CHAPTER 2

GETTING STARTED

This chapter helps you become acquainted with some of the development tools, and describes how to develop a distributed, object-based application with VisiBroker for Java. A sample application is used to illustrate each step of the process. This sample application is used repeatedly throughout this manual as new features are introduced. This chapter includes the following major sections:

The Bank Application 2-2
Developing Applications 2-2
Setting up Your Environment 2-4
The Bank Module 2-4
Running the idl2java Compiler 2-4
The Client 2-6
The Server 2-9
Comparing Java and C++ Code 2-12
Building the Example Programs 2-13

THE BANK APPLICATION

In this chapter, you will build a sample client application that can query on the balance in a bank account. From this example, you will learn how to:

- Implement simple interfaces in IDL for `Account` and `AccountManager`.
- Generate client stubs and server skeletons using the `idl2java` compiler.
- Implement the client program: initialize the ORB, bind to the `AccountManager` object, obtain the balance, and handle exceptions.
- Implement the server object: initialize the ORB and BOA, instantiate an implementation, and prepare to receive requests.
- Build the example.
- Run the example: start the Smart Agent, start the `Account Manager` server, and run the client program.

The code for this example is provided in the **bank** directory within the **java_examples** directory where the VisiBroker for Java package was installed. If you do not know the location of the VisiBroker for Java package, see your system administrator.

DEVELOPING APPLICATIONS

When you develop distributed applications with VisiBroker, you must first identify the objects required by the application. You will then usually follow these steps:

1. Write a specification for each object using the Interface Definition Language (IDL).
2. Use the IDL compiler to generate the client stub code and server skeleton code.
3. Write the client application code.
4. Write the server object code.
5. Compile the client and server code.
6. Start the server.
7. Run the client application.

Figure 2-1 on the following page illustrates these steps.

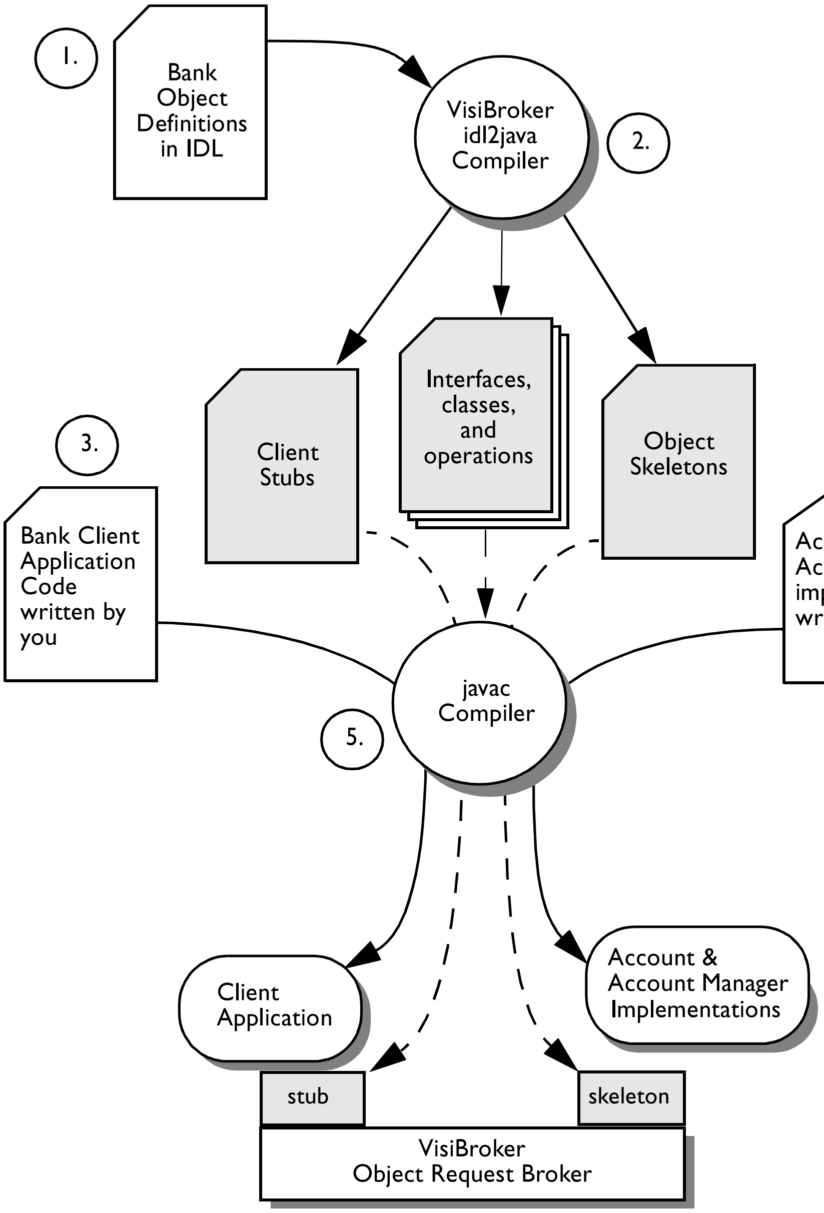
1.
Bank Object Definitions in IDL
VisiBroker idl2java Compiler
2.
Client Stubs
Interfaces, classes, and operations
Object Skeletons
3.
Bank Client Application Code written by you
4.
Account & Account Manager implementations written by you
javac Compiler
5.
Client Application
Account & Account Manager Implementations
stub
skeleton
VisiBroker Object Request Broker

Figure 2-1 Developing the sample Bank application

SETTING UP YOUR ENVIRONMENT

Before you can compile and run your client and server applications, make sure that the following environment variables are set up correctly:

- `PATH`
- `VBROKER_ADM`
- `CLASSPATH`

For more information about setting up these environment variables, see the *VisiBroker for Java Installation and Administration Guide.*

THE BANK MODULE

The **Bank.idl** file defines the `Bank` module that contains two interfaces. The `Account` interface provides a method for obtaining the current balance. The `AccountManager` interface provides a method that opens an account with a specified name and returns an `Account` object reference. The `Account` object reference can then be used to obtain the balance in the account.

IDL

```
module Bank {
    interface Account {
      float balance();
    };
    interface AccountManager {
      Account open(in string name);
    };
};
```

IDL Sample 2-1 ***Bank.idl*** *provides the* `Account` *and* `AccountManager` *interface definitions.*

RUNNING THE idl2java COMPILER

The VisiBroker IDL compiler, named `idl2java,` generates Java code from a file containing IDL specifications. Since the **Bank.idl** file requires no special handling, you can compile it by typing the following command.

```
prompt> idl2java Bank.idl
```

For more information on the command line options for the `idl2java` compiler, see the *VisiBroker for Java Reference Manual.*

Code Generation

Since Java allows only one public interface or class per file, compiling the **Bank.idl** file will generate several java files. These files are stored in a generated sub-directory named **Bank**, which is the `module` name specified in the IDL file. IDL modules are mapped to Java packages, so all of the files listed below will be part of the `Bank` package.

- **Account.java** - The `Account` interface declaration.
- **AccountManager.java** - The `AccountManager` interface declaration.
- **AccountHelper.java** - Declares the `AccountHelper` class, which contains an `AccountHelper`. This class defines helpful utility functions.
- **AccountManagerHelper.java** - Declares the `AccountManagerHelper` class. This class defines helpful utility functions.

NOTE *The* `<interface name>_var` *class has been removed because of changes in VisiBroker to comply with the new OMG IDL to Java language mapping specification. This class has been split into two. The new names are* `<interface name>Helper` and `<interface name>Holder`.

- **AccountHolder.java** - Declares the `AccountHolder` class, which provides a holder for passing parameters.
- **AccountManagerHolder.java** - Declares the `AccountManagerHolder` class, which provides a holder for passing parameters.
- **_st_Account.java** - Stub code for the `Account` object on the client side.
- **_st_AccountManager.java** - Stub code for the `AccountManager` object on the client side.
- **_AccountImplBase.java** - Skeleton code (implementation base code) for the `Account` object implementation on the server side.
- **_AccountManagerImplBase.java** - Skeleton code (implementation base code) for the `AccountManager` object implementation on the server side.

NOTE *The* `_sk_<interface name>` *class has been deprecated because of changes in VisiBroker to comply with the new OMG IDL to Java language mapping specification. The new name is* `_<interface name>ImplBase`. *While the old format,* `_sk_<interface name>` *will still be generated for backward compatibility, the new format* `_<interface name>ImplBase` *should be used when implementing new objects.*

- **_tie_Account.java** - Class used to implement the `Account` object on the server side using the tie mechanism, described in Chapter 9.

- **_tie_AccountManager.java** - Class used to implement the `AccountManager` object on the server side using the tie mechanism, described in Chapter 9.
- **AccountOperations.java** - The Operation classes work together with the tie classes to provide the tie mechanism.
- **AccountManagerOperation.java** - The Operation classes work together with the tie classes to provide the tie mechanism.
- **_example_Account.java** - Code you can fill in to implement the `Account` object on the server side.
- **_example_AccountManager.java** - Code you can fill in to implement the `AccountManager` object on the server side.

For more information about the Helper, Holder, and Operations classes, see the “Generated Classes” chapter in the *VisiBroker for Java Reference Manual.*

THE CLIENT

Many of the files used in implementing the bank client are contained in the `Bank` package generated by the `idl2java` compiler as shown in the previous section. The **Client.java** file is provided to illustrate this example. Normally you would create this file.

Client.java

The `Client` class implements the client application that obtains the current balance of a bank account. See Code Sample 2-1 for a code sample of **Client.java**. The program performs the following steps:

- Initializes the Object Request Broker.
- Binds to an `AccountManager`.
- Requests the `AccountManager` to open the `Account` with the specified name. If no account name is supplied, a default name is used.
- Gets the balance of the `Account` using the object reference returned by the `open` method.
- Prints out the balance.

CODE

```
// Client.java

public class Client {

public static void main(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Locate an account manager.
    Bank.AccountManager manager =
      Bank.AccountManagerHelper.bind(orb,"BankManager");
    // use args[0] as the account name, or a default.
    String name = args.length > 0 ? args[0] : "Jack B. Quick";
    // Request the account manager to open a named account.
    Bank.Account account = manager.open(name);
    // Get the balance of the account.
    float balance = account.balance();
    // Print out the balance.
    System.out.println
        ("The balance in " + name + "'s account is $" + balance);
     }
}
```

Code Sample 2-1 ***Client.java*** *provides the client side program, written in Java.*

AccountManagerHelper.java

This file is located in the `Bank` package. It contains an `AccountManagerHelper` object and defines several methods for binding to the server that implements this object. The bind method transparently contacts a Smart Agent, described in Chapter 10, to locate the object. Our example application uses the version of the `bind` method that accepts an object name, but the client may optionally specify a particular host and special bind options. For information on these additional methods, see the “Specifying Bind Options” on page 4-7. For more information about Helper classes, see the “Generated Classes” chapter in the *VisiBroker for Java Reference Manual.*

NOTE *The bind method has changed from previous versions of VisiBroker. The static bind operations on the Helper class require that an instance of org.omg.CORBA.ORB be supplied.*

CODE

```
package Bank;
abstract public class AccountManagerHelper {
    ...

  public static Bank.AccountManager bind(org.omg.CORBA.ORB orb,
                                         java.lang.String name) {
    ...
  }
}
```

*Code Sample 2-2 A portion of the **AccountManagerHelper.java** file.*

THE bind METHOD

When your application invokes the `bind` method, the ORB locates and establishes a connection with an appropriate server and returns a handle to the `AccountManager` object. If the ORB cannot locate or connect to the Bank.AccountManager server object, the `bind` method will raise a CORBA system exception.

OTHER METHODS

Several other methods are provided that allow your client application to manipulate an `AccountManager` object reference. Most of these methods are not used in the example client application, but they are described in detail in the *VisiBroker for Java Reference Manual*.

_st_AccountManager.java

This file is located in the `Bank` package. It contains the stub code for the AccountManager object on the client side. In the example, the client uses the `AccountManager` object `open` method to open an account.

The `open` method generated by the `idl2java` compiler in this file is really a stub method. When your client application calls the `open` method, a request is sent to the ORB with all the necessary parameters. The ORB ensures that the request is sent to the server that implements the `AccountManager` object. Once the method is executed on the server, the ORB returns the results to your client application.

CAUTION *Do not modify this generated file.*

_st_Account.java

This file is located in the `Bank` package. It contains the stub code for the Account object on the client side. In the example, the client uses the `Account` object `balance` method to obtain an account balance.

The `balance` method generated by the idl2java compiler in this file is also a stub method. When your client application calls the `balance` method, a request is sent to the ORB with all the necessary parameters. The ORB ensures that the request is sent to the server that implements the `Account` object. Once the method is executed on the server, the ORB returns the results to your client application.

CAUTION *Do not modify this generated file.*

THE SERVER

Just as with the client, many of the files used in implementing the bank server are contained in the `Bank` package generated by the `idl2java` compiler. The **Server.java** file is a server implementation included for the purposes of illustrating this example. Normally you, the programmer, would create this file.

Server.java

This file implements the `Server` class for the server-side of our banking example.

The `Server` class does the following:

- Initializes the Object Request Broker.
- Initializes the Basic Object Adaptor. For more information, see “What is the Basic Object Adapter?” on page 5-2.
- Creates an `AccountManager` object.
- Activates the newly created object.
- Prints out a status message.
- Waits for incoming client requests.

CODE

```
// Server.java

public class Server {

  public static void main(String[] args) {
     // Initialize the ORB.
     org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
     // Initialize the BOA.
     org.omg.CORBA.BOA boa = orb.BOA_init();
     // Create the account manager object.
     Bank.AccountManager manager =
       new AccountManagerImpl("BankManager");
     // Export the newly created object.
     boa.obj_is_ready(manager);
     System.out.println(manager + " is ready.");
     // Wait for incoming requests
     boa.impl_is_ready();
  }
}
```

*Code Sample 2-3 A portion of the **Server.java** file, showing the server implementation.*

CODE

```
// AccountManagerImpl.java
import java.util.*;

public class AccountManagerImpl extends Bank._AccountManagerImplBase {
  public AccountManagerImpl(String name) {
    super(name);
  }
  public synchronized Bank.Account open(String name) {
    // Lookup the account in the account dictionary.
    Bank.Account account = (Bank.Account) _accounts.get(name);
    // If the account is not in the dictionary, create an account.
    if(account == null) {
      // Create a new account with between 0 and 1000 dollars.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
      // Create the account implementation, given the balance.
      account = new AccountImpl(balance);
      // Make the object available to the ORB.
      _boa().obj_is_ready(account);
      System.out.println("Created " + name + "'s account: " + account);
      // Save the account in the account dictionary.
      _accounts.put(name, account);
    }
    // Return the account.
    return account;
  }
  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();
}
```

Code Sample 2-4 *A portion of the* ***AccountManagerImpl.java*** *file, showing the* `AccountManager` *implementation.*

The `AccountManagerImpl` class in this file extends the `Bank._AccountManagerImplBase` class generated in the `Bank` package. This class provides the actual implementation of the `open` method used to obtain or create an `Account`. The manager uses a Dictionary to contain all the accounts. When a request to open an account is received, the `open` method does the following:

- Looks for an `Account` with a matching name.
- If a match is not found, a new `Account` is created with that name and a randomly generated balance. A new object is activated.
- An object reference to the new or existing `Account` is returned.

The AccountManager Class Hierarchy

The `AccountManagerImpl` class that you implement extends the `Bank` package's `_AccountManagerImplBase` class that was generated by the IDL compiler. The skeleton class, `AccountManagerImpl`, implements the `AccountManager` interface. Figure 2-2 shows the relationship between these classes.

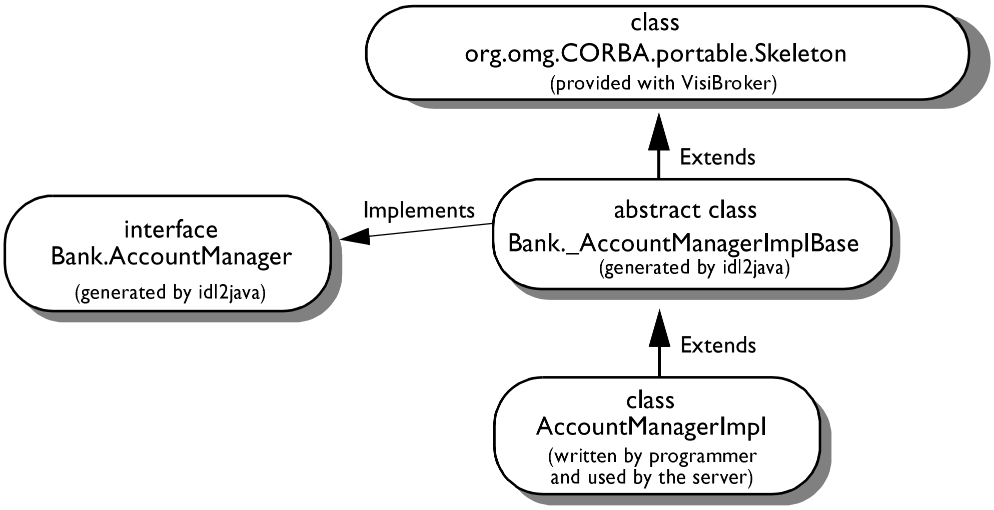
class
org.omg.CORBA.portable.Skeleton
(provided with VisiBroker)
Extends
Implements
interface
Bank.AccountManager
(generated by idl2java)
abstract class
Bank._AccountManagerImplBase
(generated by idl2java)
Extends
class
AccountManagerImpl
(written by programmer
and used by the server)

Figure 2-2 Class hierarchy for the `AccountManager` *interface.*

COMPARING JAVA AND C++ CODE

If you are more familiar with C++, it may be useful to examine the C++ files that are provided with the `Bank` example. Compare the Java and C++ code to see how the IDL/Java language mapping relates to the IDL/C++ language mapping. The same IDL file was used to generate both the Java and the C++ examples. VisiBroker for C++, sold separately, was used to generate the C++ files. The two C++ source files are:

- **Client.C** - provides the client side program, written in C++. This file is called **Client.cpp** on Windows systems.
- **Server.C** - provides the server side program, written in C++. This file is called **Server.cpp** on Windows systems.

BUILDING THE EXAMPLE PROGRAMS

W Assuming the VisiBroker distribution was installed in `C:\vbroker`, use the following commands:

```
prompt> cd c:\vbroker\java_examples\bank
prompt> vbmake
```

The command `vbmake` is a batch file which runs the `idl2java` compiler and then compiles each file.

U Assuming the VisiBroker distribution was installed in **/usr/local**, issue the following commands:

```
prompt> cd /usr/local/vbroker/java_examples/bank
prompt> make
```

In this example, `make` is the standard UNIX utility. If you do not have it in your path, see your system administrator.

If you encounter problems during the build process, see our FAQ at ORB Central on the Visigenic web site at <**http://www.visigenic.com**>.

Starting a Smart Agent (osagent)

The Smart Agent, or *osagent*, provides a fault-tolerant object location service. Ordinarily, you will want the osagent to be running on at least one machine in the LAN environment for you to run your applications. The Smart Agent is described in detail in Chapter 10. See the *VisiBroker for C++ Administration and Installation Guide* for details on starting the Smart Agent. Before starting the osagent, you must set the VBROKER_ADM environment variable properly. The basic command for starting the Smart Agent is as follows:

```
prompt> osagent
```

If you're running Windows NT, you can start the Smart Agent as an NT service through the Services Control Panel. Alternatively, you can use the following command:

NT

```
prompt> osagent -c (or osagent -C)
```

The `-c` (or `-C`) option allows the OSAgent to run in console mode. Otherwise, it is considered an NT service.

Running the Server

Now, you can start the server using VisiBroker's `vbj` command. The vbj command invokes the virtual machine and offers other special features. For more information, see "Commands" in the *VisiBroker for Java Reference Manual*.

```
prompt> vbj Server
```

If you have problems starting the Server, refer to *The VisiBroker for Java GateKeeper Guide*.

Running a Client

After starting the osagent and the server, a client can be started in a different window. If a command line argument is provided to the client, that argument will be used as the name of the bank account to be opened:

```
prompt> vbj Client
The balance in Jack B. Quick's account is $71.62

prompt> vbj Client "Jack Sprat"
The balance in Jack Sprat's account is $322.38
```

Figure 2-3 Running the bank client application with the `vbj` *command.*

Since the account balance is computed randomly, the actual output you see will differ from that shown in Figure 2-3. If you encounter problems running the client application, refer to *The VisiBroker for Java GateKeeper Guide*.

Since the Server prints out a message every time it creates a new Account, you will see the output following on the server side:

```
prompt> vbj Server
AccountManager[Server,oid=...] is ready.

Created Jack B. Quick's account: Account[Server,oid=...]

Created Jack Sprat's account: Account[Server,oid=...]
```

Figure 2-4 Output from the bank server.