Writing an Ice Application with Java





This page shows how to create an Ice application with Java.

On this page:

- Compiling a Slice Definition for Java
- Writing and Compiling a Server in Java
- Writing and Compiling a Client in Java
- Running Client and Server in Java

Compiling a Slice Definition for Java

The first step in creating our Java application is to compile our Slice definition to generate Java proxies and skeletons. You can compile the definition as follows:

```
$ mkdir generated
$ slice2java --output-dir generated Printer.ice
```

The --output-dir option instructs the compiler to place the generated files into the generated directory. This avoids cluttering the working directory with the generated files. The slice2java compiler produces a number of Java source files from this definition. The exact contents of these files do not concern us for now — they contain the generated code that corresponds to the Printer interface we defined in Printer.ice.

Back to Top ^

Writing and Compiling a Server in Java

To implement our Printer interface, we must create a servant class. By convention, a servant class uses the name of its interface with an I-suffix, so our servant class is called PrinterI and placed into a source file PrinterI.java:

Java

```
public class PrinterI implements Demo.Printer
{
    public void printString(String s, com.zeroc.Ice.Current current)
    {
        System.out.println(s);
    }
}
```

The PrinterI class implements the interface Printer, which is generated by the slice2java compiler. The interface defines a printString method that accepts a string for the printer to print and a parameter of type Current. (For now we will ignore the Current parameter.) Our implementation of the printString method simply writes its argument to the terminal.

The remainder of the server code is in a source file called Server. java, shown in full here:

```
Java
public class Server
{
    public static void main(String[] args) throws Exception
    {
        try(com.zeroc.Ice.Communicator communicator = com.zeroc.Ice.Util.initialize(args))
        {
            com.zeroc.Ice.ObjectAdapter adapter = communicator.createObjectAdapterWithEndpoints
("SimplePrinterAdapter", "default -p 10000");
            com.zeroc.Ice.Object object = new PrinterI();
            adapter.add(object, com.zeroc.Ice.Util.stringToIdentity("SimplePrinter"));
            adapter.activate();
            communicator.waitForShutdown();
        }
    }
}
```

The main method is declared to throw Exception. If the code throws an exception, it will be handled by the JVM which typically prints out the exception and then returns failure to the operating system. The body of main contains a try-with-resources block in which we place all the server code.

The Communicator object implements java.lang.AutoCloseable, which allows us to use the try-with-resources statement for the initialization of the Communicator object. This ensures the communicator destroy method is called when the try block goes out of scope. Doing this is essential in order to correctly finalize the lce run time.

A Failure to call destroy on the communicator before the program exits results in undefined behavior.

The body of our try block contains the actual server code.

The code goes through the following steps:

- 1. We initialize the lce run time by calling com.zeroc.lce.Util.initialize. (We pass args to this call because the server may have command-line arguments that are of interest to the run time; for this example, the server does not require any command-line arguments.) The call to initialize returns a Communicator reference, which is the main object in the lce run time.
- 2. We create an object adapter by calling createObjectAdapterWithEndpoints on the Communicator instance. The arguments we pass are "SimplePrinterAdapter" (which is the name of the adapter) and "default -p 10000", which instructs the adapter to listen for incoming requests using the default protocol (TCP/IP) at port number 10000.
- 3. At this point, the server-side run time is initialized and we create a servant for our Printer interface by instantiating a PrinterI object.
- 4. We inform the object adapter of the presence of a new servant by calling add on the adapter; the arguments to add are the servant we have just instantiated, plus an identifier. In this case, the string "SimplePrinter" is the name of the Ice object. (If we had multiple printers, each would have a different name or, more correctly, a different object identity.)
- 5. Next, we activate the adapter by calling its activate method. (The adapter is initially created in a holding state; this is useful if we have many servants that share the same adapter and do not want requests to be processed until after all the servants have been instantiated.)
- 6. Finally, we call waitForShutdown. This call suspends the calling thread until the server implementation terminates, either by making a call to shut down the run time, or in response to a signal. (For now, we will simply interrupt the server on the command line when we no longer need it.)

Note that, even though there is quite a bit of code here, that code is essentially the same for all servers. You can put that code into a helper class and, thereafter, will not have to bother with it again. (Ice provides such a helper class, called Application.) As far as actual application code is concerned, the server contains only a few lines: seven lines for the definition of the PrinterI class, plus three lines to instantiate a PrinterI object and register it with the object adapter.

We can compile the server code as follows:

```
$ mkdir classes
```

\$ javac -d classes -classpath classes:\$ICE_HOME/lib/ice.jar Server.java PrinterI.java generated/Demo/*.java

This compiles both our application code and the code that was generated by the Slice compiler. We assume that the ICE_HOME environment variable is set to the top-level directory containing the Ice run time. (For example, if you have installed Ice in /opt/Ice, set ICE_HOME to that path.) Note that Ice for Java uses Gradle to control building of source code. (Gradle is similar to make, but more flexible for Java applications.) You can have a look at the demo code that ships with Ice to see how to use this tool.

Back to Top ^

Writing and Compiling a Client in Java

The client code, in Client.java, looks very similar to the server. Here it is in full:

```
Java
public class Client
{
    public static void main(String[] args)
    {
        try(com.zeroc.Ice.Communicator communicator = com.zeroc.Ice.Util.initialize(args))
        {
            com.zeroc.Ice.ObjectPrx base = communicator.stringToProxy("SimplePrinter:default -p 10000");
            Demo.PrinterPrx printer = Demo.PrinterPrx.checkedCast(base);
            if(printer == null)
            {
                throw new Error("Invalid proxy");
            }
            printer.printString("Hello World!");
        }
    }
}
```

Note that the overall code layout is the same as for the server: we use the same try and catch blocks to deal with errors. The code in the try block does the following:

- 1. As for the server, we initialize the lce run time by calling com.zeroc.Ice.Util.initialize within the Java try-with-resources statement.
- 2. The next step is to obtain a proxy for the remote printer. We create a proxy by calling stringToProxy on the communicator, with the string "Sim plePrinter:default -p 10000". Note that the string contains the object identity and the port number that were used by the server. (Obviously, hard-coding object identities and port numbers into our applications is a bad idea, but it will do for now; we will see more architecturally sound ways of doing this when we discuss lceGrid.)
- 3. The proxy returned by stringToProxy is of type com.zeroc.Ice.ObjectPrx, which is at the root of the inheritance tree for interfaces and classes. But to actually talk to our printer, we need a proxy for a Printer interface, not an Object interface. To do this, we need to do a down-cast by calling PrinterPrx.checkedCast. A checked cast sends a message to the server, effectively asking "is this a proxy for a Printer interface?" If so, the call returns a proxy of type Demo::Printer; otherwise, if the proxy denotes an interface of some other type, the call returns null.
- 4. We test that the down-cast succeeded and, if not, throw an error message that terminates the client.
- 5. We now have a live proxy in our address space and can call the printString method, passing it the time-honored "Hello World!" string. The server prints that string on its terminal.

Compiling the client looks much the same as for the server:

\$ javac -d classes -classpath classes:\$ICE_HOME/lib/ice.jar Client.java PrinterI.java generated/Demo/*.java

Back to Top ^

Running Client and Server in Java

To run client and server, we first start the server in a separate window:

\$ java Server

At this point, we won't see anything because the server simply waits for a client to connect to it. We run the client in a different window:

```
$ java Client
$
```

The client runs and exits without producing any output; however, in the server window, we see the "Hello World!" that is produced by the printer. To get rid of the server, we interrupt it on the command line for now. (We will see cleaner ways to terminate a server in our discussion of the Application class.)

If anything goes wrong, the client will print an error message. For example, if we run the client without having first started the server, we get something like the following:

```
com.zeroc.Ice.ConnectionRefusedException
      error = 0
          at ...
          at Client.run(Client.java:65)
Caused by: java.net.ConnectException: Connection refused
         . . .
```

Note that, to successfully run client and server, your CLASSPATH must include the Ice library and the classes directory, for example:

\$ export CLASSPATH=\$CLASSPATH:./classes:\$ICE_HOME/lib/ice.jar

Please have a look at the demo applications that ship with Ice for the details for your platform.

See Also

- Client-Side Slice-to-Java Mapping
- Server-Side Slice-to-Java Mapping
- Application Helper Class
 The Current Object







Back to Top ^