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

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

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

Whenever we show Unix commands, we assume a Bourne or Bash shell. The commands for Windows are essentially identical and therefore not shown.

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`.

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 extends Demo._PrinterDisp {
    public void
    printString(String s, Ice.Current current)
    {
        System.out.println(s);
    }
}
```

The `PrinterI` class inherits from a base class called `_PrinterDisp`, which is generated by the `slice2java` compiler. The base class is abstract and contains a `printString` method that accepts a string for the printer to print and a parameter of type `Ice.Current`. (For now we will ignore the `Ice.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)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            Ice.ObjectAdapter adapter =
                ic.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000");
            Ice.Object object = new PrinterI();
            adapter.add(object, ic.stringToIdentity("SimplePrinter"));
            adapter.activate();
            ic.waitForShutdown();
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

Note the general structure of the code:

Java

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {

            // Server implementation here...

        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

The body of `main` contains a `try` block in which we place all the server code, followed by two `catch` blocks. The first block catches all exceptions that may be thrown by the Ice run time; the intent is that, if the code encounters an unexpected Ice run-time exception anywhere, the stack is unwound all the way back to `main`, which prints the exception and then returns failure to the operating system. The second block catches `Exception` exceptions; the intent is that, if we encounter a fatal error condition somewhere in our code, we can simply throw an exception with an error message. Again, this unwinds the stack all the way back to `main`, which prints the error message and then returns failure to the operating system.

Before the code exits, it destroys the communicator (if one was created successfully). Doing this is essential in order to correctly finalize the Ice run time: the program *must* call `destroy` on any communicator it has created; otherwise, undefined behavior results.

The body of our `try` block contains the actual server code:

Java

```
            ic = Ice.Util.initialize(args);
            Ice.ObjectAdapter adapter =
                ic.createObjectAdapterWithEndpoints("SimplePrinterAdapter", "default -p 10000");
            Ice.Object object = new PrinterI();
            adapter.add(object, ic.stringToIdentity("SimplePrinter"));
            adapter.activate();
            ic.waitForShutdown();
```

The code goes through the following steps:

1. We initialize the Ice run time by calling `Ice.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 an `Ice.Communicator` reference, which is the main object in the Ice 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 servant. (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 `Ice.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 the `ant` build environment to control building of source code. (`ant` 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.

Writing and Compiling a Client in Java

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

```
public class Client {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            Ice.ObjectPrx base = ic.stringToProxy("SimplePrinter:default -p 10000");
            Demo.PrinterPrx printer = Demo.PrinterPrxHelper.checkedCast(base);
            if (printer == null)
                throw new Error("Invalid proxy");

            printer.printString("Hello World!");
        } catch (Ice.LocalException e) {
            e.printStackTrace();
            status = 1;
        } catch (Exception e) {
            System.err.println(e.getMessage());
            status = 1;
        }
        if (ic != null) {
            // Clean up
            //
            try {
                ic.destroy();
            } catch (Exception e) {
                System.err.println(e.getMessage());
                status = 1;
            }
        }
        System.exit(status);
    }
}
```

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 Ice run time by calling `Ice.Util.initialize`.
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 `"SimplePrinter: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 IceGrid.)
3. The proxy returned by `stringToProxy` is of type `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 `PrinterPrxHelper.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
```

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 `Ice.Application`.)

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:

```
Ice.ConnectionRefusedException
       error = 0
           at IceInternal.ConnectRequestHandler.getConnection(ConnectRequestHandler.java:240)
           at IceInternal.ConnectRequestHandler.sendRequest(ConnectRequestHandler.java:138)
           at IceInternal.Outgoing.invoke(Outgoing.java:66)
           at Ice._ObjectDelM.ice_isA(_ObjectDelM.java:30)
           at Ice.ObjectPrxHelperBase.ice_isA(ObjectPrxHelperBase.java:111)
           at Ice.ObjectPrxHelperBase.ice_isA(ObjectPrxHelperBase.java:77)
           at Demo.HelloPrxHelper.checkedCast(HelloPrxHelper.java:228)
           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