## NAME

idlj − Generates Java bindings for a specified Interface Definition Language (IDL) file.

## SYNOPSIS

**idlj** [ *options* ] *idlfile*

*options* The command-line options. See Options. Options can appear in any order, but must precede the **idlfile**.

*idlfile* The name of a file that contains Interface Definition Language (IDL) definitions.

## DESCRIPTION

The IDL-to-Java Compiler generates the Java bindings for a specified IDL file. For binding details, see Java IDL: IDL to Java Language Mapping at http://docs.oracle.com/javase/8/docs/technotes/guides/idl/mapping/jidlMapping.html

Some earlier releases of the IDL-to-Java compiler were named **idltojava**.

### EMIT CLIENT AND SERVER BINDINGS

The following **idlj** command generates an IDL file named **My.idl** with client-side bindings.

**idlj My.idl**

The previous syntax is equivalent to the following:

**idlj −fclient My.idl**

The next example generates the server-side bindings, and includes the client-side bindings plus the skeleton, all of which are POA (Inheritance Model).

**idlg −fserver My.idl**

If you want to generate both client and server-side bindings, then use one of the following (equivalent) commands:

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

There are two possible server-side models: the Portal Servant Inheritance Model and the Tie Model. See Tie Delegation Model.

**Portable Servant Inheritance Model**. 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 the **My** interface, and the **My** interface must inherit from the **MyPOA** class. **MyPOA.java** is a stream-based skeleton that extends the **org.omg.PortableServer.Servant** class at http://docs.oracle.com/javase/8/docs/api/org/omg/PortableServer/Servant.html The **My** interface implements the **callHandler** 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. See Portable Object Adapter (POA) at http://docs.oracle.com/javase/8/docs/technotes/guides/idl/POA.html 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 can be called by the application programmer, and methods that are called by the POA and that can be overridden by the user to control aspects of servant behavior.Another option for the Inheritance Model is to use the **-oldImplBase** flag to generate server-side bindings that are compatible with releases of the Java programming language before Java SE 1.4. The **-oldImplBase** flag is nonstandard, and these APIs are deprecated. You would use this flag only for compatibility with existing servers written in Java SE 1.3. In that case, you would need to modify an existing make file to add the **-oldImplBase** flag to the **idlj** compiler. Otherwise POA-based server-side

mappings are generated. To generate server-side bindings that are backward compatible, do the following:

**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 the **My** interface, and the **My** interface must inherit from the **_MyImplBase** class.

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

```
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 when 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, then that slot is used up. With 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 a method is called.

For server-side generation, Tie model bindings that are compatible with versions of the IDL to Java language mapping in versions earlier than Java SE 1.4.

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

For the **My** interface, the this generates **My_Tie.java**. The constructor to the **My_Tie** class takes an **impl** object. 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 example:

```
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);
```

### SPECIFY ALTERNATE LOCATIONS FOR EMITTED FILES

If you want to direct the emitted files to a directory other than the current directory, then call the compiler this way: **idlj -td /altdir My.idl**.

For the **My** interface, the bindings are emitted to **/altdir/My.java**, etc., instead of **./My.java**.

### SPECIFY ALTERNATE LOCATIONS FOR INCLUDE FILES

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

```
idlj −i /includes −i /moreIncludes My.idl
```

Because this form of **include** can become long, another way to indicate 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** variable. 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 the Windows platform, use a semicolon, on the Unix platform, use a colon, and so on.

### EMIT BINDINGS FOR INCLUDE FILES

By default, only those interfaces, structures, and so on, 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 file:
#include <MyOther.idl>
interface My
{
};
MyOther.idl file:
interface MyOther
{
};
```

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

```
My.idl file:
#include <MyOther.idl>
interface My
```

**{**
**#include <Embedded.idl>**
**};**
**MyOther.idl file:**
**interface MyOther**
**{**
**};**
**Embedded.idl**
**enum E {one, two, three};**

Run**idlj My.idl**to generate the following list of Java files. Notice that **MyOther.java** is not generated because it is defined in an import-like **#include**. But **E.java** was generated because it was defined in a true **#include**. Notice that because the **Embedded.idl** file is included within the scope of the interface **My**, it appears within the scope of **My** (in **MyPackage**). If the **-emitAll** flag had been used, then all types in all included files would have been emitted.

**./MyHolder.java**
**./MyHelper.java**
**./_MyStub.java**
**./MyPackage**
**./MyPackage/EHolder.java**
**./MyPackage/EHelper.java**
**./MyPackage/E.java**
**./My.java**

## INSERT PACKAGE PREFIXES

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

**Widgets.idl file:**
**module Widgets**
**{**
**interface W1 {...};**
**interface W2 {...};**
**};**

If you run this file through the IDL-to-Java compiler, then the Java bindings for W1 and W2 are placed within the **Widgets** package. There is an industry convention that states that a company's packages should reside within a package named **com.<company name>**. To follow this convention, the package name 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 that includes Widgets.idl, then 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, then it might be easier to place them into the idl.config file described previously. Each package prefix line should be of the form: **PkgPrefix.<type>=<prefix>**. The line for the previous example would be **PkgPrefix.Widgets=com.abc**. This option does not affect the Repository ID.

## DEFINE SYMBOLS BEFORE COMPILATION

You might 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 equivalent to putting the line **#define MYDEF** inside My.idl.

**PRESERVE PREEXISTING BINDINGS**

If the Java binding files already exist, then the **-keep** flag keeps the compiler from overwriting them. The default is to generate all files without considering that they already exist. If you have 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** emits all client-side bindings that do not already exist.

**VIEW COMPILATION PROGRESS**

The IDL-to-Java compiler generates status messages as it progresses through its phases of execution. Use the **-v** option to activate the verbose mode: **idlj -v My.idl**.

By default the compiler does not operate in verbose mode

**DISPLAY 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*

-demitAll

Emit all types, including those found in **#include** files.

-fside

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

-i *include-path*

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

-i *keep*

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

-noWarn

Suppress warning messages.

-oldImplBase

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

-pkgPrefix *typeprefix*

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 *typepackage*

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. The type value 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, then the longest match is chosen as shown in the

following example:

*Command*:

**pkgTranslate type pkg −pkgTranslate type2.baz pkg2.fizz**

*Resulting Translation*:

**type => pkg**
**type.ext => pkg.ext**
**type.baz => pkg2.fizz**
**type2.baz.pkg => pkg2.fizz.pkg**

The following package names **org**, **org**.o**mg**, or any subpackages of **org.omg** cannot be translated. Any attempt to translate these packages results in uncompilable code, and the use of these packages as the first argument after **-pkgTranslate** is 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**), and **_%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*
Use **xxx%yyy** according to the pattern. The defaults are: **%POA** for the **POA** base class (**-fserverTie or -fallTie**), and **_%Tie** for the **oldImplBase** tie class (**-oldImplBase**) and (**-fserverTie** or **-fallTie**))

-nowarn, -verbose
Displays release information and terminates.

-version
Displays release information and terminates.

## RESTRICTIONS

Escaped identifiers in the global scope cannot have the same spelling as IDL primitive types, **Object**, or **ValueBase**. This is because the symbol table is preloaded 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 is generated for global identifiers. If you call an unexported local **impl** object, then you do get an exception, but it seems to be due to a **NullPointerException** in the **ServerDelegate** DSI code.