The Server-Side main Method in C-Sharp

On this page:

- A Basic Main Method in C#
- The Ice.Application Class in C#
 - Using Ice.Application on the Client Side in C#
 - Catching Signals in C#
 - Ice.Application and Properties in C#
 - Limitations of Ice.Application in C#

A Basic Main Method in C#

The main entry point to the Ice run time is represented by the local interface Ice::Communicator. As for the client side, you must initialize the Ice run time by calling Ice.Util.initialize before you can do anything else in your server. Ice.Util.initialize returns a reference to an instance of an Ice.Communicator:

Ice.Util.initialize accepts the argument vector that is passed to Main by the operating system. The method scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when Ice.Util. initialize returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, initialize throws an exception.

Before leaving your Main method, you must call Communicator.destroy. The destroy operation is responsible for finalizing the Ice run time. In particular, destroy waits for any operation implementations that are still executing in the server to complete. In addition, destroy ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory. Never allow your Main method to terminate without calling destroy first; doing so has undefined behavior.

The general shape of our server-side ${\tt Main}$ method is therefore as follows:

C#

```
using System;
public class Server
    public static void Main(string[] args)
        int status = 0;
        Ice.Communicator communicator = null;
        try {
            communicator = Ice.Util.initialize(ref args);
        } catch (Exception ex) {
            Console.Error.WriteLine(ex);
            status = 1;
        if (communicator != null) {
            try {
                communicator.destroy();
            } catch (Exception ex) {
                Console.Error.WriteLine(ex);
                status = 1;
        Environment.Exit(status);
    }
}
```

Note that the code places the call to Ice.Util.initialize into a try block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

The Ice. Application Class in C#

The preceding structure for the Main method is so common that Ice offers a class, Ice.Application, that encapsulates all the correct initialization and finalization activities. The synopsis of the class is as follows (with some detail omitted for now):

C#

The intent of this class is that you specialize Ice.Application and implement the abstract run method in your derived class. Whatever code you would normally place in Main goes into the run method instead. Using Ice.Application, our program looks as follows:

```
C#
```

```
using System;

public class Server
{
    class App : Ice.Application
    {
        public override int run(string[] args)
        {
            // Server code here...
            return 0;
        }
    }

    public static void Main(string[] args)
    {
        App app = new App();
        Environment.Exit(app.main(args));
    }
}
```

Note that Application.main is overloaded: you can pass an optional file name or an InitializationData structure.

If you pass a configuration file name to main, the property settings in this file are overridden by settings in a file identified by the ICE_CONFIG environment variable (if defined). Property settings supplied on the command line take precedence over all other settings.

The Application.main method does the following:

- 1. It installs an exception handler for System. Exception. If your code fails to handle an exception, Application. main prints the name of the exception and a stack trace on Console. Error before returning with a non-zero return value.
- 2. It initializes (by calling Ice.Util.initialize) and finalizes (by calling Communicator.destroy) a communicator. You can get access to the communicator for your server by calling the static communicator accessor.
- 3. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your run method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.
- 4. It provides the name of your application via the static appName method. You can get at the application name from anywhere in your code by calling Ice.Application.appName (which is usually required for error messages).
- 5. It installs a signal handler that properly destroys the communicator.
- 6. It installs a per-process logger if the application has not already configured one. The per-process logger uses the value of the Ice.

 ProgramName property as a prefix for its messages and sends its output to the standard error channel. An application can also specify an alt ernate logger.

Using Ice.Application ensures that your program properly finalizes the lce run time, whether your server terminates normally or in response to an exception. We recommend that all your programs use this class; doing so makes your life easier. In addition, Ice.Application also provides features for signal handling and configuration that you do not have to implement yourself when you use this class.

Using Ice. Application on the Client Side in C#

You can use Ice.Application for your clients as well: simply implement a class that derives from Ice.Application and place the client code into its run method. The advantage of this approach is the same as for the server side: Ice.Application ensures that the communicator is destroyed correctly even in the presence of exceptions.

Catching Signals in C#

The simple server we developed in Hello World Application had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

To make it easier to deal with signals, Ice. Application encapsulates the low-level signal handling tasks, allowing you to cleanly shut down on receipt of a signal.

C#

The methods behave as follows:

destroyOnInterrupt

This method installs a handler that destroys the communicator if it is interrupted. This is the default behavior.

• shutdownOnInterrupt

This method installs a handler that shuts down the communicator if it is interrupted.

• ignoreInterrupt

This method causes signals to be ignored.

• callbackOnInterrupt

This method configures Ice. Application to invoke interruptCallback when a signal occurs, thereby giving the subclass responsibility for handling the signal.

• holdInterrupt

This method temporarily blocks signal delivery.

releaseInterrupt

This method restores signal delivery to the previous disposition. Any signal that arrives after holdInterrupt was called is delivered when you call releaseInterrupt.

interrupted

This method returns true if a signal caused the communicator to shut down, false otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal. This is useful, for example, for logging purposes.

• interruptCallback

A subclass overrides this method to respond to signals. The method may be called concurrently with any other thread and must not raise exceptions.

By default, Ice. Application behaves as if destroyOnInterrupt was invoked, therefore our server Main method requires no change to ensure that the program terminates cleanly on receipt of a signal. (You can disable the signal-handling functionality of Ice. Application by passing the enumerator NoSignalHandling to the constructor. In that case, signals retain their default behavior, that is, terminate the process.) However, we add a diagnostic to report the occurrence, so our run method now looks like:

C#

Ice. Application and Properties in C#

Apart from the functionality shown in this section, Ice.Application also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or port number for a server. The main method of Ice.Application is overloaded; the second version allows you to specify the name of a configuration file that will be processed during initialization.

Limitations of Ice. Application in C#

Ice. Application is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use Ice. Application. Instead, you must structure your code as we saw in Hello World Application (taking care to always destroy the communicator).

See Also

- Hello World Application
- Properties and Configuration
- Communicator Initialization
- Logger Facility