var\`*Domain*`\dbs\COMet`

If an automation client cannot find the types it needs in the COMet typestore, the COMet typestore automatically attempts to load the required types from the CORBA interface repository (IFR).

CORBA interface repository

The IFR is a CORBA-specific type repository. In general, you can populate the IFR using the Orbix `idl` compiler utility. For example:

`idl -R` *IDLFile*`.idl`

The `fnb\build.xml` ant build file provides a `populate_ifr` target to register the demonstration IDL files.

CORBA naming service

Visual Basic clients can use the COMet API to look up CORBA object references in the naming service. For example, in this demonstration the ATM client looks up the `FNBBA_BusinessSessionManager` name in order to bind to the `fnbba::BusinessSessionManager` object in the mid-tier server.

Starting the ATM demonstration

You can run the ATM client demonstration as follows:

1. Make sure that the basic Orbix services, FNB back-end (`itant start_backend`) and FNB mid-tier (`itant start_fnbba`) are all running.
2. If the COMet typestore is not already primed, you need to populate the IFR with the relevant IDL interfaces. Do this by invoking the following `ant` target from the *OrbixInstallDir*`\asp\6.1\demos\fnb` directory:

   ```
   itant populate_ifr
   ```

3. Run the ATM Visual Basic client as follows:

   ```
   cd OrbixInstallDir\asp\6.1\demos\common\fnb\atm
   atm.exe
   ```

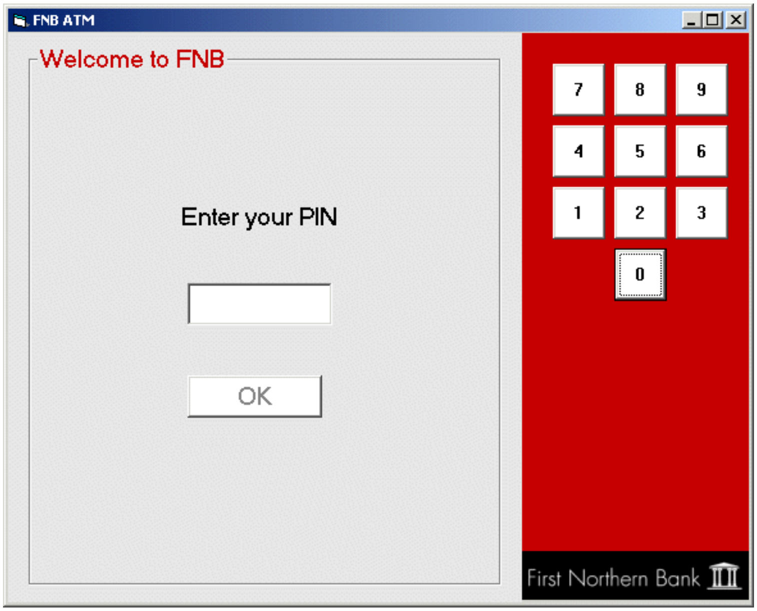


Figure 42: *The ATM Client Welcome Screen*

ATM demonstration session

A typical ATM client session consists of the following steps:

1. Start the ATM session—when you run `atm.exe`, the welcome screen appears as shown in Figure 42.

   Normally, if you were using a real ATM, the machine would know which account you want to access as soon as you insert your card. The ATM client simulates this behavior by picking an account implicitly (the first in the list), instead of asking you for an account number.

2. Validate the PIN—you must enter a four-digit PIN before you can proceed. In this demonstration, the PIN is not checked, but it must be four digits long.

3. Show account details—the ATM client contacts the back-end server to retrieve the account balance and the list of recent transactions for this account.
4. Withdraw cash—the ATM client debits the specified amount from the customer's account in the back-end.

Implementation of the Visual Basic Client

Overview

This section presents some code extracts from the `ATMForm.frm` file, discussing aspects of the code that are relevant to CORBA programming in Visual Basic.

Location of the demonstration code

The ATM Visual Basic client code is located in the following directory:

OrbixInstallDir`\asp\`*Version*`\demos\common\fnb\atm`

In this section

This section contains the following subsections:

Starting the ATM Session	page 183
Showing Account Details	page 186
Withdrawing Cash	page 188

Starting the ATM Session

Overview

This section describes the Visual Basic subroutine, `Form_Load()`, that runs during start-up to initialize the ATM client application.

This example shows you how to use the COMet API to bind to remote CORBA objects—for example, by looking up object references in the CORBA naming service. Also, this example shows you how to *narrow* a base OMG IDL interface type to a derived interface type.

Form_Load subroutine

The `Form_Load()` subroutine from the `ATMForm.frm` file is defined in Example 30.

Example 30: *The ATM Form_Load() Subroutine*

```
   Private Sub Form_Load()

     ' Set up the ORB

1    Set objORB = CreateObject("CORBA.ORB.2")
2    Set objFact = CreateObject("CORBA.Factory")
     If RunningInIde Then
3      objORB.RunningInIde
     End If
4    Dim objSessMgr As Object, objSessType As Object
5    Set objSessMgr =
      objFact.GetObject("fnbba/BusinessSessionManager:NAME_SERVICE:
      FNBBA_BusinessSessionManager")

     ' Get an ATM session

6    Set objSessType = objFact.createtype(
                           Nothing,
                           "fnbba/SessionInfo_s"
                       )
     objSessType.username = "ATMUser"
     objSessType.password = "kj8yhj"
     objSessType.session_type = "ATM"
     objSessType.client_id = "ATM" & Rnd(200)
7    Set objSess = objSessMgr.openSession(objSessType)

     ' It returns a generic session so convert it into the ATM
      Session object
```

Example 30: *The ATM Form_Load() Subroutine*

```
     Dim ior As String
8    ior = objORB.objecttostring(objSess)
9    Set objSess = objFact.GetObject("fnbba/ATMSession:" & ior)

     ' Simulate the swiping of a card by picking the first
     ' current account listed
     Dim accts
     accts = objSess.getAccountList("Current")
     accNo = accts(0)

     ' Ask for the PIN

     showPINFrame
   End Sub
```

The preceding Visual Basic subroutine can be explained as follows:

1. This line creates a `CORBA::ORB` object (defined by the `DIOrbixORBObject` automation interface), which the client application can use to control certain properties of the ORB. `CORBA.ORB.2` is the standard Automation/CORBA-compliant ProgID for the local ORB object.
2. This line creates a new CORBA factory object (defined by the `DICORBAFactory` and `DICORBAFactoryEx` automation interfaces). The CORBA factory object is used to create new object references that bind to remote CORBA objects. `CORBA.Factory` is the standard Automation/CORBA-compliant ProgID for the CORBA factory.
3. The `RunningInIDE` method changes the internal shutdown policy, so COMet can run in the Visual Basic studio debugger.
4. This line allocates space for two CORBA objects references, as follows:
   - `objSessMgr`—a session manager object, which is an instance of the `fnbba::BusinessSessionManager` OMG IDL interface.
   - `objSessType`—a structure data type, which is an instance of the `fnbba::SessionInfo_s` OMG IDL data type.
5. This line contacts the CORBA naming service to obtain a reference to a business session manager object. The string argument to `GetObject()` has the following format:

CORBATypeID`:NAME_SERVICE:`*ObjectName*
Where *CORBATypeID* is the scoped name of the IDL type, using `/` instead of `::` as the scope separator; `NAME_SERVICE` indicates that you want to look up the object in the CORBA naming service; and *ObjectName* is the name of the object in the naming service.

6. The `CreateType()` method is used to create an instance of an OMG IDL complex type.

   The first parameter indicates the scope with respect to which the second parameter is interpreted. Global scope is indicated by passing the `Nothing` parameter. The second parameter is the scoped name of the IDL type, using `/` instead of `::` as the scope separator
7. This line calls the `fnbba::BusinessSessionManager::openSession()` IDL operation to create a new user session on the middle-tier server. The return value, `objSess`, is a session object of `fnbba::BusinessSession` type, which is the base type for a user session.
8. Before you can use the session object, `objSess`, it must be narrowed (or cast) to the type, `fnbba::ATMSession`, which derives from the `fnbba::BusinessSession` IDL interface.

   The first step is to convert `objSess` into a stringified Interoperable Object Reference (IOR), by calling the `ObjectToString()` method on the ORB with `objSess` as the argument.
9. The user session is now converted to an object of `fnbba::ATMSession` type by calling the `GetObject()` method on the CORBA factory. The argument to `GetObject()` has the following form:

   DerivedCORBATypeID`:`*StringifiedIOR*
   Where *DerivedCORBATypeID* is the type ID of the derived type that you want to narrow to. The *StringifiedIOR* consists of `IOR:` followed by a long sequence of two-digit hexadecimal numbers (essentially, a hex dump of the IOR’s contents).

Showing Account Details

Overview

This section describes the Visual Basic subroutine, `showDetsFrame()`, that retrieves a customer's account transaction history from the CORBA back-end server.

This example illustrates how a complex OMG IDL type maps to Visual Basic. The transaction list is represented as an array of structures in Visual Basic.

showDetsFrame subroutine

The `showDetsFrame()` subroutine from the `ATMForm.frm` file is defined in Example 31.

Example 31: *The ATM showDetsFrame() Subroutine*

```
   Private Sub showDetsFrame()
1    Dim txns
     Dim txno As Integer
     FramePIN.Visible = False
     FrameAction.Visible = False
     FrameDets.Visible = True
     FrameWithdraw.Visible = False
2    txtBal.Text = acc.accountbalance
     lstTxn.Clear
3    txns = acc.recentTransactions
4    For txno = UBound(txns) To 0 Step -1
       lstTxn.AddItem txns(txno).Date & " - " &
      txns(txno).record_type + " - " & txns(txno).Value
     Next txno
   End Sub
```

The preceding Visual Basic subroutine can be explained as follows:

1. This line allocates an object, `txns`, which will be used to hold the complex CORBA type, `bankobjects::AccountTransactions`.
2. This line invokes the remote `bankobjects::Account::accountBalance` attribute on the CORBA back-end server.
3. This line invokes the remote `bankobjects::Account::recentTransactions()` operation, with a return value of `bankobjects::AccountTransactions` type (an IDL sequence).

4. The `txns` object is an array of structures in Visual Basic. It is derived from the `AccountTransactions` OMG IDL type, defined as follows:

```
// IDL
...
module bankobjects {
  ...
  struct BankTransaction {
    short id;
    string date;
    string record_type;
    string value;
  };

  typedef sequence<BankTransaction> AccountTransactions;
  ...
};
```

Withdrawing Cash

Overview

This section describes the Visual Basic subroutine, `withdraw()`, that implements withdrawing cash from a customer's current account.

This example illustrates how to handle exceptions raised by a remote CORBA server.

withdraw subroutine

The `withdraw()` subroutine from the `ATMForm.frm` file is defined as Example 32.

Example 32: *The ATM withdraw() Subroutine*

```
  Private Sub withdraw(amount As Integer)
1   On Error Resume Next
2   acc.withdrawfunds (amount)
    ' check if there was an error
3   If Err.Description = "CORBA User Exception
     :[bankobjects::INSUFFICIENT_FUNDS]" Then
      MsgBox "Insufficient funds to withdraw"
      Exit Sub
    End If
    If Err.Number = 0 Then
      MsgBox "Please take your cash"
    Else
      MsgBox "Communication error"
    End If
    showActionFrame
  End Sub
```

The preceding Visual Basic subroutine can be explained as follows:

1. Because the remote operation is liable to throw an exception, this line instructs the application to catch the error locally.
2. Invoke the remote IDL operation, `withdrawFunds()`, on the `bankobjects::Account` IDL interface. This operation will throw an exception, if the amount to withdraw exceeds the customer's overdraft limit.
3. This line checks for a CORBA user exception. The `Err.Description` string for CORBA user exceptions has the following format:

   `"CORBA User Exception :[`*ScopedExceptionName*`]"`

Where *ScopedExceptionName* is the scoped exception name in OMG IDL syntax (that is, using `::` as the scope separator).

CHAPTER 9

C# .NET Client

The FNB demonstration includes a Web services application that simulates making credit card purchases online. Complimentary to this, FNB provides a simple C# client implemented using .NET technology that allows an administrator to monitor the list of registered merchants using the service.

In this chapter

This chapter discusses the following topics:

Overview of the C# Client	page 192
Implementation of the C# Client	page 196

Overview of the C# Client

Overview

Figure 43 shows the architecture of the C# online purchasing client application. The interface to e-commerce clients is exposed as a Web service over SOAP/HTTP. Merchants use this Web interface to register themselves and transact online purchases. The C# client is a monitoring utility that lists merchant details and is intended as an aid for Web site administrators.

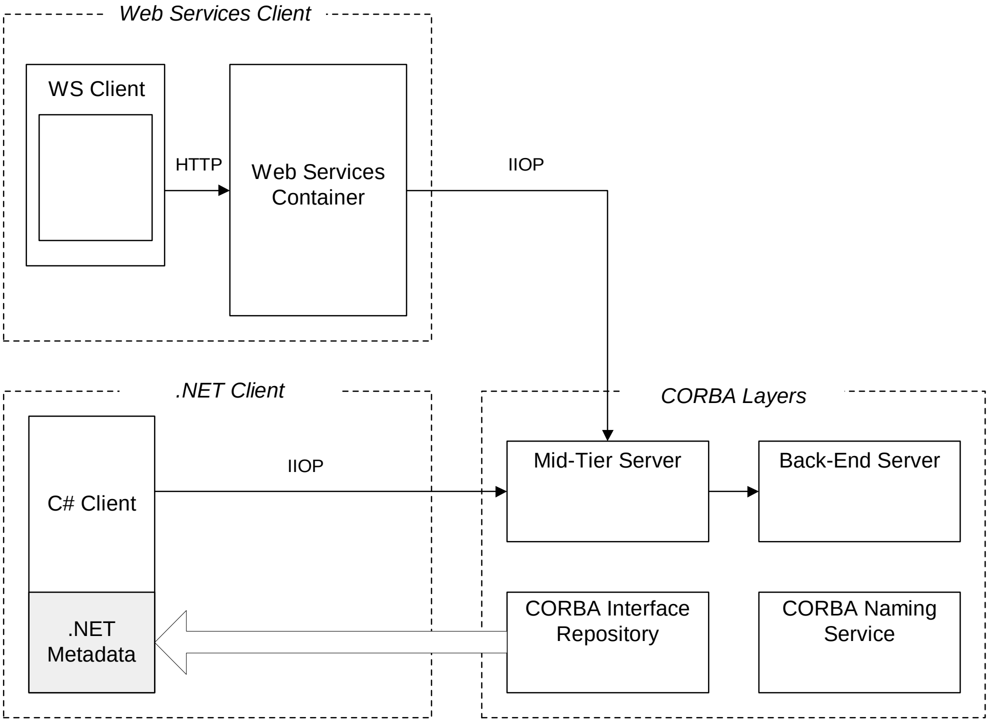


Figure 43: *Architecture of the C# Online Purchasing Client Application*

Web services client

The Web services client is a browser-based client that can be used to register merchants and make purchases online. The online purchasing Web service is intended to be used by e-commerce companies (that is, *merchants*) that sell goods online by debiting a customer's credit card.

For complete details of how to build and run the Web services application, see the *First Northern Bank Tutorial*.

C# .NET client

The C# .NET client is a simple utility that lists details of the merchant accounts currently registered with the online purchasing manager. The Orbix .NET connector technology is used to bridge between the C# .NET client and the mid-tier CORBA server.

CORBA interface repository

The interface repository (IFR) is a CORBA-specific type repository. In general, you can populate the IFR using the Orbix `idl` compiler utility. For example:

```
idl -R IDLFile.idl
```

.NET metadata

The .NET metadata must be populated by the types obtained from the `fnbba` and `bankobjects` OMG IDL modules. The .NET client cannot bind to the FNBBA server or use any of the CORBA data types unless the .NET metadata is populated.

For example, you can populate the .NET metadata with the types from the `fnbba` and `bankobjects` modules as follows:

```
idl -R BusinessSessionManager.idl Account.idl
itts2il fnbba bankobjects
```

The first command populates the CORBA interface repository with the `fnbba` and `bankobjects` type definitions. The second command populates the .NET metadata with all of the type definitions from the CORBA interface repository, producing a single DLL file:

`fnbba.dll`

This file is called a *.NET metadata assembly*. It is packaged in the form of a DLL file and contains the MSIL type definitions derived from the `fnbba` and `bankobjects` OMG IDL modules.

CORBA naming service

The C# .NET client uses the .NET remoting API to look up CORBA object references in the naming service. For example, in this demonstration the .NET client looks up the `FNBBA_BusinessSessionManager` name in order to bind to the `fnbba::BusinessSessionManager` object in the mid-tier server.

Prerequisites for developing

If you are planning to develop C# .NET applications, you need at least the Microsoft .NET Framework 1.1 and Microsoft Visual Studio .NET 2003 installed on your machine.

Running the C# .NET client

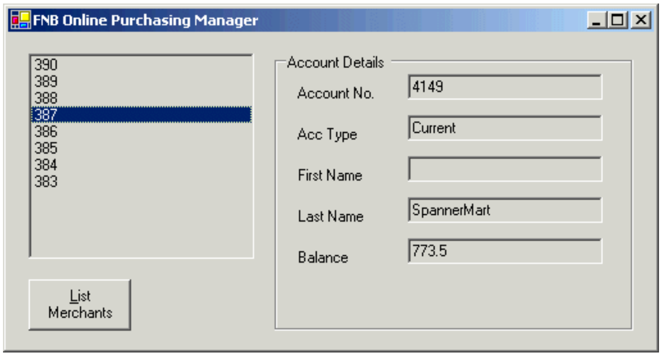
In order to run the C# .NET client executable, the following prerequisites must be satisfied:

- You have the Microsoft .NET Framework 1.1 installed on your machine (available from http://windowsupdate.microsoft.com/).
- The .NET metadata has been primed with the types mapped from the `fnbba` IDL module.
- The requisite .NET metadata assemblies are on your path:
    - `fnbba.dll`
    - `Orbix.Remoting.dll`
- The following Visual C++ runtime DLLs must be on your path:
    - `msvcr71.dll`
    - `msvcp71.dll`

You can then run the C# .NET client from the `\fnb\onlinepurchasingmanager\onlinepurchasingmanager\bin\Release` directory, as follows:

```
onlinepurchasing.exe
```

Assuming that you have already registered a few merchant accounts using the Web services client, you will see a GUI window similar to Figure 44.



Account No. 4149

Acc Type Current

First Name

Last Name SpannerMart

Balance 773.5

Figure 44: *The Online Purchasing Manager C# Client*

Implementation of the C# Client

Overview

This section describe the basic steps required to develop a C# client that uses the Orbix .NET connector technology. The code extracts in this section are taken from the `Form1.cs` file.

Location of the demonstration code

The online purchasing manager client code is located in the following directory:

OrbixInstallDir`\asp\`*Version*`\demos\common\fnb\onlinepurchasingmanager`
    `\onlinepurchasingmanager\`

In this section

This section contains the following subsections:

Importing .NET Metadata	page 197
Initializing the Online Purchasing Manager Client	page 198

Importing .NET Metadata

Overview

A basic prerequisite for accessing CORBA servers from a .NET application is that all of the OMG IDL data types be converted into .NET metadata. The .NET metadata enables .NET applications to access CORBA objects and data using C# syntax.

Orbix remoting .NET metadata

To integrate a .NET application with Orbix, you must import the Orbix remoting .NET metadata from the following file:

OrbixInstallDir`\bin\Orbix.Remoting.dll`

Generating .NET metadata

For each OMG IDL module that you want to access, you need to generate a .NET metadata assembly.

For example, to produce a .NET metadata assembly for the `fnbba` and `bankobjects` OMG IDL modules:

```
itts2il fnbba bankobjects
```

This command produces the following DLL file:

`fnbba.dll`

Importing .NET metadata

To import the .NET metadata assemblies into your .NET project, use the Visual Studio .NET **Project|Add References** dialog.

Initializing the Online Purchasing Manager Client

Overview

This section describes the C# subroutine, `Form1_Load()`, that runs during start-up to initialize the online purchasing manager client.

This example shows you how to use Orbix .NET connector to look up a CORBA object reference in the CORBA naming service and invoke operations on the object reference.

Form1_Load function

The `Form1_Load()` subroutine from the `Form1.cs` file is defined in Example 33.

Example 33: *The Online Purchasing Form1_Load Function*

```
// C#
...
1 using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using IONA.Remoting;
using fnbba;
using bankobjects;
...
namespace onlinepurchasingmanager
{
        public class Form1 : System.Windows.Forms.Form
        {
                private Random r = new Random();
                private OnlinePurchasing op;
                private BusinessSession sess;
                ...

                private void Form1_Load(
                    object sender,
                    System.EventArgs e
                )
                {
2                       ChannelServices.RegisterChannel(
                            new OrbixClientChannel()
                        );
3                     op=(OnlinePurchasing) Activator.GetObject(
                            typeof(OnlinePurchasing),
                            "NS:FNBBA_OnlinePurchasing"
                        );
```

Example 33: *The Online Purchasing Form1_Load Function*

```
4               BusinessSessionManager
                 bsm = (BusinessSessionManager)
                  Activator.GetObject(
                    typeof(BusinessSessionManager),
                   "NS:FNBBA_BusinessSessionManager"
                  );
5               SessionInfo_s sis=new SessionInfo_s();
                sis.session_type="Business";
                sis.client_id="onlinepurch" +
                                            r.Next(10000);
                sis.username="aidan";
                sis.password="foo";

6               sess = bsm.openSession(ref sis);
            }
            ...
```

The preceding C# code can be explained as follows:

1. The using statements indicate that the client is using the .NET remoting interfaces, `System.Runtime.Remoting`, the Orbix .NET connector interfaces, `IONA.Remoting`, and the `fnbba` .NET metadata, `fnbba`.
2. The call to `RegisterChannel()` initializes the Orbix .NET connector, making it available through the .NET remoting API.
3. This line, invoking `GetObject()`, shows you how to get a reference to an `fnbba::OnlinePurchasing` CORBA object by looking up the CORBA naming service. The first parameter is the C# type of the object. The second parameter consists of `NS:` followed by the name of the object as registered in the CORBA naming service.
4. This line shows you how to get a reference to an `fnbba::BusinessSessionManager` CORBA object by looking up the CORBA naming service.

5. The `SessionInfo_s` C# type is based on the following OMG IDL type:

    ```
    // IDL
    module fnbba {
      ...
      struct SessionInfo_s {
          string username;
          string password;
          string session_type;
          string client_id;
      };
      ...
    };
    ```

    The `fnbba::SessionInfo_s` OMG IDL struct type maps to the `SessionInfo_s` C# struct type.

6. This line invokes the `openSession()` operation on the remote `fnbba::BusinessSessionManager` object to initiate a client session on the mid-tier server.

Index

Symbols

#include preprocessor directive 54
<assembly-descriptor> element 103, 144
<cmp-field> element 144, 146
<container-transaction> element 132, 144
<context-root> element 152
<ejb-class> element 132
<ejb-jar> element 103
<ejb-name> element 132
<enterprise-beans> element 103
<entity> element 103, 143
<home> element 132
<jsp:forward> tag 166
<jsp:setProperty> tag 162, 166, 172
<jsp:useBean> tag 162, 166, 172
<persistence-type> element 144
<primkey-field> element 144
<remote> element 132
<session> element 103, 132
<welcome-file-list> element 153

A

accessor methods 127, 138
AccountDelegate class 28
Account interface
  implementation 21
AccountManager POA 16
AccountMgr interface
  implementation 39
AccountMgr object
  lifecycle responsibilities 33
AccountMgr type 5
accountNumList 10
Account objects
  creating 33, 41
  lifecycle 32
  opening 34
Account type
  description of design 5
  inheritance hierarchy 5
activation
  and servants 19
  description 15
  of a session bean 128
  of CORBA objects 14
addUser() method 159
AllDayBanking application
  overview 90
application.xml file
  <context-root> element 152
  and EAR file 100
  and EARSCO 95
architecture
  of CORBA bank application 13
ATMSessionDelegate class 60
ATMSession type 49
attributes
  in IDL 11
  readonly 11

B

back-end server
  purpose 4
BankObjects POA 16
bean attributes 127, 138
bean class
  for entity bean 136
  for session bean 125
bean-managed persistence
  and ejbFind() methods 140
BusinessSessionDelegate class 60
BusinessSessionManager.idl file 51
BusinessSessionManager POA 56
BusinessSessionManager type
  design 48
BusinessSession POA 56
BusinessSession type
  design 49
  inheritance hierarchy 48

C

CCGT 13
class files
  in Web module WAR file 106
Class-Path: entry 121

code generation
  C++ or Java 13
container configuration file 133
container-managed persistence
  and deployment descriptor 144
  overview 145
Container persistence type 144
CORBA back-end
  in a J2EE application 90
CORBA client
  GUI organization 64
  implementation 81
  purpose 64
CORBA Code Generation Toolkit 13, 20, 38
CORBA middle-tier server
  IDL for 51
  purpose 48
CORBA naming service 43
  and name resolution 57
  rebind() operation 46
  resolve() operation 59
CORBA objects
  activation 14
  and object ID 15
  lifecycle of Account 32
CORBA Persistent State Service 13
CORBA Session Management Plug-In 56
CORBA user exceptions 10
CosNaming package 45
CreateException type 138
create_reference_with_id() operation 43
CreditCardAccount type 5, 6
CurrentAccount type 5, 6
CustomerSession bean 166
  methods 155
  purpose 155

D
delegate class, serializing 30
delegate object, lifecycle of Account 32
delegation approach 18
  and BusinessSession interfaces 60
  base classes 61
  classes and interfaces for 19
  constructor 62
deployment descriptor
  for entity beans 142
  for session beans 131
  Web 106

E
EAR file
  and J2EE development cycle 92
  application.xml file 100
  definition 94
  directory layout 100
  MANIFEST.MF file 100
  META-INF directory 100
EARSCO
  and J2EE development cycle 92
  directory layout 95
  ejb-jar.xml file 96
  EJB module layout 96
  jboss.xml file 96
  MANIFEST.MF file 96
  purpose 94
  tmp directory 97
  web.xml file 96
  Web module layout 96
ejbActivate() method
  in entity bean 140
  in session bean 128
ejbCreate() method
  implementation for entity bean 141
  implementation for session bean 129
  in a session bean 128
ejbCreate() methods
  in an entity bean 139
EJB deployment descriptor
  for entity beans 142
ejbFind() methods 138
  in an entity bean 140
ejb-jar.xml file
  <assembly-descriptor> element 103, 144
  <cmp-field> element 144, 146
  <container-transaction> element 132, 144
  <ejb-name> element 132
  <enterprise-beans> element 103
  <entity> element 143
  <home> element 132
  <persistence-type> element 144
  <primkey-field> element 144
  <remote> element 132
  <session> element 132
  and container-managed persistence 146
  and EARSCO 96
  and entity bean configuration 142
  in EJB module JAR 102
  session bean configuration 131

ejb-jar file
   <ejb-class> element 132
ejbLoad() method
   in entity bean 140
EJB module
   and EARSCO 96
   class files 104
   directory layout 102
   META-INF directory 102
   referencing extra JAR files 97
ejbPassivate() method
   in entity bean 140
   in session bean 128
ejbPostCreate() methods
   in entity bean 139
ejbRemove() method
   in entity bean 139
   in session bean 128
EJB session bean 124
ejbStore() method
   in entity bean 140
EJB to CORBA link 110
enterprise application archive 94
enterprise application archive source code organization 94
EntityBean base class 137
entity beans
   base class 137
   bean class 136
   configuration 142
   definition 136
   ejbActivate() method 140
   ejbCreate() methods 139
   ejbFind() methods 140
   ejbLoad() method 140
   ejbPassivate() method 140
   ejbPostCreate() methods 139
   ejbRemove() method 139
   home interface 136
   lifecycle methods 139
   remote interface 136
enum type 54
exceptions, raises() clause 11

F
file suffix, for EAR file 98
finder methods 138
Forte for Java 72
   IDE screen layout 73
   Source Editor window 75
forward declaration, of IDL interface 11

G
getInetAccount() method
   calling from a JSP 174
   implementation 160
GUI client
   lodge funds dialog 69
   main screen 65
   new account dialog 67
   open account dialog 66
   transfer funds dialog 71
   withdraw funds dialog 70
GUI organization 64

H
home interface
   for entity beans 136
   for session beans 125
HttpServletResponse class 166

I
IdAssignmentPolicy type 16
IDL
   compiling 7
   definition of 7
   inout parameters 54
   interface 11
   language mappings 13
   language neutrality of 7
   modules 10
   operations 11
   preprocessor macros 10
   readonly attribute 11
   scoping of identifiers 54
   sequence type 10, 11
   struct type 10
   typedef 10
   user exceptions 10
IDL attributes 11
IDL compiler 7
   -jbase option 119
   -jpoa option 119
IDL inheritance 12
idlstubs.jar file 119
index.html file 164
inetAccountBean class

structure 126
inetAccount session bean
as CORBA client 110
purpose 124
InetSession type 49
inheritance
hierarchy for Account 5
in IDL 12
inheritance approach
base classes 40
classes and interfaces 38
constructor 40
overview 37
inheritance hierarchy
of Account types 21
of BusinessSession types 48
initialization service 45
inout parameters 54
integrated development environments
Forte for Java 72
NetBeans 72
interface
in IDL 11
interface, forward declaration 11
interfaces
implementing using delegation 18

J

J2EE development cycle 92
J2EE presentation layer
overview 150
JAR files
creating idlstubs.jar 120
referencing from EJB module 97, 121
referencing from Web module 106
Java compiler 119
Java extension mechanism 97, 121
java_poa_genie.tcl genie 38
-jbase option, of IDL compiler 119
jboss.xml file
and EARSCO 96
in session bean 133
-jpoa option, of IDL compiler 119
JSP
<jsp:forward> tag 166
<jsp:setProperty> tag 162, 166, 172
<jsp:useBean> tag 162, 166, 172
and processing Web forms 161
index.html file 164
in the J2EE presentation layer 150
main.jsp file 165
NewUser.jsp file 169
PayBill.jsp file 173
register.jsp file 171
response object 166

L

lifecycle, of Account objects 32
lifecycle methods
for a session bean 128
for entity beans 139
LifespanPolicy type 16
and BusinessSessionManager POA 56
lodgeFunds.form file 69
lodgeFunds.java file 69
lodge funds dialog
description 69
login form 163

M

main.jsp file 165
main screen 65
and dialog windows 81
MANIFEST.MF
in EJB module JAR file 102
in Web module WAR file 105
MANIFEST.MF file
and EAR file 100
and EARSCO 96
and Java extension mechanism 121
META-INF directory
and EAR files 100
in EJB module JAR 102
in Web module WAR file 105
modifier methods 127, 138
multi-threaded POA 14
multi-threading, and CORBA 14

N

NameComponent type 45, 58
NameService object identifier 45
NamingContextExt interface 46, 59
NamingContextPackage package 45
NetBeans 72
newAccount() method 41
newAccount.form file 68
newAccount.java file 68

new account dialog 67
NewRegSession bean
  attributes and methods 158
  purpose 158
NewUser.jsp file 169
New User Registration Web form 167

O

object ID
  and activation 15
  and object references 43
  and servants 19
object-oriented design 4
object references
  and the naming service 57
  creating 43
  definition 43
  publishing 44
  resolving 77
openAccount() method 42
openAccount.form file 66
openAccount.java file 66
openAccountActionPerformed() method
  implementation 83
open account dialog
  description 66
  implementation 83
operations, in IDL 11
Operations, Java interface suffix 20
ORB_CTRL_MODEL policy value 16

P

passivation
  of an entity bean 140
  of a session bean 128
PayBill.jsp file 173
persistence
  and CORBA 13
  serializing AccountDelegate 30
PERSISTENT policy value 16
POA
  and object references 43
  hierarchy for back-end server 15
  hierarchy for CORBA middle-tier 55
  role of 14
  threading 14
POA policies 16
POATie class 19
policies
  and the POA 16
  IdAssignmentPolicy type 16
  LifespanPolicy type 16
  ORB_CTRL_MODEL value 16
  PERSISTENT value 16
  SINGLE_THREAD_MODEL value 16
  SYSTEM_ID value 16
  threading 14
  ThreadPolicy type 16
  TRANSIENT value 16
  USER_ID value 16
portable object adapter, role of 14
postinvoke() method 35
preinvoke() method 35
preprocessor
  #include directive 54
  and IDL 10
presentation layer
  overview 91, 150
primary keys
  and ejbCreate() 138
  and EJB deployment descriptor 144
programming languages supported by CORBA 13
PSS 13
public files
  in a Web archive 151
  in Web module WAR file 107

R

readonly attribute 11
rebind() operation
  and CORBA naming 46
register.jsp file 171
registration of a new user 159
RemoteException type 126, 137
remote interface
  for entity bean 136
  for session bean 125
repository ID, and object references 43
Required transaction attribute 132, 144
resolution
  of BusinessSessionManager object reference 77
resolution of names 57
resolve() operation 59
resolveAccount() method 124
resolve_initial_references() operation 45
response object, in a JSP 166
root POA 55

S
SavingsAccount type 5, 6
scoping
  in IDL 10
  of IDL identifiers 54
sequence type
  AccountTransactions type 11
  and the naming service 45
  IDL example 10
Serializable Java interface 28
serializing, writeObject() method 30
Servant Java interface 18
servant locator
  and activation state 32
  and updating Account objects 35
  lifecycle responsibilities 33
servant objects
  and the tie approach 18
  definition 19
services
  CORBA naming 43
  full OTS 13
  initialization 45, 58
  OTS-Lite 13
  session management 13
  SSL/TLS 13
SessionBean base class 126
session beans
  base class 126
  bean class 125
  callback methods 127
  definition 125
  ejbActivate() method 128
  ejbCreate() method 128, 129
  EJB deployment descriptor 131
  ejbPassivate() method 128
  ejbRemove() method 128
  home interface 125
  jboss.xml file 133
  lifecycle methods 128
  overview 124
  remote interface 125
single-threaded POA 14
SINGLE_THREAD_MODEL policy value 16
skeleton code 119
skeleton files, and the IDL compiler 7
struct type in IDL 10
stub code
  creating for application 119
  for CORBA back-end 111
stub files
  and IDL compiler 7
  as template for implementation 38
SYSTEM_ID policy value 16

T
TellerSessionDelegate class 60
TellerSession type 49
ThreadPolicy type
  and BusinessSessionManager POA 56
  overview 16
TIE approach
  generated TIE class 19
  overview 18
tmp directory
  and EARSCO 97
tools
  idl 7
transactions
  Required attribute 132, 144
transferFunds.form file 71
transferFunds.java file 71
transfer funds dialog
  description 71
TRANSIENT policy value 16
typedef, and IDL 10

U
UML 4
URL
  for J2EE AllDayBanking application 91
  to access AllDayBanking application 152
UserBean class, structure 137
User entity bean, purpose 135
user exceptions, in IDL 10
USER_ID policy value 16

V
validateUser() method, in CustomerSession 156
validation of user identity 156

W
WAR file 105
web.xml file
  <welcome-file-list> element 153
  and EARSCO 96

example 106
in Web module WAR file 106
Web client
in a J2EE application 91
Web deployment descriptor 106
Web form, processing with a JSP 161
WebHelper class
overview 160
using in a JSP 174
WEB-INF directory, in Web module WAR file 105
Web module
and EARSCO 96
public files 97, 151
referencing JAR files 106
source code 97
Web module WAR file
directory layout 105
WEB-INF directory 105
withdrawFunds.form file 70
withdrawFunds.java file 70
withdraw funds dialog, description 70
worker beans
description 151
purpose 154
writeObject() method 30