Deprecated AMI Language Mappings

The deprecated AMI language mappings are described in separate subsections below.

On this page:

Deprecated AMI Mapping for C++

The C++ mapping emits the following code for each AMI operation:

1. An abstract callback class whose name is formed using the pattern `AMI_`*`class`*`_`*`op`*. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Two methods must be defined by the subclass:

   C++

   ```
   void ice_response(<params>);
   void ice_exception(const Ice::Exception &);
   ```

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method returns a boolean indicating whether the request was sent synchronously. The first parameter is a smart pointer to an instance of the callback class described above. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

Slice

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

C++

```
class AMI_I_foo : public ... {
public:
    virtual void ice_response(Ice::Int, Ice::Long) = 0;
    virtual void ice_exception(const Ice::Exception&) = 0;
};
typedef IceUtil::Handle<AMI_I_foo> AMI_I_fooPtr;
```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

C++

```
bool foo_async(const AMI_I_fooPtr&, Ice::Short);
```

The overview describes the proxy methods and callback objects in greater detail.

Deprecated AMI Mapping for Java

The Java mapping emits the following code for each AMI operation:

1. An abstract callback class whose name is formed using the pattern `AMI_`*`class`*`_`*`op`*. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Three methods must be defined by the subclass:

   Java

   ```
   public void ice_response(<params>);
   public void ice_exception(Ice.LocalException ex);
   public void ice_exception(Ice.UserException ex);
   ```

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method returns a boolean indicating whether the request was sent synchronously. The first parameter is a reference to an instance of the callback class described above. The remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

Slice

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

Java

```
public abstract class AMI_I_foo extends ... {
    public abstract void ice_response(int __ret, long l);
    public abstract void ice_exception(Ice.LocalException ex);
    public abstract void ice_exception(Ice.UserException ex);
}
```

The proxy methods for asynchronous invocation of operation `foo` are generated as follows:

Java

```
public boolean foo_async(AMI_I_foo __cb, short s);
public boolean foo_async(AMI_I_foo __cb, short s, java.util.Map<String, String> __ctx);
```

As usual, the version of the operation without a context parameter forwards an empty context to the version with a context parameter.

The overview describes the proxy methods and callback objects in greater detail.

Deprecated AMI Mapping for C#

The C# mapping emits the following code for each AMI operation:

1. An abstract callback class whose name is formed using the pattern `AMI_`*`class`*`_`*`op`*. For example, an operation named `foo` defined in interface `I` results in a class named `AMI_I_foo`. The class is generated in the same scope as the interface or class containing the operation. Two methods must be defined by the subclass:

C#

```
public abstract void ice_response(<params>);
public abstract void ice_exception(Ice.Exception ex);
```

2. An additional proxy method, having the mapped name of the operation with the suffix `_async`. This method returns a boolean indicating whether the request was sent synchronously. The first parameter is a reference to an instance of the callback class described above. The remaining parameters comprise the in parameters of the operation, in the order of declaration.

For example, suppose we have defined the following operation:

Slice

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

The callback class generated for operation `foo` is shown below:

C#

```
public abstract class AMI_I_foo : ...
{
    public abstract void ice_response(int __ret, long l);
    public abstract void ice_exception(Ice.Exception ex);
}
```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

C#

```
bool foo_async(AMI_I_foo __cb, short s);
bool foo_async(AMI_I_foo __cb, short s, Dictionary<string, string> __ctx);
```

As usual, the version of the operation without a context parameter forwards an empty context to the version with a context parameter.

The overview describes the proxy methods and callback objects in greater detail.

Deprecated AMI Mapping for Python

For each AMI operation, the Python mapping emits an additional proxy method having the mapped name of the operation with the suffix `_async`. This method returns a boolean indicating whether the request was sent synchronously. The first parameter is a reference to a callback object; the remaining parameters comprise the `in` parameters of the operation, in the order of declaration.

Unlike the mappings for strongly-typed languages, the Python mapping does not generate a callback class for asynchronous operations. In fact, the callback object's type is irrelevant; the Ice run time simply requires that it define the `ice_response` and `ice_exception` methods:

Python

```
def ice_response(self, <params>)
def ice_exception(self, ex)
```

For example, suppose we have defined the following operation:

Slice

```
interface I {
  ["ami"] int foo(short s, out long l);
};
```

The method signatures required for the callback object of operation `foo` are shown below:

Python

```
class ...
    #
    # Operation signatures:
    #
    # def ice_response(self, _result, l)
    # def ice_exception(self, ex)
```

The proxy method for asynchronous invocation of operation `foo` is generated as follows:

Python

```
def foo_async(self, __cb, s)
```

The overview describes the proxy methods and callback objects in greater detail.

Examples using Deprecated AMI Mapping

To demonstrate the use of AMI in Ice, let's define the Slice interface for a simple computational engine:

Slice

```
module Demo {
    sequence<float> Row;
    sequence<Row> Grid;

    exception RangeError {};

    interface Model {
        ["ami"] Grid interpolate(Grid data, float factor)
            throws RangeError;
    };
};
```

Given a two-dimensional grid of floating point values and a factor, the `interpolate` operation returns a new grid of the same size with the values interpolated in some interesting (but unspecified) way. In the sections below, we present C++, Java, and C# clients that invoke `interpolate` using AMI.

C++ Client Example

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

C++

```
class AMI_Model_interpolateI : public Demo::AMI_Model_interpolate
{
public:
    virtual void ice_response(const Demo::Grid& result)
    {
        cout << "received the grid" << endl;
        // ... postprocessing ...
    }

    virtual void ice_exception(const Ice::Exception& ex)
    {
        try {
            ex.ice_throw();
        } catch (const Demo::RangeError& e) {
            cerr << "interpolate failed: range error" << endl;
        } catch (const Ice::LocalException& e) {
            cerr << "interpolate failed: " << e << endl;
        }
    }
};
```

The implementation of `ice_response` reports a successful result, and `ice_exception` displays a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

C++

```
Demo::ModelPrx model = ...;
AMI_Model_interpolatePtr cb = new AMI_Model_interpolateI;
Demo::Grid grid;
initializeGrid(grid);
model->interpolate_async(cb, grid, 0.5);
```

After obtaining a proxy for a `Model` object, the client instantiates a callback object, initializes a grid and invokes the asynchronous version of `interpolate`. When the Ice run time receives the response to this request, it invokes the callback object supplied by the client.

Java Client Example

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

Java

```
class AMI_Model_interpolateI extends Demo.AMI_Model_interpolate {
    public void ice_response(float[][] result)
    {
        System.out.println("received the grid");
        // ... postprocessing ...
    }

    public void ice_exception(Ice.UserException ex)
    {
        assert(ex instanceof Demo.RangeError);
        System.err.println("interpolate failed: range error");
    }

    public void ice_exception(Ice.LocalException ex)
    {
        System.err.println("interpolate failed: " + ex);
    }
}
```

The implementation of `ice_response` reports a successful result, and the `ice_exception` methods display a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

Java

```
Demo.ModelPrx model = ...;
AMI_Model_interpolate cb = new AMI_Model_interpolateI();
float[][] grid = ...;
initializeGrid(grid);
model.interpolate_async(cb, grid, 0.5);
```

After obtaining a proxy for a `Model` object, the client instantiates a callback object, initializes a grid and invokes the asynchronous version of `interpolate`. When the Ice run time receives the response to this request, it invokes the callback object supplied by the client.

C# Client Example

We must first define our callback implementation class, which derives from the generated class `AMI_Model_interpolate`:

C#

```
using System;

class AMI_Model_interpolateI : Demo.AMI_Model_interpolate {
    public override void ice_response(float[][] result)
    {
        Console.WriteLine("received the grid");
        // ... postprocessing ...
    }

    public override void ice_exception(Ice.Exception ex)
    {
        Console.Error.WriteLine("interpolate failed: " + ex);
    }
}
```

The implementation of `ice_response` reports a successful result, and the `ice_exception` method displays a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

C#

```
Demo.ModelPrx model = ...;
AMI_Model_interpolate cb = new AMI_Model_interpolateI();
float[][] grid = ...;
initializeGrid(grid);
model.interpolate_async(cb, grid, 0.5);
```

Python Client Example

We must first define our callback implementation class:

Python

```
class AMI_Model_interpolateI(object):
    def ice_response(self, result):
        print "received the grid"
        # ... postprocessing ...

  def ice_exception(self, ex):
      try:
          raise ex
      except Demo.RangeError, e:
          print "interpolate failed: range error"
      except Ice.LocalException, e:
          print "interpolate failed: " + str(e)
```

The implementation of `ice_response` reports a successful result, and the `ice_exception` method displays a diagnostic if an exception occurs.

The code to invoke `interpolate` is equally straightforward:

Python

```
model = ...
cb = AMI_Model_interpolateI()
grid = ...
initializeGrid(grid)
model.interpolate_async(cb, grid, 0.5)
```

See Also

- Overview of Deprecated AMI Mapping