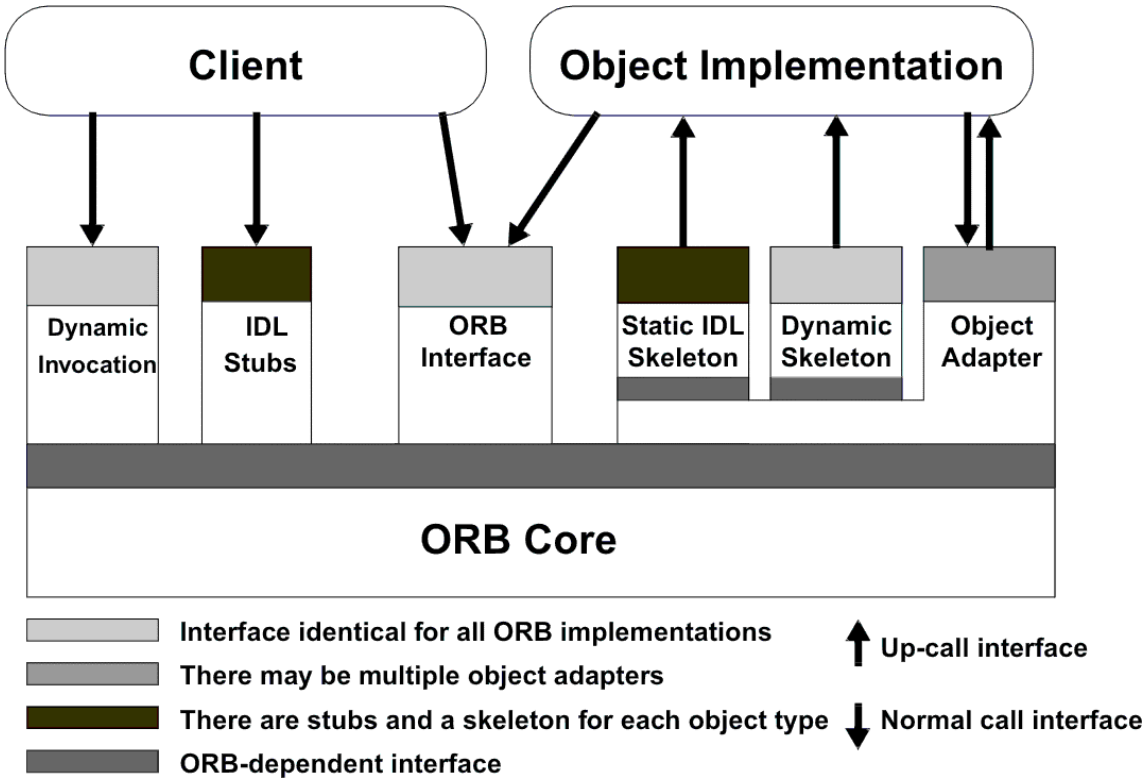

Figure 2-2 The Structure of Object Request Interfaces

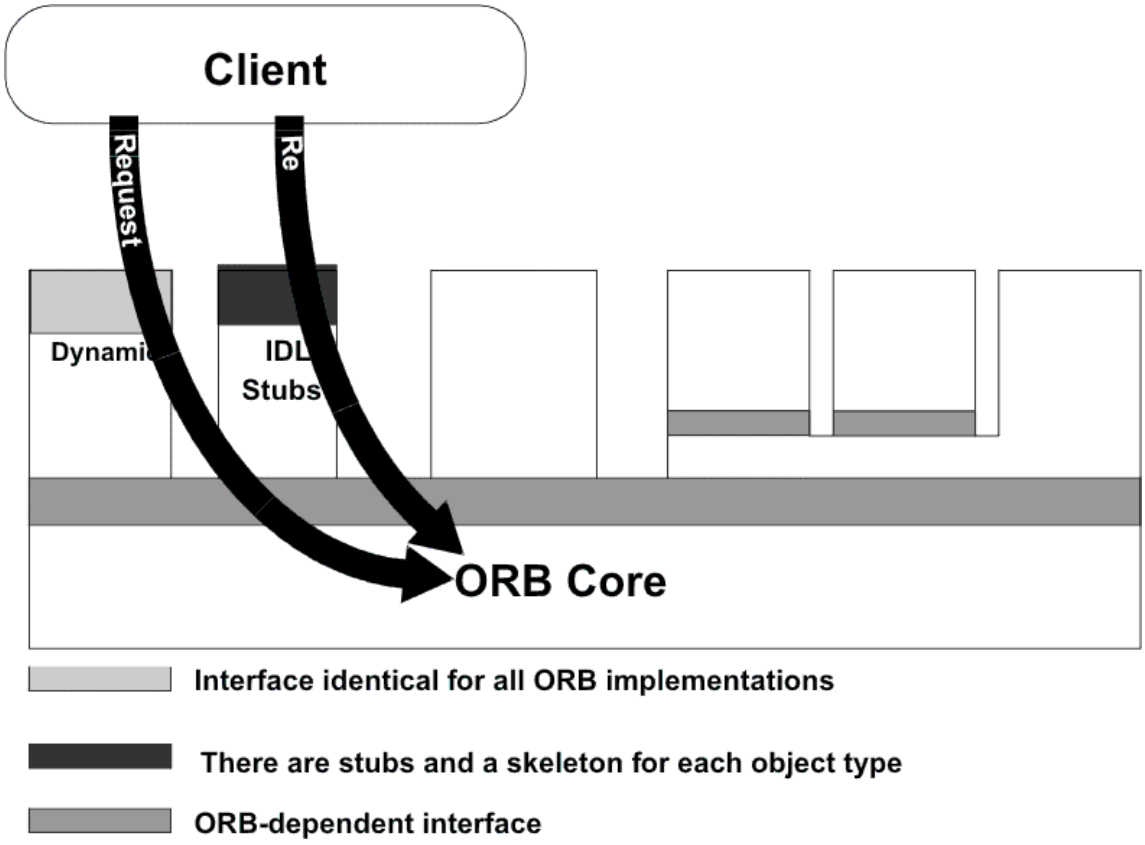

Figure 2-3 A Client Using the Stub or Dynamic Invocation Interface

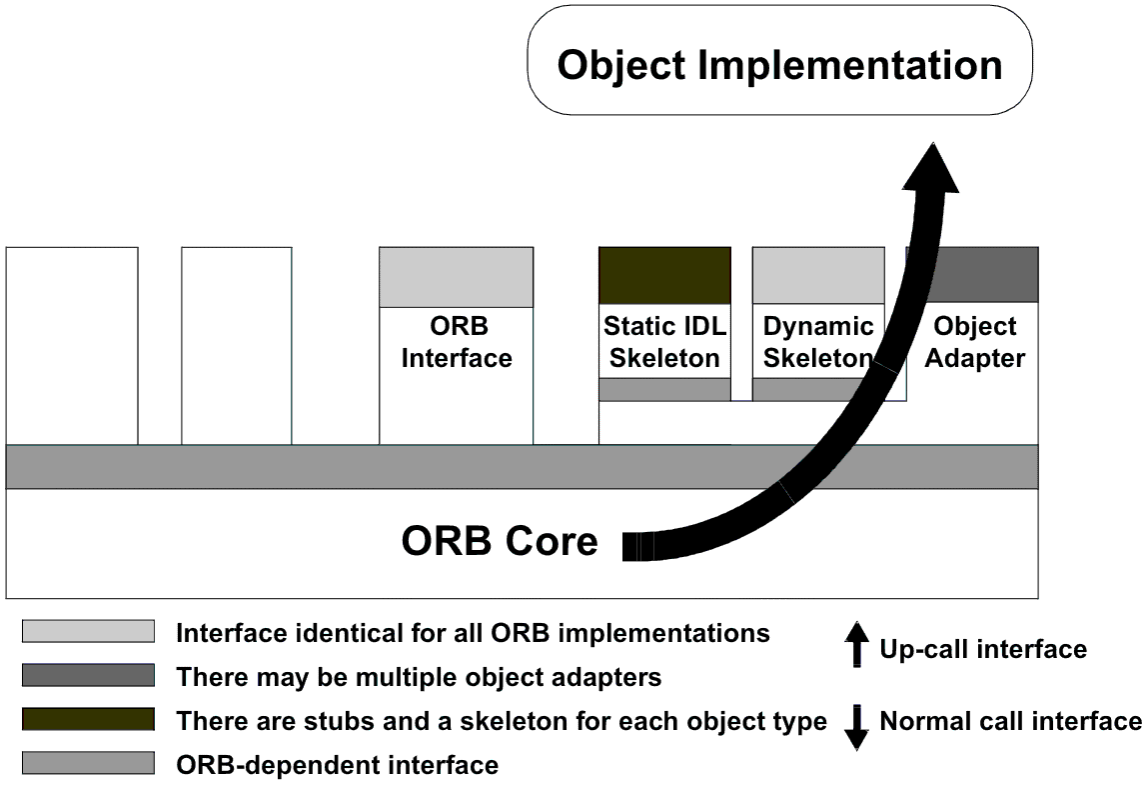

Figure 2-4 An Object Implementation Receiving a Request

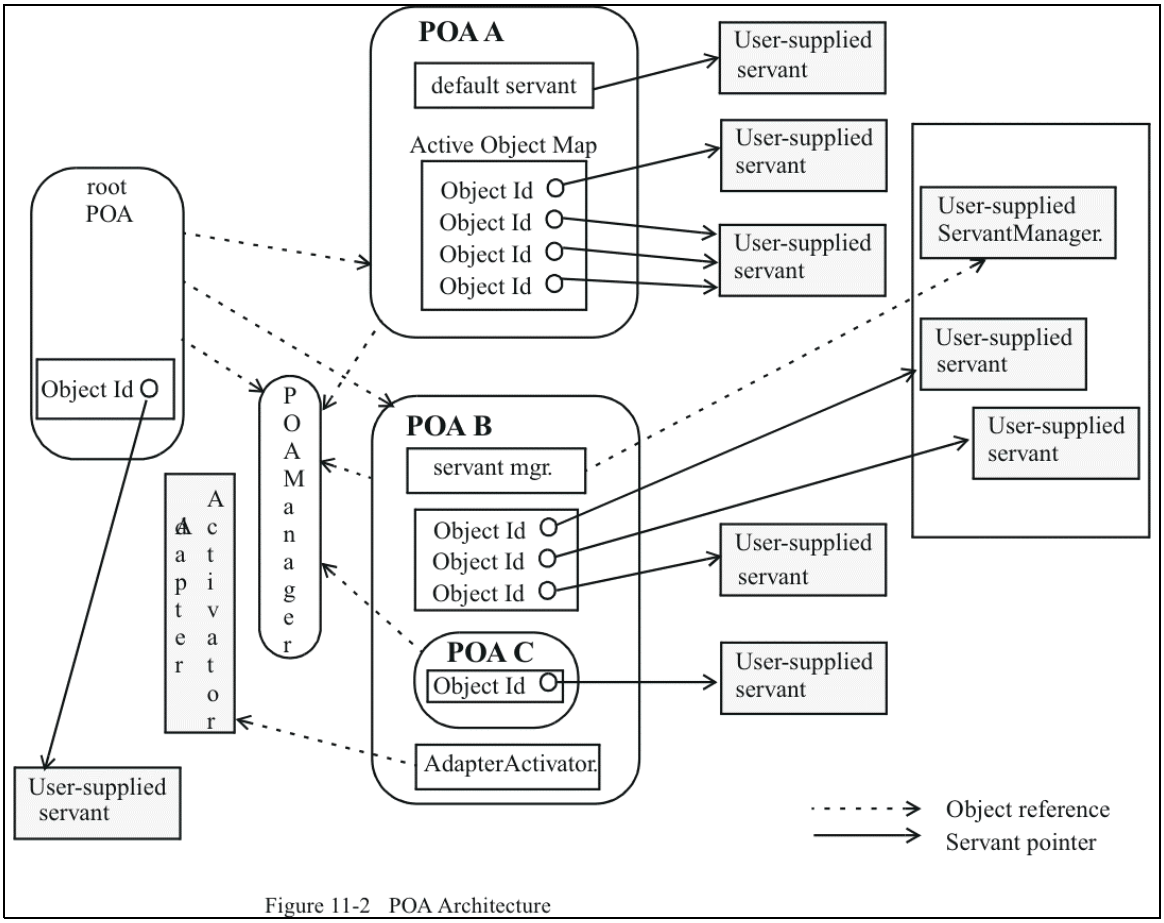

Figure 11-2 POA Architecture

Tworzenie Hello.idl

1. Stwórz nowy katalog, np. Hello.
2. W katalogu tym otwórz do edycji Hello.idl
3. W **Hello.idl** napisz

```
module HelloApp
{
  interface Hello
  {
    string sayHello();
    oneway void shutdown();
  };
};
```

4. Zapamiętaj plik.

Kompilacja Hello.idl

1. Z linii poleceń wydaj komendę:

   idlj -fall Hello.idl

```
//  idlj -fallTie Hello.idl
// -fallTie generuje, HelloPOATie, który służyc będzie do tworzenia Tie.
```

W bieżącym katalogu powstanie katalog HelloApp zawierający 6 plików.
Hello.java jest interfejem sygnatury (*signature interface)* i jest używany jako “signature type” w deklaracjach metod kiedy interfejsy wyspecyfikowanego typu są używane przez inne interfejsy (w poprzednich wersjach kompilatora IDL były tu również operacje):

```
//Hello.java
package HelloApp;

/**
* HelloApp/Hello.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Hello.idl
*/

public interface Hello extends HelloOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity
{
} // interface Hello
```

HelloOperations.java jest interfejsem operacji (*operations interface)* (począwszy od J2SDK v1.3.0)) i zawiera wszystkie operacje interfejsu. Interfejs operacji używany jest “in the server-side mapping “ oraz jako mechanizm pozwalający na optymalizację wywołań dla „co-located clients and servers”

//**HelloOperations.java**

```
package HelloApp;

/**
* HelloApp/HelloOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Hello.idl
*/

public interface HelloOperations
{
  String sayHello ();
  void Shutdown ();
} // interface HelloOperations
```

Understanding the idlj Compiler Output

- HelloPOA.java

  This abstract class is the stream-based server skeleton, providing basic CORBA functionality for the server. It extends org.omg.PortableServer.Servant, and implements the InvokeHandler interface and the HelloOperations interface. The server class, HelloServant, extends HelloPOA.

- _HelloStub.java

  This class is the client stub, providing CORBA functionality for the client. It extends org.omg.CORBA.portable.ObjectImpl and implements the Hello.java interface.

- Hello.java

  This interface contains the Java version of our IDL interface. The Hello.java interface extends org.omg.CORBA.Object, providing standard CORBA object functionality. It also extends the HelloOperations interface and org.omg.CORBA.portable.IDLEntity.

- HelloHelper.java

  This class provides auxiliary functionality, notably the narrow() method required to cast CORBA object references to their proper types. The Helper class is responsible for reading and writing the data type to CORBA streams, and inserting and extracting the data type from Anys. The Holder class delegates to the methods in the Helper class for reading and writing.

- HelloHolder.java

  This final class holds a public instance member of type Hello. Whenever the IDL type is an out or an inout parameter, the Holder class is used. It provides operations for org.omg.CORBA.portable.OutputStream and org.omg.CORBA.portable.InputStream arguments, which CORBA allows, but which do not map easily to Java's semantics. The Holder class delegates to the methods in the Helper class for reading and writing. It implements org.omg.CORBA.portable.Streamable.

- HelloOperations.java

  This interface contains the methods sayHello() and shutdown(). The IDL-to-Java mapping puts all of the operations defined on the IDL interface into this file, which is shared by both the stubs and skeletons.

1. The files generated by the idlj compiler for Hello.idl, with the -fallTie command line option, are:
   - HelloPOATie.java

     The constructor to MyPOATie takes a delegate and/or a poa. You must provide the implementations for delegate and/or poa, but the delegate does not have to inherit from any other class, only the interface HelloOperations. For more information, refer to the *IDL to Java Language Mapping Specification*.

Table 3-6 Keywords

abstract	double	long	readonly	unsigned
any	enum	module	sequence	union
attribute	exception	native	short	ValueBase
boolean	factory	Object	string	valuetype
case	FALSE	octet	struct	void
char	fixed	oneway	supports	wchar
const	float	out	switch	wstring
context	in	private	TRUE	
custom	inout	public	truncatable	
default	interface	raises	typedef	

Table 1-1 Basic Type Mappings

IDL Type	Java type	Exceptions
boolean	`boolean`	
char	`char`	CORBA::DATA_CONVERSION
wchar	`char`	CORBA::DATA_CONVERSION
octet	`byte`	
string	`java.lang.String`	CORBA::MARSHAL CORBA::DATA_CONVERSION
wstring	`java.lang.String`	CORBA::MARSHAL CORBA::DATA_CONVERSION
short	`short`	
unsigned short	`short`	
long	`int`	
unsigned long	`int`	
long long	`long`	
unsigned long long	`long`	
float	`float`	
double	`double`	
fixed	`java.math.BigDecimal`	CORBA::DATA_CONVERSION

IDL declarations that collide with the following methods on java.lang.Object (from the Java Language Specification 1.0 First Edition, Section 20.1):

```
clone equals finalize getClass hashCode notify notifyAll toString wait
```

HelloServer.java

```
// HelloServer.java
// Copyright and License
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

import java.util.Properties;

class HelloImpl extends HelloPOA {
  private ORB orb;

  public void setORB(ORB orb_val) {
    orb = orb_val;
  }

  // implement sayHello() method
  public String sayHello() {
    return "\nHello world !!\n";
  }

  // implement shutdown() method
  public void shutdown() {
    orb.shutdown(false);
  }
}


public class HelloServer {

  public static void main(String args[]) {
    try{
      // create and initialize the ORB
      ORB orb = ORB.init(args, null);

      // get reference to rootpoa & activate the POAManager
      POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();

      // create servant and register it with the ORB
      HelloImpl helloImpl = new HelloImpl();
      helloImpl.setORB(orb);

      // get object reference from the servant
      org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
      Hello href = HelloHelper.narrow(ref);
```

```
// create a tie, with servant being the delegate.
///     HelloPOATie tie = new HelloPOATie(helloImpl, rootpoa);

      // obtain the objectRef for the tie
      // this step also implicitly activates the
      // the object
///     Hello href = tie._this(orb);


  // get the root naming context
      // NameService invokes the name service
      org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
      // Use NamingContextExt which is part of the Interoperable
      // Naming Service (INS) specification.
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      // bind the Object Reference in Naming
      String name = "Hello";
      NameComponent path[] = ncRef.to_name( name );
      ncRef.rebind(path, href);

      System.out.println("HelloServer ready and waiting ...");

      // wait for invocations from clients
      orb.run();
    }

      catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
      }

      System.out.println("HelloServer Exiting ...");

  }
}
```

HelloClient.java

```
// Copyright and License

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

public class HelloClient
{
 static Hello helloImpl;

 public static void main(String args[])
   {
    try{
      // create and initialize the ORB
         ORB orb = ORB.init(args, null);

      // get the root naming context
      org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
      // Use NamingContextExt instead of NamingContext. This is
      // part of the Interoperable naming Service.
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      // resolve the Object Reference in Naming
      String name = "Hello";
      helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));

      System.out.println("Obtained a handle on server object: " + helloImpl);
      System.out.println(helloImpl.sayHello());
      helloImpl.shutdown();

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

         }
```

start orbd -ORBInitialPort 1050 -ORBInitialHost *servermachinename*

idlj - The IDL-to-Java Compiler

idlj generates Java bindings from a given IDL file.

Synopsis

idlj [options] *idl-file*
where *idl-file* is the name of a file containing Interface Definition Language (IDL) definitions. *Options* may appear in any order, but must precede the *idl-file*.

Description

The IDL-to-Java Compiler generates the Java bindings for a given IDL file. For binding details, see the OMG IDL to Java Language Language Mapping Specification. Some previous releases of the IDL-to-Java compiler were named idltojava.

Emitting Client and Server Bindings

To generate Java bindings for an IDL file named My.idl:

idlj My.idl

This generates the client-side bindings and is equivalent to:

idlj **-fclient** My.idl

The client-side bindings do not include the server-side skeleton. If you want to generate the server-side bindings for the interfaces:

idlj **-fserver** My.idl

Server-side bindings include the client-side bindings plus the skeleton, all of which are POA (that is, Inheritance Model) classes. If you want to generate both client and server-side bindings, use one of the following (equivalent) commands:

idlj **-fclient -fserver** My.idl
idlj **-fall** My.idl

There are two possible server-side models: the Inheritance Model and the Tie Delegation Model.
NEW in 1.4! The default server-side model is the *Portable Servant Inheritance Model*. Given an interface My defined in My.idl, the file MyPOA.java is generated. You must provide the implementation for My and it must inherit from MyPOA.
MyPOA.java is a stream-based skeleton that extends org.omg.PortableServer.Servant and implements the InvokeHandler interface and the operations interface associated with the IDL interface the skeleton implements.

The PortableServer module for the Portable Object Adapter (POA) defines the native Servant type. In the Java programming language, the Servant type is mapped to the Java org.omg.PortableServer.Servant class. It serves as the base class for all POA servant implementations and provides a number of methods that may be invoked by the application programmer, as well as methods which are invoked by the POA itself and may be overridden by the user to control aspects of servant behavior.
Another option for the Inheritance Model is to use the -oldImplBase flag in order to generate server-side bindings that are compatible with older version of the Java programming language (prior to J2SE 1.4). Note that using the -oldImplBase flag is non-standard: these APIs are being deprecated. You would use this flag ONLY for compatibility with existing servers written in J2SE 1.3. In that case, you would need to modify an existing MAKEFILE to add the -oldImplBase flag to the idlj compiler, otherwise POA-based server-side mappings will be generated. To generate server-side bindings that are backwards compatible:

idlj **-fclient -fserver -oldImplBase** My.idl
idlj **-fall -oldImplBase** My.idl

Given an interface My defined in My.idl, the file _MyImplBase.java is generated. You must provide the implementation for My and it must inherit from _MyImplBase.

The other server-side model is called the Tie Model. This is a delegation model. Because it is not possible to generate ties and skeletons at the same time, they must be generated separately. The following commands generate the bindings for the Tie Model:

idlj **-fall** My.idl
idlj **-fallTIE** My.idl

For the interface My, the second command generates MyPOATie.java. The constructor to MyPOATie takes a delegate. In this example, using the default POA model, the constructor also needs a poa. You must provide the implementation for delegate, but it does not have to inherit from any other class, only the interface MyOperations. But to use it with the ORB, you must wrap your implementation within MyPOATie. For instance:

```
ORB orb = ORB.init(args, System.getProperties());

// Get reference to rootpoa & activate the POAManager
POA rootpoa = (POA)orb.resolve_initial_references("RootPOA");
rootpoa.the_POAManager().activate();

// create servant and register it with the ORB
MyServant myDelegate = new MyServant();
myDelegate.setORB(orb);

// create a tie, with servant being the delegate.
MyPOATie tie = new MyPOATie(myDelegate, rootpoa);

// obtain the objectRef for the tie
```

```
My ref = tie._this(orb);
```

You might want to use the Tie model instead of the typical Inheritance model if your implementation must inherit from some other implementation. Java allows any number of interface inheritance, but there is only one slot for class inheritance. If you use the inheritance model, that slot is used up . By using the Tie Model, that slot is freed up for your own use. The drawback is that it introduces a level of indirection: one extra method call occurs when invoking a method.

To generate server-side, Tie model bindings that are compatible with older version of the IDL to Java language mapping in versions of J2SE before 1.4.

idlj **-oldImplBase -fall** My.idl
idlj **-oldImplBase -fallTIE** My.idl

For the interface My, this will generate My_Tie.java. The constructor to My_Tie takes a impl. You must provide the implementation for impl, but it does not have to inherit from any other class, only the interface HelloOperations. But to use it with the ORB, you must wrap your implementation within My_Tie. For instance:

```
ORB orb = ORB.init(args, System.getProperties());

// create servant and register it with the ORB
MyServant myDelegate = new MyServant();
myDelegate.setORB(orb);

// create a tie, with servant being the delegate.
MyPOATie tie = new MyPOATie(myDelegate);

// obtain the objectRef for the tie
My ref = tie._this(orb);
```

Specifying Alternate Locations for Emitted Files

If you want to direct the emitted files to a directory other than the current directory, invoke the compiler as:

idlj **-td /altdir** My.idl

For the interface My, the bindings will be emitted to /altdir/My.java, etc., instead of ./My.java.

Specifying Alternate Locations for Include Files

If My.idl included another idl file, MyOther.idl, the compiler assumes that MyOther.idl resides in the local directory. If it resides in /includes, for example, then you would invoke the compiler with the following command:

idlj **-i /includes** My.idl

If My.idl also included Another.idl that resided in /moreIncludes, for example, then you would invoke the compiler with the following command:

idlj **-i /includes -i /moreIncludes** My.idl

Since this form of include can become irritatingly long, another means of indicating to the compiler where to search for included files is provided. This technique is similar to the idea of an environment variable. Create a file named idl.config in a directory that is listed in your CLASSPATH. Inside of idl.config, provide a line with the following form:

includes=/includes;/moreIncludes

The compiler will find this file and read in the includes list. Note that in this example the separator character between the two directories is a semicolon (;). This separator character is platform dependent. On NT it is a semicolon, on Solaris it is a colon, etc. For more information on includes, read the CLASSPATH (Solaris) or CLASSPATH (Windows) documentation.

Emitting Bindings for Include Files

By default, only those interfaces, structs, etc, that are defined in the idl file on the command line have Java bindings generated for them. The types defined in included files are not generated. For example, assume the following two idl files:

My.idl

```
#include <MyOther.idl>
interface My
{
};
```

MyOther.idl

```
interface MyOther
{
};
```

The following command will only generate the java bindings for My:

idlj My.idl

To generate all of the types in My.idl and all of the types in the files that My.idl includes (in this example, MyOther.idl), use the following command:

idlj **-emitAll** My.idl

There is a caveat to the default rule. #include statements which appear at global scope are treated as described. These #include statements can be thought of as import statements. #include statements which appear within some enclosing scope are treated as true #include statements, meaning that the code within the included file is treated as if it appeared in the original file and, therefore, Java bindings are emitted for it. Here is an example:

My.idl

```
#include <MyOther.idl>
interface My
{
  #include <Embedded.idl>
};
```

MyOther.idl

```
interface MyOther
{
};
```

Embedded.idl

```
enum E {one, two, three};
```

Running the following command:

idlj My.idl

will generate the following list of Java files:

./MyHolder.java
./MyHelper.java
./_MyStub.java
./MyPackage
./MyPackage/EHolder.java

./MyPackage/EHelper.java
./MyPackage/E.java
./My.java

Notice that MyOther.java was not generated because it is defined in an import-like #include. But E.java *was* generated because it was defined in a true #include. Also notice that since Embedded.idl was included within the scope of the interface My, it appears within the scope of My (that is,in MyPackage).

If the *-emitAll* flag had been used in the previous example, then all types in all included files would be emitted.

Inserting Package Prefixes

Suppose that you work for a company named ABC that has constructed the following IDL file:

Widgets.idl

```
module Widgets
{
  interface W1 {...};
  interface W2 {...};
};
```

Running this file through the IDL-to-Java compiler will place the Java bindings for W1 and W2 within the package Widgets. But there is an industry convention that states that a company's packages should reside within a package named com.<company name>. The Widgets package is not good enough. To follow convention, it should be com.abc.Widgets. To place this package prefix onto the Widgets module, execute the following:

idlj **-pkgPrefix Widgets com.abc** Widgets.idl

If you have an IDL file which includes Widgets.idl, the *-pkgPrefix* flag must appear in that command also. If it does not, then your IDL file will be looking for a Widgets package rather than a com.abc.Widgets package.

If you have a number of these packages that require prefixes, it might be easier to place them into the idl.config file described above. Each package prefix line should be of the form:

PkgPrefix.<type>=<prefix>

So the line for the above example would be:

PkgPrefix.Widgets=com.abc

The use of this option does not affect the Repository ID.

Defining Symbols Before Compilation

You may need to define a symbol for compilation that is not defined within the IDL file, perhaps to include debugging code in the bindings. The command

idlj **-d** MYDEF My.idl

is the equivalent of putting the line #define MYDEF inside My.idl.

Preserving Pre-Existing Bindings

If the Java binding files already exist, the *-keep* flag will keep the compiler from overwriting them. The default is to generate all files without considering if they already exist. If you've customized those files (which you should not do unless you are very comfortable with their contents), then the *-keep* option is very useful. The command

idlj **-keep** My.idl

emit all client-side bindings that do not already exist.

Viewing Progress of Compilation

The IDL-to-Java compiler will generate status messages as it progresses through its phases of execution. Use the -v option to activate this "verbose" mode:

idlj **-v** My.idl

By default the compiler does not operate in verbose mode.

Displaying Version Information

To display the build version of the IDL-to-Java compiler, specify the *-version* option on the command-line:
idlj -version
Version information also appears within the bindings generated by the compiler. Any additional options appearing on the command-line are ignored.

Options

-d *symbol*
This is equivalent to the following line in an IDL file:
#define *symbol*
-emitAll
Emit all types, including those found in #include files.
-f*side*

Defines what bindings to emit. *side* is one of client, server, serverTIE, all, or allTIE. The -fserverTIE and -fallTIE options cause delegate model skeletons to be emitted. Assumes -fclient if the flag is not specified.

-i *include-path*

By default, the current directory is scanned for included files. This option adds another directory.

-keep

If a file to be generated already exists, do not overwrite it. By default it is overwritten.

-noWarn

Suppresses warning messages.

-oldImplBase

Generates skeletons compatible with old (pre-1.4) JDK ORBs. By default, the POA Inheritance Model server-side bindings are generated. This option provides backward-compatibility with older versions of the Java programming language by generating server-side bindings that are ImplBase Inheritance Model classes.

-pkgPrefix *type prefix*

Wherever *type* is encountered at file scope, prefix the generated Java package name with *prefix* for all files generated for that type. The *type* is the simple name of either a top-level module, or an IDL type defined outside of any module.

-pkgTranslate *type package*

Whenever the module name *type* is encountered in an identifier, replace it in the identifier with *package* for all files in the generated Java package. Note that pkgPrefix changes are made first. *type* is the simple name of either a top-level module, or an IDL type defined outside of any module, and must match the full package name exactly.

If more than one translation matches an identifier, the longest match is chosen. For example, if the arguments include:

-pkgTranslate foo bar -pkgTranslate foo.baz buzz.fizz

The following translations would occur:

foo => bar
foo.boo => bar.boo
foo.baz => buzz.fizz
foo.baz.bar => buzz.fizz.bar

The following package names cannot be translated:

- org
- org.omg or any subpackages of org.omg

Any attempt to translate these packages will result in uncompilable code, and the use of these packages as the first argument after -pkgTranslate will be treated as an error.

-skeletonName *xxx%yyy*

Use *xxx%yyy* as the pattern for naming the skeleton. The defaults are:

- %POA for the POA base class (-fserver or -fall)
- _%ImplBase for the oldImplBase class (-oldImplBase and (-fserver or -fall))

-td *dir*

Use *dir* for the output directory instead of the current directory.

-tieName *xxx%yyy*

Name the tie according to the pattern. The defaults are:

- %POATie for the POA tie base class (-fserverTie or -fallTie)
- %_Tie for the oldImplBase tie class (-oldImplBase and (-fserverTie or -fallTie))

-nowarn, -verbose
Verbose mode.
-version
Display version information and terminate.

See the Description section for more option information.

Restrictions:

- Escaped identifiers in the global scope may not have the same spelling as IDL primitive types, Object, or ValueBase. This is because the symbol table is pre-loaded with these identifiers; allowing them to be redefined would overwrite their original definitions. (Possible permanent restriction).
- The fixed IDL type is not supported.

Known Problems:

- No import generated for global identifiers. If you invoke on an unexported local impl, you do get an exception, but it seems to be due to a Null Ptr Exception in the ServerDelegate DSI code.

PersistentHello.idl

```
module Persistent {
   interface Hello {
      string sayHello( );
      oneway void shutdown();
   };
};
```

To complete the application, you simply provide the server (PersistentServer.java), servant (PersistentHelloServant.java), and client (PersistentClient.java) implementations

PersistentServer.java

```
// PersistentServer.java
// Copyright and License
import java.util.Properties;
import org.omg.CORBA.Object;
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.CORBA.Policy;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.*;
import org.omg.PortableServer.Servant;

public class PersistentServer {

   public static void main( String args[] ) {
      Properties properties = System.getProperties();
      properties.put( "org.omg.CORBA.ORBInitialHost",
         "localhost" );
      properties.put( "org.omg.CORBA.ORBInitialPort",
         "1050" );

      try {
         // Step 1: Instantiate the ORB
         ORB orb = ORB.init(args, properties);

         // Step 2: Instantiate the servant
         PersistentHelloServant servant = new PersistentHelloServant(orb);

         // Step 3 : Create a POA with Persistent Policy
         // ******************
         // Step 3-1: Get the rootPOA
         POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // Step 3-2: Create the Persistent Policy
         Policy[] persistentPolicy = new Policy[1];
         persistentPolicy[0] = rootPOA.create_lifespan_policy(
            LifespanPolicyValue.PERSISTENT);
         // Step 3-3: Create a POA by passing the Persistent Policy
```

```
            POA persistentPOA = rootPOA.create_POA("childPOA", null,
                persistentPolicy );
            // Step 3-4: Activate PersistentPOA's POAManager, Without this
            // All calls to Persistent Server will hang because POAManager
            // will be in the 'HOLD' state.
            persistentPOA.the_POAManager().activate( );
            // **********************

            // Step 4: Associate the servant with PersistentPOA
            persistentPOA.activate_object( servant );

            // Step 5: Resolve RootNaming context and bind a name for the
            // servant.
            // NOTE: If the Server is persistent in nature then using Persistent
            // Name Service is a good choice. Even if ORBD is restarted the Name
            // Bindings will be intact. To use Persistent Name Service use
            // 'NameService' as the key for resolve_initial_references() when
            // ORBD is running.
            org.omg.CORBA.Object obj = orb.resolve_initial_references(
                "NameService" );
            NamingContextExt rootContext = NamingContextExtHelper.narrow( obj );

            NameComponent[] nc = rootContext.to_name(
                "PersistentServerTutorial" );
            rootContext.rebind( nc, persistentPOA.servant_to_reference(
                servant ) );

            // Step 6: We are ready to receive client requests
            orb.run();
        } catch ( Exception e ) {
            System.err.println( "Exception in Persistent Server Startup " + e );
        }
    }
}
```

Implementing the Servant (PersistentHelloServant.java)

The example servant, PersistentHelloServant, is the implementation of the Hello IDL interface; each Hello instance is implemented by a PersistentHelloServant instance. The servant is a subclass of HelloPOA, which is generated by the idlj compiler from the example IDL. The servant contains one method for each IDL operation, in this example, the sayHello() and shutdown() methods. Servant methods are just like ordinary Java methods; the extra code to deal with the ORB, with marshaling arguments and results, and so on, is provided by the skeleton.

PersistentHelloServant.java

```
// PersistentHelloServant.java
// Copyright and License
import org.omg.CORBA.ORB;

public class PersistentHelloServant extends Persistent.HelloPOA {
   private ORB orb;

   public PersistentHelloServant( ORB orb ) {
      this.orb = orb;
   }

   /**
    *  sayHello() method implementation returns a simple message.
    */
   public String sayHello( ) {
      return "Hello From Persistent Server...";
   }

   /**
    *  shutdown() method shuts down the Persistent Server.
    *  See NOTE below.
    */
   public void shutdown( ) {
      orb.shutdown( false );
   }
}
```

Note: For convenience of presentation in this example, the shutdown() method is included as part of the servant. This has been done in order to demonstrate the persistence of the server in this example. This is not a recommended programming convention for the following reasons:

- If the orb.shutdown() method is called with parameter true (meaning "wait for completion") within the implementation of a remote method, the ORB will hang in a deadlock. Other threads can invoke orb.shutdown() without deadlock.
- If you have multiple servants associated with the ORB, using the shutdown(false) method by one of them will make all of them unavailable.
- The orb.shutdown(false) method should be called as part of the SERVER code under more controlled circumstances.

PersistentClient.java

```
// Copyright and License

import java.util.Properties;
import org.omg.CORBA.ORB;
import org.omg.CORBA.OBJ_ADAPTER;
import org.omg.CosNaming.NamingContext;
import org.omg.CosNaming.NamingContextHelper;
import org.omg.CosNaming.NameComponent;
import org.omg.PortableServer.POA;

import Persistent.HelloHelper;
import Persistent.Hello;

public class PersistentClient {

   public static void main(String args[]) {

      try {
         // Step 1: Instantiate the ORB
         ORB orb = ORB.init(args, null);

         // Step 2: Resolve the PersistentHelloServant by using INS's
         // corbaname url. The URL locates the NameService running on
         // localhost and listening on 1050 and resolve
         // 'PersistentServerTutorial' from that NameService
         org.omg.CORBA.Object obj = orb.string_to_object(
            "corbaname::localhost:1050#PersistentServerTutorial");

         Hello hello = HelloHelper.narrow( obj );

         // Step 3: Call the sayHello() method every 60 seconds and shutdown
         // the server. Next call from the client will restart the server,
         // because it is persistent in nature.
         while( true ) {
            System.out.println( "Calling Persistent Server.." );
            String helloFromServer = hello.sayHello();
            System.out.println("Message From Persistent Server: " +
               helloFromServer );
            System.out.println( "Shutting down Persistent Server.." );
            hello.shutdown( );
            Thread.sleep( 60000 );
         }
      } catch ( Exception e ) {
         System.err.println( "Exception in PersistentClient.java..." + e );
         e.printStackTrace( );
      }
   }
}
```

1. Start orbd.

   To start orbd, enter:
   orbd -ORBInitialPort 1050 -serverPollingTime 200
   Note that 1050 is the port on which you want the name server to run. The -ORBInitialPort argument is a required command-line argument. Note that 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.
   The -serverPollingTime 200 argument specifies how often ORBD checks for the health of persistent servers registered via servertool. The default value is 1,000 ms. We are setting this parameter to 200 ms in this example to enable more frequent monitoring of failures. In the event that a server failure is detected, the server will be restarted to its proper state.

2. Start the Hello server:

   To register a persistent server with the ORBD, the server must be started using *servertool*, which is a command-line interface for application programmers to register, unregister, startup, and shutdown a persistent server. When the servertool is started, you must specify the port and the host (if different) on which orbd is executing.
   To start the Hello server,
   1. Start the servertool from the command line as follows:
   2. servertool -ORBInitialPort 1050
      Make sure the name server (orbd) port is the same as in the previous step, for example, -ORBInitialPort 1050. The servertool must be started on the same port as the name server.
      The servertool command line interface appears:

   3. Register the PersistentServer from the servertool prompt, as shown below. Type the information in one long string without returns.
   4. servertool > register -server PersistentServer -applicationName s1
   5. -classpath *path_to_server_class_files*
      The servertool registers the server, assigns it the name of "s1", and displays its server id.

3. Run the client application:
4. java -classpath . PersistentClient

   The terminal window or DOS prompt displays the following messages:
   Calling Persistent Server..
   Message From Persistent Server: Hello From Persistent Server...
   Shutting down Persistent Server..
   Calling Persistent Server..
   Message From Persistent Server: Hello From Persistent Server...

Shutting down Persistent Server..

In this example, the client invokes the sayHello() method every minute and then kills the persistent server, which will be automatically restarted the next time the client invokes the sayHello() method.