C-Sharp Mapping for Operations

On this page:

Basic C# Mapping for Operations

As we saw in the C# mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

C#

```
NodePrx node = ...;             // Initialize proxy
string name = node.name();      // Get name via RPC
```

This illustrates the typical pattern for receiving return values: return values are returned by reference for complex types, and by value for simple types (such as `int` or `double`).

Normal and `idempotent` Operations in C#

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example {
                string op1();
    idempotent  string op2();
};
```

The proxy interface for this is:

C#

```
public interface ExamplePrx : Ice.ObjectPrx
{
    string op1();
    string op2();
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

Passing Parameters in C#

In-Parameters in C#

The parameter passing rules for the C# mapping are very simple: parameters are passed either by value (for value types) or by reference (for reference types). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats — see Location Transparency).

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following proxy for these definitions:

C#

```
public interface ClientToServerPrx : Ice.ObjectPrx
{
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, string[] ss, Dictionary<long, string[]> st);
    void op3(ClientToServerPrx proxy);
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

C#

```
ClientToServerPrx p = ...;              // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
bool b = true;
string s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
string[] ss = new string[1];
ss[0] = "Hello world!";
Dictionary<long, string[]> st = new Dictionary<long, string[]>();
st[0] = ss;
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Out-Parameters in C#

Slice `out` parameters simply map to C# `out` parameters.

Here again are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction:

Slice

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns, out StringSeq ss, out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Slice compiler generates the following code for these definitions:

C#

```
public interface ServerToClientPrx : Ice.ObjectPrx
{
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out string[] ss,
             out Dictionary<long, string[]> st);
    void op3(out ServerToClientPrx proxy);
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

C#

```
ClientToServerPrx p = ...;              // Get proxy...

int i;
float f;
bool b;
string s;
p.op1(out i, out f, out b, out s);

NumberAndString ns;
string[] ss;
Dictionary<long, string[]> st;
p.op2(out ns, out ss, out st);

ServerToClientPrx stc;
p.op3(out stc);

System.Console.WriteLine(i);   // Show one of the values
```

Null Parameters in C#

Some Slice types naturally have "empty" or "not there" semantics. Specifically, C# sequences (if mapped to `CollectionBase`), dictionaries, strings, and structures (if mapped to classes) all can be `null`, but the corresponding Slice types do not have the concept of a null value.

- Slice sequences, dictionaries, and strings cannot be null, but can be empty. To make life with these types easier, whenever you pass a C# n `ull` reference as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.
- If you pass a C# `null` reference to a Slice structure that is mapped to a C# class as a parameter or return value, the Ice run time automatically sends a structure whose elements are default-initialized. This means that all proxy members are initialized to `null`, sequence and dictionary members are initialized to empty collections, strings are initialized to the empty string, and members that have a value type are initialized to their default values.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are structures, sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid `NullReferenceException`. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Exception Handling in C#

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as C# exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

C#

```
ChildPrx child = ...;   // Get child proxy...

try
{
    child.askToCleanUp();
}
catch (Tantrum t)
{
    System.Console.Write("The child says: ");
    System.Console.WriteLine(t.reason);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

C#

```
public class Client
{
    private static void run() {
        ChildPrx child = ...;       // Get child proxy...
        try
        {
            child.askToCleanUp();
        }
        catch (Tantrum t)
        {
            System.Console.Write("The child says: ");
            System.Console.WriteLine(t.reason);
            child.scold();          // Recover from error...
        }
        child.praise();             // Give positive feedback...
    }

    static void Main(string[] args)
    {
        try
        {
            // ...
            run();
            // ...
        }
        catch (Ice.Exception e)
        {
            System.Console.WriteLine(e);
        }
    }
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application.)

Note that the `ToString` method of exceptions prints the name of the exception, any inner exceptions, and the stack trace. Of course, you can be more selective in the way exceptions are displayed. For example, `e.GetType().Name` returns the (unscoped) name of an exception.

Exceptions and Out-Parameters in C#

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [1] but does not provide the strong exception guarantee.

This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

See Also

- Operations
- C-Sharp Mapping for Exceptions
- C-Sharp Mapping for Interfaces
- Location Transparency

References

1. Sutter, H. 1999. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Reading, MA: Addison-Wesley.