**4. Overview of JavaIDL**

Java TM IDL is a technology for distributed objects, that is, objects interacting on different platforms across a network. Java IDL is similar to RMI (Remote Method Invocation), which supports distributed objects written entirely in the Java programming language. However, Java IDL enables objects to interact regardless of whether they're written in the Java programming language or another language such as C, C++, COBOL, or others.

This is possible because Java IDL is based on the Common Object Request Brokerage Architecture (CORBA), an industry-standard distributed object model. A key feature of CORBA is IDL, a language-neutral Interface Definition Language. Each language that supports CORBA has its own IDL mapping--and as its name implies, Java IDL supports the mapping for Java.

To support interaction between objects in separate programs, Java IDL provides an Object Request Broker, or ORB. The ORB is a class library that enables low-level communication between Java IDL applications and other CORBA-compliant applications.

Any relationship between distributed objects has two sides: the client and the server. The server provides a remote interface, and the client calls a remote interface. These relationships are common to most distributed object standards, including RMI and CORBA. Note that in this context, the terms client and server define object-level rather than application-level interaction--any application could be a server for some objects and a client of others. In fact, a single object could be the client of an interface provided by a remote object and at the same time implement an interface to be called remotely by other objects.

On the client side, the application includes a reference for the remote object. The object reference has a stub method, which is a stand-in for the method being called remotely. The stub is actually wired into the ORB, so that calling it invokes the ORB's connection capabilities, which forwards the invocation to the server.

On the server side, the ORB uses skeleton code to translate the remote invocation into a method call on the local object. The skeleton translates the call and any parameters to their implementation-specific format and calls the method being invoked. When the method returns, the skeleton code transforms results or errors, and sends them back to the client via the ORBs.

Between the ORBs, communication proceeds by means of a shared protocol, IIOP--the Internet Inter-ORB Protocol. IIOP, which is based on the standard TCP/IP internet protocol, defines how CORBA-compliant ORBs pass information back and forth. Like CORBA and IDL, the IIOP standard is defined by OMG, the Object Management Group.

**The Java IDL Development Process**

The following steps provide a general guide to designing and developing a distributed object application with Java IDL. Links to the relevant steps of the tutorial will guide you through creating this sample application.

***1. Define the remote interface***

Note that if you're implementing a client for an existing CORBA service, or a server for an existing client, you would get the IDL interfaces from the implementer--such as a service provider or vendor. You would then run the **idlj** compiler over those interfaces and follow these steps.

*1.1 Writing the IDL file bank.idl*

This section teaches you how to write a simple IDL interface definition and how to translate the IDL interface to Java. It also describes the purpose of each file generated by the **idlj** compiler.

To create the bank.idl file,

* + 1. Start your favorite text editor and create a file named **bank.idl**.
    2. In your file, enter the code for the interface definition:

module bankidl {  
 interface Account {  
 readonly attribute float balance;  
 exception rejected { string reason; };  
 void deposit(in float value) raises( rejected);  
 void withdraw(in float value) raises( rejected);  
 };  
  
 interface Bank {  
        exception rejected { string reason; };  
        Account newAccount( in string name) raises( rejected);  
        Account getAccount ( in string name);  
        boolean deleteAccount( in string acc );  
    };  
};

* + 1. Save the file.

*1.2 Understanding the IDL file*

OMG IDL is a purely declarative language designed for specifying programming-language-independent operational interfaces for distributed applications. OMG specifies a mapping from IDL to several different programming languages, including C, C++, Smalltalk, COBOL, Ada, and Java. When mapped, each statement in OMG IDL is translated to a corresponding statement in the programming language of choice. You can use the tool **idlj** to map an IDL interface to Java and implement the client class. When you map the same IDL to C++ and implement the server in that language, the Java client and C++ server interoperate through the ORB as though they were written in the same language.

You need perform these steps:

* + 1. Declare the CORBA IDL module
    2. Declare the interface
    3. Declare the operations
    4. Declare the exceptions (if needed, as you do for our **Bank** program)
    5. Declare the attributes (if needed, as you do for our **Bank** program)

*1.3 Declaring the CORBA IDL Module*

A CORBA module is a namespace that acts as a container for related interfaces and declarations. It corresponds closely to a Java package. Each module statement in an IDL file is mapped to a Java package statement.

The module statement looks like this:

module bankidl {  
    // Subsequent lines of code here.  
};  
  
When you compile the IDL, the module statement will generate a package statement in the Java code.

*1.4 Declaring the Interfaces*

Like Java interfaces, CORBA interfaces declare the API contract an object has with other objects. Each interface statement in the IDL maps to a Java interface statement when mapped.

In your **bank.idl** file, the interface statement looks like this:

module bankidl {  
 interface Account {  
 ...  
 };  
  
    interface Bank {  
 ...  
    };  
};

When you compile the IDL, this statement will generate an interface statement in the Java code.

*1.5 Declaring the Operations*

CORBA operations are the behaviour that servers promise to perform on behalf of clients that invoke them. Each operation statement in the IDL generates a corresponding method statement in the generated Java interface.

In your **bank.idl** file, the operation statement looks like this:

module bankidl  
{  
    interface Account {  
        void deposit(in float value) raises( rejected);  
        void withdraw(in float value) raises( rejected);  
    };  
    interface Bank {  
        Account newAccount( in string name) raises( rejected);  
        Account getAccount ( in string name);  
        boolean deleteAccount( in string acc );  
    };  
};

*1.6 Declaring the Exceptions*

In your **bank.idl** file, the exception statement looks like this:

module bankidl {  
 interface Account {  
 exception rejected { string reason; };  
    };  
  
    interface Bank {  
 exception rejected { string reason; };  
    };  
};

*1.7 Declaring Attributes*

In your **bank.idl** file, the exception statement looks like this:

module bankidl {  
    interface Account {  
 readonly attribute float balance;  
    };  
  
    interface Bank {  
 // no attributes here  
    };  
};

***2. Compile the remote interface***

When you run the idlj compiler over your interface definition file, it generates the Java version of the interface, as well as the class code files for the stubs and skeletons that enable your applications to hook into the ORB.

*2.1 Mapping bank.idl to Java*

The tool **idlj** reads OMG IDL files and creates the required Java files. The **idlj** compiler defaults to generating only the client-side bindings. If you need both client-side bindings and server-side skeletons (as you do for our Bank program), you must use the **-fall**option when running the **idlj** compiler.

1. Go to a command line prompt.
2. Change to the directory containing your bank.idl file.
3. Enter the compiler command: **idlj -fall -oldImplBase bank.idl**Note that the option **-oldImplBase** is an optional and it dictates to generate bindings that are backwards compatible to J2SE 1.4 (like in this example). Read more about [idlj options](http://download.oracle.com/javase/6/docs/technotes/tools/share/idlj.html) in the documentation.

If you list the contents of the directory, you will see that a directory called **bankidl** has been created and that it contains a number of files. Open **bank.java** in your text editor. **bank.java** is the *signature interface* and is used as the signature type in method declarations when interfaces of the specified type are used in other interfaces. It looks like this:

package bankidl;  
  
/\*\*  
\* bankidl/Bank.java .  
\* Generated by the IDL-to-Java compiler (portable), version "3.2"  
\* from bank.idl  
\* den 25 oktober 2007 kl 20:51 CEST  
\*/  
  
public interface Bank extends BankOperations, org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity   
{  
} // interface Bank

With an interface this simple, it is easy to see how the IDL statements map to the generated Java statements.

|  |  |
| --- | --- |
| **IDL Statement** | **Java Statement** |
| module bankidl | package bankidl; |
| interface Bank | public interface Bank |

The single surprising item is the extends statement. All CORBA objects are derived from **org.omg.CORBA.Object** to ensure required CORBA functionality. The required code is generated by **idlj**; you do not need to do any mapping yourself.

In previous versions of the **idlj** compiler (known as **idltojava**), the operations defined on the IDL interface would exist in this file as well. Starting with J2SDK v1.3.0, the IDL-to-Java mapping puts all of the operations defined on the IDL interface in the *operations interface*, **BankOperations.java**. The operations interface is used in the server-side mapping and as a mechanism for providing optimized calls for co-located clients and servers. For **bank.idl**, this file looks like this:

package bankidl;  
  
/\*\*  
\* bankidl/BankOperations.java .  
\* Generated by the IDL-to-Java compiler (portable), version "3.2"  
\* from bank.idl  
\* den 25 oktober 2007 kl 20:51 CEST  
\*/  
  
public interface BankOperations   
{  
 bankidl.Account newAccount (String name) throws bankidl.BankPackage.rejected;  
 bankidl.Account getAccount (String name);  
 boolean deleteAccount (String acc);  
} // interface BankOperations

It is easy to see how the IDL statements map to the generated Java statements.

|  |  |
| --- | --- |
| **IDL Statement** | **Java Statement** |
| Account newAccount( in string name) raises( rejected); | bankidl.Account newAccount (String name) throws bankidl.BankPackage.rejected; |

*2.2 Understanding the****idlj****Compiler Output*

The **idlj** compiler generates a number of files. The actual number of files generated depends on the options selected when the IDL file is compiled. The generated files provide standard functionality, so you can ignore them until it is time to deploy and run your program. The files generated by the **idlj** compiler for **bank.idl** of the**Bank** interface, with the **-fall**and **-oldImplBase**options, are:

**\_BankImplBase.java**

This abstract class is the server skeleton, providing basic CORBA functionality for the server. It implements the**Bank.java** interface. The servant class **BankImpl** extends **\_BankImplBase**. Note that in this example, the

**-oldImplBase**

option dictates to generate bindings that are backwards compatible to J2SE 1.4. If this option is omitted, the compiler generates the **BankPOA.java**file instead of **\_BankImplBase.java**, and the servant class **BankImpl** to inherit and implement the **Bank** interface must extend **BankPOA** (rather than **\_BankImplBase**).

**\_BankStub.java**

This class is the client stub, providing CORBA functionality for the client. It implements the **Bank.java** interface.

**Bank.java**

This signature interface contains the Java version of our IDL interface. The **Bank.java** interface extends**org.omg.CORBA.Object**, providing standard CORBA object functionality. It also extends **IDLEntity**, and is used as the signature type in method declarations when interfaces of the specified type are used in other interfaces.

**BankHelper.java**

This final class provides auxiliary functionality, notably the narrow() method required to cast CORBA object references to their proper types.

**BankHolder.java**

This final class holds a public instance member of type **Bank**. It provides operations for out and inout arguments, which CORBA allows, but which do not map easily to Java's semantics.

**BankOperations.java**

This operations interface contains the methods **Account newAccount( in string name) raises(rejected)**,**Account getAccount ( in string name)**, **boolean deleteAccount( instringacc )**. The IDL-to-Java mapping puts all of the operations defined on the IDL interface into this file. The operations interface is used in the server-side mapping and as a mechanism for providing optimized calls for co-located clients and servers. When you write the IDL interface, you do all the programming required to generate all these files for your distributed application. The next steps are to implement the client and server classes.

***3. Implement the server and servants***

Once you run the **idlj** compiler, you can use the skeletons it generates to put together your server application. In addition to implementing the methods of the remote interface, your server code includes a mechanism to start the ORB and wait for invocation from a remote client.

The example server consists of three classes, the servant for the bank and account and the server. The servants, **BankImpl** and **AccountImpl**, are the implementation of the **Bank** and **Account** IDL interface; each **Bank** and **Account** instance is implemented by a **BankImpl** or **AccountImpl** and instance. The servants are a subclasses of **\_BankImplBase** and **\_AccountImplBase**, which are generated by the **idlj** compiler from the example IDL. The servants contain methods for each IDL operation. Servant methods are just like ordinary Java methods; the extra code to deal with the ORB, with marshalling arguments and results, and so on, is provided by the server and the stubs.

The server class has the server's **main()** method, which:

* Creates an ORB instance
* Creates a servant instance (the implementation of one CORBA Bank object) and tells the ORB about it
* Gets a CORBA object reference for a naming context in which to register the new CORBA object
* Registers the new object in the naming context under the name of the bank
* Waits for invocations of the new object

*Understanding Server.java*

This section explains each line of **Server.java**, describing what the code does, as well as why it is needed for this application.

The structure of a CORBA server program is the same as most Java applications: You import required library packages, declare the server class, define a **main()** method, and handle exceptions.

First, we import the packages required for the server class:

// Server will use the naming service.  
import org.omg.CosNaming.\*;  
  
// The package containing special exceptions thrown by the name service.  
import org.omg.CosNaming.NamingContextPackage.\*;  
  
// All CORBA applications need these classes.  
import org.omg.CORBA.\*;

The next step is to declare the server class:

public class Server   
{  
  // The main() method goes here.  
}

Every Java application needs a **main** method. It is declared within the scope of the **Server** class:

  public static void main(String args[])  
  {  
    // The try-catch block goes here.  
  }

Because all CORBA programs can throw CORBA system exceptions at runtime, all of the **main()** functionality is placed within a **try**-**catch** block. CORBA programs throw runtime exceptions whenever trouble occurs during any of the processes (marshalling, unmarshalling, upcall) involved in invocation. The exception handler simply prints the exception and its stack trace to standard output so you can see what kind of thing has gone wrong.

The **try**-**catch** block is set up inside **main()**, as shown:

    try{  
 // The rest of the Server code goes here.  
 } catch(Exception e) {  
        System.err.println("ERROR: " + e);  
        e.printStackTrace(System.out);  
    }

*Creating an ORB Object*

Just like in the client application, a CORBA server also needs a local ORB object. Every server instantiates an ORB and registers its servant objects so that the ORB can find the server when it receives an invocation for it.

The ORB variable is declared and initialized inside the try-catch block.

      ORB orb = ORB.init(args, null);

The call to the ORB's **init()** method passes in the server's command line arguments, allowing you to set certain properties at runtime.

*Managing the Servant Object*

A server is a process that instantiates one or more servant objects. The servant implements the interface generated by **idlj** and actually performs the work of the operations on that interface. Our **Server** needs a**BankImpl**.

*Instantiating the Servant Object*

We instantiate the servant object inside the **try**-**catch** block, just below the call to **init()**, as shown:

      BankImpl bankRef = new BankImpl(args[2]);

The next step is to connect the servant to the ORB, so that the ORB can recognize invocations on it and pass them along to the correct servant:

      orb.connect(bankRef);

*Defining the Servant Class*

public class BankImpl extends bankidl.\_BankImplBase  
{  
  // The operation methods goes here.  
}

The servant is a subclass of **\_BankImplBase** so that it inherits the general CORBA functionality generated for it by the compiler.

 Next, we declare the required methods:

    public bankidl.Account newAccount(java.lang.String name) throws bankidl.BankPackage.rejected {  
 // The method implementation goes here.  
    }  
  
    public bankidl.Account getAccount(java.lang.String name) {  
 // The method implementation goes here.  
    }  
  
    public boolean deleteAccount(java.lang.String name) {  
 // The method implementation goes here.  
    }

*Working with COS Naming*

The Server works with the naming service to make the servant object's operations available to clients. The server needs an object reference to the name service, so that it can register itself and ensure that invocations on the Bank interface are routed to its servant object.

*Obtaining the Initial Naming Context*

In the **try**-**catch** block, below instantiation of the servant, we call **orb.resolve\_initial\_references()** to get an object reference to the name server:

      org.omg.CORBA.Object objRef = orb.resolve\_initial\_references("NameService");

The string **NameService** is defined for all CORBA ORBs. When you pass in that string, the ORB returns a naming context object that is an object reference for the name service.

*Narrowing the Object Reference*

As with all CORBA object references, **objRef** is a generic CORBA object. To use it as a **NamingContext** object, you must narrow it to its proper type. The call to **narrow()** is just below the previous statement:

      NamingContext ncRef = NamingContextHelper.narrow(objRef);

Here you see the use of an **idlj**-generated helper class, similar in function to **BankHelper**. The **ncRef** object is now an **org.omg.CosNaming.NamingContext** and you can use it to access the naming service and register the server, as shown in the next topic.

*Registering the Servant with the Name Server*

Just below the call to **narrow()**, we create a new **NameComponent** member:

      NameComponent nc = new NameComponent(args[2], "");

This statement sets the **id** field of **nc** to **arg[2]** and the **kind** component to the empty string. Make sure there are no spaces between the "".

Because the path to the **Bank** has a single element, we create the single-element array that**NamingContext.resolve** requires for its work:

      NameComponent path[] = {nc};

Finally, we passpath and the servant object to the naming service, binding the servant object to the **args[2]** id:

      ncRef.rebind(path, bankRef);

Now, when the client calls **resolve(args[2])** on the initial naming context, the naming service returns an object reference to the Bank servant.

*Waiting for Invocation*

The previous sections describe the code that makes the server ready; the next section explains the code that enables it to simply wait around for a client to request its service. The following code, which is at the end of (but within) the **try**-**catch** block, shows how to accomplish this.

      java.lang.Object sync = new java.lang.Object();  
      synchronized(sync) {  
          System.out.println("Waiting ...");  
          sync.wait();  
      }

This form of **Object.wait()** requires **Server** to remain alive (though quiescent) until an invocation comes from the ORB. Because of its placement in **main()**, after an invocation completes, the server will wait again.

*Compiling the Server and Servant Classes*

Now we will compile all three classes so that we can correct any error before continuing with this tutorial.

1. Change to your project directory.
2. Run the Java compiler as follows:  
   **javac Server.java BankImpl.java AccountImpl.java bankidl/\*.java**
3. Correct any errors in your file and recompile if necessary.
4. The files **Server.class**, **AccountImpl.class**, and **BankImpl.class** are generated in your project directory.

***4. Implement the client***

Similarly, you use the stubs generated by the **idlj** compiler as the basis of your client application. The client code builds on the stubs to start its ORB, look up the server using the name service provided with Java IDL, obtain a reference for the remote object, and call its method.

*Creating Client.java*

To create **Client.java**,

1. Start your text editor and create a file named **Client.java** in your project directory.
2. Enter the code for **Client.java** in the text file. The following section, Understanding **Client.java**, explains each line of code in some detail.
3. Save and close **Client.java**.

*Understanding Client.java*

This section explains each line of **Client.java**, describing what the code does, as well as why it is needed for this application.

*Performing Basic Setup*

The basic shell of a CORBA client is the same as many Java applications: You import required library packages, declare the application class, define amain method, and handle exceptions.

*Importing Required Packages*

First, we import the packages required for the client class:

import bank.\*;               // The package containing our stubs.  
import org.omg.CosNaming.\*;  // Client will use the naming service.  
import org.omg.CORBA.\*;      // All CORBA applications need these classes.

*Declaring the Client Class*

The next step is to declare the client class:

public class Client {  
  // The main() method goes here.  
}

*Defining a****main()****Method*

Every Java application needs amain() method. It is declared within the scope of theClient class, as follows:

  public static void main(String args[]) {  
    // The try-catch block goes here.  
  }

Our **main()** method invokes first the constructor of the **Client** class, and then **main()** invokes the method **run()**on the object of class **Client**.

*Handling CORBA System Exceptions*

Because all CORBA programs can throw CORBA system exceptions at runtime, all of the **Client(String[] args)**functionality is placed within a **try**-**catch** block. CORBA programs throw system exceptions whenever trouble occurs during any of the processes (marshaling, unmarshaling, upcall) involved in invocation.

Our exception handler simply prints the name of the exception and its stack trace to standard output so you can see what kind of thing has gone wrong.

The **try**-**catch** block is set up inside **Client(String[] args)**,

        try {  
            ...  
        } catch (Exception se) {  
            System.out.println("ERROR : " + e);  
            e.printStackTrace(System.out);  
            System.exit(0);  
        }

*Creating an ORB Object*

A CORBA client needs a local ORB object to perform all of its marshaling and IIOP work. Every client instantiates an **org.omg.CORBA.ORB** object and initializes it by passing to the object certain information about itself.

The ORB variable is declared and initialized inside the **try**-**catch** block.

      ORB orb = ORB.init(args, null);

The call to the ORB's **init()** method passes in your application's command line arguments, allowing you to set certain properties at runtime.

*Finding the Bank Server*

Now that the application has an ORB, it can ask the ORB to locate the actual service it needs, in this case the Bank server. There are a number of ways for a CORBA client to get an initial object reference; our client application will use the COS Naming Service specified by OMG and provided with Java IDL. See Using Stringified Object References for information on how to get an initial object reference when there is no naming service available.

*Obtaining the Initial Naming Context*

The first step in using the naming service is to get the initial naming context. In the **try**-**catch** block, below your ORB initialization, you call **orb.resolve\_initial\_references()** to get an object reference to the name server:

      org.omg.CORBA.Object objRef = orb.resolve\_initial\_references("NameService");

The string **NameService** is defined for all CORBA ORBs. When you pass in that string, the ORB returns the initial naming context, an object reference to the name service.

*Narrowing the Object Reference*

As with all CORBA object references, **objRef** is a generic CORBA object. To use it as a **NamingContext** object, you must narrow it to its proper type.

      NamingContext ncRef = NamingContextHelper.narrow(objRef);

Here we see the use of an **idlj**-generated helper class, similar in function to **BankHelper**. The **ncRef** object is now an **org.omg.CosNaming.NamingContext** and you can use it to access the naming service and find other services. You will do that in the next step.

*Finding a Service in Naming*

Names can have different structures depending upon the implementation of the naming service. Consequently, CORBA name servers handle complex names by way of **NameComponent** objects. Each **NameComponent**holds a single part, or element, of the name. An array of **NameComponent** objects can hold a fully specified path to an object on any computer file or disk system.

To find the Bank server, you first need a **NameComponent** to hold an identifying string for the Bank server.

      NameComponent nc = new NameComponent(bankname, "");

This statement sets theid field ofnc tobankname and thekind field to an empty string. Be sure this is an empty string, do not enter a space between "".

Because the path to the Bank object has just one element, we have created a single-element array out of **nc**. The **NamingContext.resolve()** method requires this array for its work:

      NameComponent path[] = {nc};

Finally, we pass **path** to the naming service's **resolve()** method to get an object reference to the **Bank** server and narrow it to a **Bank** object:

      bankobj = BankHelper.narrow(ncRef.resolve(path));

Here you see the **BankHelper** helper class at work. The **resolve()** method returns a generic CORBA object as you saw above when locating the name service itself. Therefore, you immediately narrow it to a **Bank** object, which is the object reference you need to perform the rest of your work.

*Invoking the****newAccount()****Operation*

CORBA invocations look like a method call on a local object. The complications of marshaling parameters to the wire, routing them to the server-side ORB, unmarshaling, and placing the upcall to the server method are completely transparent to the client programmer. Because so much is done for you by generated code, invocation is really the easiest part of CORBA programming.

      bankobj.newAccount(name);

*Compiling****Client.java***

Now we will compile **Client.java** so that we can correct any errors before continuing with this tutorial.

To compile **Client.java**,

* + 1. Change to your project directory.
    2. Run the Java compiler on **Client.java**:  
       **javac Client.java bankidl/\*.java**
    3. The **Client.class** is generated to your project directory.

***5. Running the Bank Application***

* + 1. Start the Java IDL Name Server. Enter:

tnameserv -ORBInitialPort 1050

* + 1. In this example, port 1050 has been chosen for the naming service. You can change this to a different value if port 1050 is occupied on your system. If the *nameserverport*is not specified, port 900 will be chosen by default. When using Solaris software, you must become root to start a process on a port under 1024. For this reason, we recommend that you use a port number greater than or equal to 1024.
    2. From a second prompt or shell, start the Bank server:

java bankidl.Server -ORBInitialPort 1050 BankOfEstonia

You can leave out **-ORBInitialPort***nameserverport*if the name server is running on the default port.If the Bank server is running on a different host (machine) than the naming service, you would need to specify where the naming service is running when you start the server. To do this, you would use the **-ORBInitialHost***nameserverhost* argument.

* + 1. From a third prompt or shell, run the Bank application client:

java bankidl.Client BankOfEstonia -ORBInitialPort 1050

You can leave out -ORBInitialPort

*nameserverport*

if the name server is running on the default port.

If the Bank client is running on a different host (machine) than the naming service, you would need to specify where the naming service is running when you start the client. To do this, you would use the -ORBInitialHost*nameserverhost* argument.