Writing an Ice Application with Java

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

On this page:

Create Projects for your Client and Server Applications

We will use Gradle to create our application projects. You must install Gradle before continuing with this tutorial.

Open a new Command Prompt and run the following commands to generate a new project:

```
mkdir printer
cd printer
gradle init
```

For this demo we're going to use a project with two sub-projects to build the Client and Server applications. The requirements for our sub-projects are the same so we'll do all the setup in the `subprojects` block of the root project, which applies to all sub-projects. Edit the generated `build.gradle` file to look like the one below:

build.gradle

```
//
// Install the gradle Ice Builder plug-in from the plug-in portal
//
plugins {
    id 'com.zeroc.gradle.ice-builder.slice' version '1.4.7' apply false
}

subprojects {
    //
    // Apply Java and Ice Builder plug-ins to all sub-projects
    //
    apply plugin: 'java'
    apply plugin: 'com.zeroc.gradle.ice-builder.slice'

    //
    // Both Client and Server projects share the Printer.ice Slice definitions
    //
    slice {
        java {
            files = [file("../Printer.ice")]
        }
    }

    //
    // Use Ice JAR files from maven central repository
    //
    repositories {
        mavenCentral()
    }

    //
    // Both Client and Server depend only on Ice JAR
    //
    dependencies {
        implementation 'com.zeroc:ice:3.7.2'
    }

    //
    // Create a JAR file with the appropriate Main-Class and Class-Path attributes
    //
    jar {
        manifest {
            attributes(
                "Main-Class": project.name.capitalize(),
                "Class-Path": configurations.runtime.resolve().collect { it.toURI() }.join(' ')
            )
        }
    }
}
```

We must also edit the generated `settings.gradle` to define our sub-projects:

settings.gradle

```
rootProject.name = 'printer'
include 'client'
include 'server'
```

Finally we need to create the directories for client and server projects:

```
mkdir client
mkdir server
```

Compiling a Slice Definition for Java

The next step is to add the Slice file (`Printer.ice`), and then compile this Slice file. When building the project, the `sliceCompile` task (added automatically by the Ice Builder plug-in) compiles `Printer.ice` and places the generated code into `build/generated-src` using the Slice to Java compiler, `slice2java`.

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 `server/src/main/java/PrinterI.java`:

server/src/main/java/PrinterI.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/src/main/java/Server.java`, shown in full here:

server/src/main/java/Server.java

```
public class Server
{
    public static void main(String[] args)
    {
        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 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 Ice run time.

> A communicator starts a number of non-background threads. Destroying the communicator terminates all these threads.

The body of our `try` block contains the actual server code. The code goes through the following steps:

1. We initialize the Ice run time by calling `com.zeroc.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 a `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 transport 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 is shut down (For now, we will simply interrupt the server on the command line when we no longer need it, which terminates the server immediately.)

We can compile the server code as follows:

```
gradlew :server:build
```

Writing and Compiling a Client in Java

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

client/src/main/java/Client.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 Ice 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 `"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 `com.zeroc.Ice.ObjectPrx`, which is at the root of the inheritance tree for interfaces. 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:

```
gradlew :client:build
```

Running Client and Server in Java

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

```
java -jar server/build/libs/server.jar
```

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 -jar client/builds/libs/client.jar
```

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.

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
         ...
```

See Also

- Client-Side Slice-to-Java Mapping
- Server-Side Slice-to-Java Mapping
- The Current Object
- IceGrid