The Server-Side main Method in Java

On this page:

A Basic `main` Method in Java

The main entry point to the Ice run time is represented by the local Slice 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`:

Java

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // ...
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        }
        // ...
    }
}
```

`Ice.Util.initialize` accepts the argument vector that is passed to `main` by the operating system. The function scans the argument vector for any command-line options that are relevant to the Ice run time, but does not remove those options. If anything goes wrong during initialization, `initialize` throws an exception.

The semantics of Java arrays prevents `Ice.Util.initialize` from modifying the size of the argument vector. However, another overloading of `Ice.Util.initialize` is provided that allows the application to obtain a new argument vector with the Ice options removed.

Before leaving your `main` function, 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` function to terminate without calling `destroy` first; doing so has undefined behavior.

The general shape of our server-side `main` function is therefore as follows:

Java

```
public class Server {
    public static void
    main(String[] args)
    {
        int status = 0;
        Ice.Communicator ic = null;
        try {
            ic = Ice.Util.initialize(args);
            // ...
        } catch (Exception e) {
            e.printStackTrace();
            status = 1;
        }
        if (ic != null) {
            try {
                ic.destroy();
            } catch (Exception e) {
                e.printStackTrace();
                status = 1;
            }
        }
        System.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 Java

The preceding structure for the `main` function 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):

Java

```
package Ice;

public enum SignalPolicy { HandleSignals, NoSignalHandling }

public abstract class Application {
    public Application()

    public Application(SignalPolicy signalPolicy)

    public final int main(String appName, String[] args)

    public final int main(String appName, String[] args, String configFile)

    public final int main(String appName, String[] args, InitializationData initData)

    public abstract int run(String[] args)

    public static String appName()

    public static Communicator communicator()

    // ...
}
```

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:

Java

```
public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Server code here...

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

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` function does the following:

1. It installs an exception handler for `java.lang.Exception`. If your code fails to handle an exception, `Application.main` prints the name of an exception and a stack trace on `System.err` 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` member function. The return value from this call is the first argument in the call to `Application.main`, so you can get at this name from anywhere in your code by calling `Ice.Application.appName` (which is usually required for error messages). In the example above, the return value from `appName` would be `Server`.
5. It installs a shutdown hook that properly shuts down 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 alternate logger.

Using `Ice.Application` ensures that your program properly finalizes the Ice 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 Java

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 Java

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.

Java does not provide direct support for signals, but it does allow an application to register a *shutdown hook* that is invoked when the JVM is shutting down. There are several events that trigger JVM shutdown, such as a call to `System.exit` or an interrupt signal from the operating system, but the shutdown hook is not provided with the reason for the shut down.

`Ice.Application` registers a shutdown hook by default, allowing you to cleanly terminate your application prior to JVM shutdown.

Java

```
package Ice;

public abstract class Application {
    // ...

    synchronized public static void destroyOnInterrupt()
    synchronized public static void shutdownOnInterrupt()
    synchronized public static void setInterruptHook(Thread t)
    synchronized public static void defaultInterrupt()
    synchronized public static boolean interrupted()
}
```

The functions behave as follows:

- `destroyOnInterrupt`
  This function installs a shutdown hook that calls `destroy` on the communicator. This is the default behavior.
- `shutdownOnInterrupt`
  This function installs a shutdown hook that calls `shutdown` on the communicator.
- `setInterruptHook`
  This function installs a custom shutdown hook that takes responsibility for performing whatever action is necessary to terminate the application. Refer to the Java documentation for `Runtime.addShutdownHook` for more information on the semantics of shutdown hooks.
- `defaultInterrupt`
  This function removes the shutdown hook.
- `interrupted`
  This function returns true if the shutdown hook caused the communicator to shut down, false otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by the JVM. This is useful, for example, for logging purposes.

By default, `Ice.Application` behaves as if `destroyOnInterrupt` was invoked, therefore our server `main` function requires no change to ensure that the program terminates cleanly on JVM shutdown. (You can disable this default shutdown hook by passing the enumerator `NoSignalHandling` to the constructor. In that case, shutdown is not intercepted and terminates the VM.) However, we add a diagnostic to report the occurrence, so our `main` function now looks like:

Java

```
public class Server extends Ice.Application {
    public int
    run(String[] args)
    {
        // Server code here...

        if (interrupted())
            System.err.println(appName() + ": terminating");

        return 0;
    }

    public static void
    main(String[] args)
    {
        Server app = new Server();
        int status = app.main("Server", args);
        System.exit(status);
    }
}
```

During the course of normal execution, the JVM does not terminate until all non-daemon threads have completed. If an interrupt occurs, the JVM ignores the status of active threads and terminates as soon as it has finished invoking all of the installed shutdown hooks.

In a subclass of `Ice.Application`, the default shutdown hook (as installed by `destroyOnInterrupt`) blocks until the application's main thread completes. As a result, an interrupted application may not terminate successfully if the main thread is blocked. For example, this can occur in an interactive application when the main thread is waiting for console input. To remedy this situation, the application can install an alternate shutdown hook that does not wait for the main thread to finish:

Java

```
public class Server extends Ice.Application {
    class ShutdownHook extends Thread {
        public void
        run()
        {
            try
            {
                communicator().destroy();
            }
            catch(Ice.LocalException ex)
            {
                ex.printStackTrace();
            }
        }
    }

    public int
    run(String[] args)
    {
        setInterruptHook(new ShutdownHook());

        // ...
    }
}
```

After replacing the default shutdown hook using `setInterruptHook`, the JVM will terminate as soon as the communicator is destroyed.

`Ice.Application` and Properties in Java

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` function 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 Java

`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