Application Helper Class

Previous Next

On this page:

The Ice `Application` Helper Class

Ice provides an `Application` helper class for most programming languages. This helper class initializes the Ice run time then runs your application's code. It also makes sure the Ice run time is properly finalized when an exception is thrown or when the application receives a signal such a keyboard interrupt.

Application is defined as follows (with some detail omitted for now):

C++11

```
namespace Ice
{
    enum class SignalPolicy : unsigned char { HandleSignals, NoSignalHandling };

    class Application
    {
    public:

        Application(SignalPolicy = SignalPolicy::HandleSignals);
        virtual ~Application();

        int main(int argc, const char* const argv[], const InitializationData& initData = InitializationData(),
int version = ICE_INT_VERSION);
        int main(int argc, const char* const argv[], const std::string& configFile, int version =
ICE_INT_VERSION);

#ifdef _WIN32
        int main(int argc, const wchar_t* const argv[], const InitializationData& initData =
InitializationData(), int version = ICE_INT_VERSION);
        int main(int argc, const wchar_t* const argv[], const std::string& configFile, int version =
ICE_INT_VERSION);
#endif

        int main(const StringSeq& args, const InitializationData& initData = InitializationData(), int version
= ICE_INT_VERSION);
        int main(const StringSeq& args, const std::string& configFile, int version = ICE_INT_VERSION);

        virtual int run(int argc, char* argv[]) = 0;

        static const char* appName();

        static std::shared_ptr<Communicator> communicator();
        ...
    };
}
```

C++98

```
namespace Ice
{
    enum class SignalPolicy { HandleSignals, NoSignalHandling };

    class Application
    {
    public:

        Application(SignalPolicy = HandleSignals);
        virtual ~Application();

        int main(int argc, const char* const argv[], const InitializationData& initData = InitializationData(),
int version = ICE_INT_VERSION);
        int main(int argc, const char* const argv[], const char* configFile, int version = ICE_INT_VERSION);

#ifdef _WIN32
        int main(int argc, const wchar_t* const argv[], const InitializationData& initData =
InitializationData(), int version = ICE_INT_VERSION);
        int main(int argc, const wchar_t* const argv[], const char* configFile, int version = ICE_INT_VERSION);
#endif

        int main(const StringSeq& args, const InitializationData& initData = InitializationData(), int version
= ICE_INT_VERSION);
        int main(const StringSeq& args, const char* configFile, int version = ICE_INT_VERSION);

        virtual int run(int argc, char* argv[]) = 0;

        static const char* appName();

        static CommunicatorPtr communicator();
        ...
    };
}
```

C#

```
namespace Ice
{
    public abstract class Application
    {
        public abstract int run(string[] args);

        public Application();

        public Application(SignalPolicy signalPolicy);

        public int main(string[] args);
        public int main(string[] args, string configFile);
        public int main(string[] args, InitializationData init);

        public static string appName();

        public static Communicator communicator();

        ...
    }
}
```

Java

```
package com.zeroc.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()

    // ...
}
```

Java Compat

```
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()

    // ...
}
```

Python

```
# in Ice module

class Application(object):

    def __init__(self, signalPolicy=0):

    def main(self, args, configFile=None, initData=None):

    def run(self, args):

    def appName():
        # ...
    appName = staticmethod(appName)

    def communicator():
        # ...
    communicator = staticmethod(communicator)
```

Ruby

```
module Ice
    class Application
        def main(args, configFile=nil, initData=nil)

        def run(args)

        def Application.appName()

        def Application.communicator()
    end
end
```

The intent of this class is that you specialize `Application` and implement the pure virtual function or abstract method `run` in your derived class. Whatever code you would normally place in `main` goes into the `run` function/method instead. With Ice `Application`, the `main` of a typical Ice-based application becomes:

C++11

```
#include <Ice/Ice.h>

class MyApplication : public Ice::Application
{
public:

    virtual int run(int argc, char* argv[]) override
    {
        // application code goes here...

        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

C++98

```
#include <Ice/Ice.h>

class MyApplication : public Ice::Application
{
public:

    virtual int run(int argc, char* argv[])
    {
        // application code goes here...

        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

C#

```
using System;

public class Server
{
    class App : Ice.Application
    {
        public override int run(string[] args)
        {
            // application code goes here...

            return 0;
        }
    }

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

Java

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

        return 0;
    }

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

Java Compat

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

        return 0;
    }

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

Python

```
import sys, Ice

class Server(Ice.Application):
    def run(self, args):
        # Application code goes here...
        return 0

app = Server()
status = app.main(sys.argv)
sys.exit(status)
```

Ruby

```
require 'Ice'

class Client < Ice::Application
    def run(args)
        # Client code here...
        return 0
    end
end

app = Client.new()
status = app.main(ARGV)
exit(status)
```

Note that the main function or method of `Application` is overloaded: you can pass a string sequence instead of the main program's arguments. This is useful if you need to parse application-specific property settings on the command line. You also can call `main` with 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 or method does the following:

1. It installs an exception handler for Ice exceptions. If your code fails to handle an Ice exception, `Application main` prints the exception details on `stderr` before returning with a non-zero return value.
2. In C++, it installs an exception handler for strings. This allows you to terminate your application in response to a fatal error condition by throwing a `std::string` or `const char*`. `Ice::Application` prints the string on `stderr` before returning a non-zero return value.
3. It initializes (by calling `Ice initialize`) and finalizes (by calling `Communicator::destroy`) a communicator. You can get access to the communicator for your application by calling the static `communicator()` member function or method.
4. 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` function or method is free of Ice-related options and only contains options and arguments that are specific to your application.
5. It sets the property `Ice.ProgramName` to the name of your application, and provides this name via the static `appName` member function or method.
6. It installs a signal handler that properly destroys the communicator; in Java, it installs a Shutdown Hook.
7. 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 `Application` ensures that your program properly finalizes the Ice run time, whether your application terminates normally or in response to an exception or signal.

Application and Threads

`Application` should be considered single-threaded until you call `main`. Then, in `run`, you can call concurrently `Application`'s functions or methods from multiple threads. If you create threads in `run`, you must join them before or when `run` returns.

The thread you use to call `main` is used to initialize the `Communicator` and then to call `run`. On Linux and macOS with C++, this thread must be the only thread of the process at the time you call `main` for signal handling to operate correctly.

Back to Top ^

Catching CTRL-C and Other Signals

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, `Application` catches CTRL-C and similar signals, and allows you to select how to handle them.

C++11

Signals Caught

Linux and macOS: `SIGINT`, `SIGHUP`, `SIGTERM`

Windows: `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, `CTRL_SHUTDOWN_EVENT`

```
namespace Ice
{
    class Application
    {
    public:
        static void destroyOnInterrupt();
        static void shutdownOnInterrupt();
        static void ignoreInterrupt();
        static void callbackOnInterrupt();
        static void holdInterrupt();
        static void releaseInterrupt();
        static bool interrupted();

        virtual void interruptCallback(int signal);
    };
}
```

C++98

Signals Caught

Linux and macOS: `SIGINT`, `SIGHUP`, `SIGTERM`

Windows: `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, `CTRL_SHUTDOWN_EVENT`

```
namespace Ice
{
    class Application
    {
    public:
        static void destroyOnInterrupt();
        static void shutdownOnInterrupt();
        static void ignoreInterrupt();
        static void callbackOnInterrupt();
        static void holdInterrupt();
        static void releaseInterrupt();
        static bool interrupted();

        virtual void interruptCallback(int signal);
    };
}
```

C#

```
namespace Ice
{
    public abstract class Application
    {
        // ...

        public static void destroyOnInterrupt();
        public static void shutdownOnInterrupt();
        public static void ignoreInterrupt();
        public static void callbackOnInterrupt();
        public static void holdInterrupt();
        public static void releaseInterrupt();

        public static bool interrupted();

        public virtual void interruptCallback(int sig);
    }
}
```

Java

```
package com.zeroc.Ice;

public abstract class Application
{
    // ...

    synchronized public static void destroyOnInterrupt()
    synchronized public static void shutdownOnInterrupt()
    synchronized public static void setInterruptHook(Runnable r)
    synchronized public static void defaultInterrupt()
    synchronized public static boolean interrupted()
}
```

Java Compat

```
package Ice;

public abstract class Application
{
    // ...

    synchronized public static void destroyOnInterrupt()
    synchronized public static void shutdownOnInterrupt()
    synchronized public static void setInterruptHook(Runnable r)
    synchronized public static void defaultInterrupt()
    synchronized public static boolean interrupted()
}
```

Python

```
class Application(object):
    # ...
    def destroyOnInterrupt():
        # ...
    destroyOnInterrupt = classmethod(destroyOnInterrupt)

    def shutdownOnInterrupt():
        # ...
    shutdownOnInterrupt = classmethod(shutdownOnInterrupt)

    def ignoreInterrupt():
        # ...
    ignoreInterrupt = classmethod(ignoreInterrupt)

    def callbackOnInterrupt():
        # ...
    callbackOnInterrupt = classmethod(callbackOnInterrupt)

    def holdInterrupt():
        # ...
    holdInterrupt = classmethod(holdInterrupt)

    def releaseInterrupt():
        # ...
    releaseInterrupt = classmethod(releaseInterrupt)

    def interrupted():
        # ...
    interrupted = classmethod(interrupted)

    def interruptCallback(self, sig):
        # Default implementation does nothing.
        pass
```

Ruby

```
class Application
    def Application.destroyOnInterrupt()

    def Application.ignoreInterrupt()

    def Application.callbackOnInterrupt()

    def Application.holdInterrupt()

    def Application.releaseInterrupt()

    def Application.interrupted()

    def interruptCallback(sig):
        # Default implementation does nothing.
    end
    # ...
end
```

These functions or methods behave as follows:

Java

- `destroyOnInterrupt`
  This method installs a shutdown hook that calls `destroy` on the communicator. This is the default behavior.
- `shutdownOnInterrupt`
  This method installs a shutdown hook that calls `shutdown` on the communicator.
- `setInterruptHook`
  This method installs a custom shutdown hook (a `Runnable`) that takes responsibility for performing whatever action is necessary to terminate the application.
- `defaultInterrupt`
  This method removes the shutdown hook.
- `interrupted`
  This method 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.

Other

- `destroyOnInterrupt`
  This function or method registers a callback that destroys the communicator when one of the monitored signals is raised. This is the default behavior.
- `shutdownOnInterrupt`
  This function or method registers a callback that shuts down the communicator when one of the monitored signals is raised.
- `ignoreInterrupt`
  This function or method causes signals to be ignored.
- `callbackOnInterrupt`
  This function or method configures `Application` to invoke `interruptCallback` when a signal occurs, thereby giving the subclass responsibility for handling the signal.
- `holdInterrupt`
  This function or method causes signals to be held.
- `releaseInterrupt`
  This function or method restores signal delivery to the previous disposition. Any signal that arrives after `holdInterrupt` was called is delivered when you call `releaseInterrupt`.
- `interrupted`
  This function or 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 function or method to respond to signals. The Ice run time may call this function or method concurrently with any other thread. If it raises an exception, the Ice run time prints a warning on `stderr` and ignores the exception.

By default, `Application` behaves as if `destroyOnInterrupt` was invoked, therefore our `main` function requires no change to ensure that the program terminates cleanly on receipt of a signal.

You can disable the signal-handling functionality of `Application` by passing the enumerator `NoSignalHandling` (or `1` in Python) to the constructor. In that case, signals retain their default behavior, that is, terminate the application.

Back to our application, we can add a diagnostic to report the occurrence of a signal:

C++11

```
#include <Ice/Ice.h>

class MyApplication : public Ice::Application
{
public:

    virtual int run(int argc, char* argv[]) override
    {
        // application code goes here...

        if(interrupted())
        {
            cerr << appName() << ": terminating" << endl;
        }
        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

C++98

```
#include <Ice/Ice.h>

class MyApplication : public Ice::Application
{
public:

    virtual int run(int argc, char* argv[])
    {
        // application code goes here...
        if(interrupted())
        {
            cerr << appName() << ": terminating" << endl;
        }
        return 0;
    }
};

int
main(int argc, char* argv[])
{
    MyApplication app;
    return app.main(argc, argv);
}
```

C#

```
using System;

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

            if(interrupted())
            {
                Console.Error.WriteLine(appName() + ": terminating");
            }

            return 0;
        }
    }

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

Java

```
public class Server extends Application
{
    public int run(String[] args)
    {
        // Application code goes 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);
    }
}
```

Java Compat

```
public class Server extends Application
{
    public int run(String[] args)
    {
        // Application code goes 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);
    }
}
```

Python

```
import sys, Ice

class MyApplication(Ice.Application):
    def run(self, args):

        # Application code goes here...

        if self.interrupted():
            print self.appName() + ": terminating"

        return 0

app = MyApplication()
status = app.main(sys.argv)
sys.exit(status)
```

Ruby

```
require 'Ice'

class MyApplication < Ice::Application
    def run(args)
        # Application code goes here...

        if Ice::Application::interrupted()
            print Ice::Application::appName() + ": terminating"
        end

        return 0
    end
end

app = MyApplication.new()
status = app.main(ARGV)
exit(status)
```

Note that, if your server is interrupted by a signal, the Ice run time waits for all currently executing operation dispatches to finish. This means that an operation that updates persistent state cannot be interrupted in the middle of what it was doing and cause partial update problems.

C++ Applications on Linux and macOS

If you handle signals with your own handler (by deriving a subclass from `Ice::Application` and calling `callbackOnInterrupt`), the handler is invoked synchronously from a separate thread. This means that the handler can safely call into the Ice run time or make system calls that are not async-signal-safe without fear of deadlock or data corruption. Note that `Ice::Application` blocks delivery of `SIGINT`, `SIGHUP`, and `SIGTERM`. If your application calls `exec`, this means that the child process will also ignore these signals; if you need the default behavior of these signals in the `exec`'d process, you must explicitly reset them to `SIG_DFL` before calling `exec`.

Java Shutdown Hook and Main Thread

In a subclass of `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 com.zeroc.Ice.Application
{
    public int run(String[] args)
    {
        setInterruptHook(() -> { communicator.destroy(); });

        // ...
    }
}
```

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

Back to Top ^

Limitations of `Application`

`Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use Ice `Application`. Instead, you must use create the `Communicators` directly with `initialize` or through `CommunicatorHolders` in C++, as we saw in the Hello World Application.

Back to Top ^

See Also

- Communicator Initialization
- Command-Line Parsing and Initialization
- CtrlCHandler Helper Class

Previous Next