C-Sharp Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Proxy Interfaces in C#

On the client side, a Slice interface maps to a C# interface with member functions that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple {
    void op();
};
```

The Slice compiler generates the following definition for use by the client:

C#

```
public interface SimplePrx : Ice.ObjectPrx
{
    void op();
    void op(System.Collections.Generic.Dictionary<string, string> __context);
}
```

As you can see, the compiler generates a *proxy interface* `SimplePrx`. In general, the generated name is `<`*`interface-name`*`>Prx`. If an interface is nested in a module `M`, the generated interface is part of namespace `M`, so the fully-qualified name is `M.<`*`interface-name`*`>Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy instance*. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `Ice.ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy class has a member function of the same name. For the preceding example, we find that the operation o has been mapped to the method `op`. Also note that `op` is overloaded: the second version of `op` has a parameter `__context`, which is a dictionary of string pairs. This parameter is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the `__context` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Because all the `<`*`interface-name`*`>Prx` types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly.The proxy references handed out by the Ice run time are always of type `<`*`interface-name`*`>Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

The `Ice.ObjectPrx` Interface in C#

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `Ice.ObjectPrx`. `ObjectPrx` provides a number of methods:

Copyright © 2017, ZeroC, Inc.

C#

```
namespace Ice
{
    public interface ObjectPrx
    {
        Identity ice_getIdentity();
        bool ice_isA(string id);
        string ice_id();
        void ice_ping();

        int GetHashCode();
        bool Equals(object r);

        // Defined in a helper class:
        //
        public static bool Equals(Ice.ObjectPrx lhs, ObjectPrx rhs);
        public static bool operator==(ObjectPrx lhs, ObjectPrx rhs);
        public static bool operator!=(ObjectPrx lhs, ObjectPrx rhs);

        // ...
    }
}
```

Note that the static methods are not actually defined in `Ice.ObjectPrx`, but in a helper class that becomes a base class of an instantiated proxy. However, this is simply an internal detail of the C# mapping — conceptually, these methods belong with `Ice.ObjectPrx`, so we discuss them here.

The methods behave as follows:

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  Slice

  ```
  module Ice {
      struct Identity {
          string name;
          string category;
      };
  };
  ```

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

  C#

  ```
  Ice.ObjectPrx o1 = ...;
  Ice.ObjectPrx o2 = ...;
  Ice.Identity i1 = o1.ice_getIdentity();
  Ice.Identity i2 = o2.ice_getIdentity();

  if (i1.Equals(i2))
      // o1 and o2 denote the same object
  else
      // o1 and o2 denote different objects
  ```

- **`ice_isA`**
  The `ice_isA` method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID. For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

Copyright © 2017, ZeroC, Inc.

C#

```
Ice.ObjectPrx o = ...;
if (o != null && o.ice_isA("::Printer"))
    // o denotes a Printer object
else
    // o denotes some other type of object
```

Note that we are testing whether the proxy is null before attempting to invoke the `ice_isA` method. This avoids getting a `NullReferenceException` if the proxy is null.

- **`ice_ids`**
  The `ice_ids` method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.
- **`ice_id`**
  The `ice_id` method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type `BasePrx`, with a static type ID of `::Base`, the return value of `ice_id` might be `::Base`, or it might something more derived, such as `::Derived`.
- **`ice_ping`**
  The `ice_ping` method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as `ObjectNotExistException` or `ConnectTimeoutException`.
- **`Equals`**
  This method compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

The `ice_isA`, `ice_ids`, `ice_id`, and `ice_ping` methods are remote operations and therefore support an additional overloading that accepts a request context. Also note that there are other methods in `ObjectPrx`, not shown here. These methods provide different ways to dispatch a call and also provide access to an object's facets.

Proxy Helpers in C#

For each Slice interface, apart from the proxy interface, the Slice-to-C# compiler creates a helper class: for an interface `Simple`, the name of the generated helper class is `SimplePrxHelper`.

> You can ignore the `ObjectPrxHelperBase` base class — it exists for mapping-internal purposes.

The helper class contains two methods of interest:

C#

```
public class SimplePrxHelper : Ice.ObjectPrxHelperBase, SimplePrx
{
    public static SimplePrx checkedCast(Ice.ObjectPrx b);
    public static SimplePrx checkedCast(
        Ice.ObjectPrx b,
        System.Collections.Generic.Dictionary<string, string> ctx);
    public static SimplePrx uncheckedCast(Ice.ObjectPrx b);
    public static string ice_staticId();

    // ...
}
```

For `checkedCast`, if the passed proxy is for an object of type `Simple`, or a proxy for an object with a type derived from `Simple`, the cast returns a non-null reference to a proxy of type `SimplePrx`; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns a null reference.

Copyright © 2017, ZeroC, Inc.

Given a proxy of any type, you can use a `checkedCast` to determine whether the corresponding object supports a given type, for example:

C#

```
Ice.ObjectPrx obj = ...;        // Get a proxy from somewhere...

SimplePrx simple = SimplePrxHelper.checkedCast(obj);
if (simple != null)
    // Object supports the Simple interface...
else
    // Object is not of type Simple...
```

Note that a `checkedCast` contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a `checkedCast` may throw a `ConnectTimeoutException` or an `ObjectNotExistException`. (This also explains the need for the helper class: the Ice run time must contact the server, so we cannot use a C# down-cast.)

In contrast, an `uncheckedCast` does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an `uncheckedCast`, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is `OperationNotExistException`. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process {
    void launch(int stackSize, int dataSize);
};

// ...

interface Rocket {
    void launch(float xCoord, float yCoord);
};
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

C#

```
Ice.ObjectPrx obj = ...;                  // Get proxy...
ProcessPrx process = ProcessPrxHelper.uncheckedCast(obj); // No worries...
process.launch(40, 60);                   // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

A final warning about down-casts: you must use either a `checkedCast` or an `uncheckedCast` to down-cast a proxy. If you use a C# cast, the behavior is undefined.

Another method defined by every helper class is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Using Proxy Methods in C#

Copyright © 2017, ZeroC, Inc.

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second timeout as shown below:

C#

```
Ice.ObjectPrx proxy = communicator.stringToProxy(...);
proxy = proxy.ice_timeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a `checkedCast` or `uncheckedCast` after using a factory method. However, a regular cast is still required, as shown in the example below:

C#

```
Ice.ObjectPrx base = communicator.stringToProxy(...);
HelloPrx hello = HelloPrxHelper.checkedCast(base);
hello = (HelloPrx)hello.ice_timeout(10000); # Type is preserved
hello.sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type.

Object Identity and Proxy Comparison in C#

Proxies provide an `Equals` method that compares proxies:

C#

```
public interface ObjectPrx {
    bool Equals(object r);
}
```

Note that proxy comparison with `Equals` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `Equals` (or `==` and `!=`) tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

C#

```
Ice.ObjectPrx p1 = ...;        // Get a proxy...
Ice.ObjectPrx p2 = ...;        // Get another proxy...

if (!p1.Equals(p2)) {
    // p1 and p2 denote different objects       // WRONG!
} else {
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `Equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `Equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Ice.Util` class:

Copyright © 2017, ZeroC, Inc.

C#

```
public sealed class Util {
    public static int proxyIdentityCompare(ObjectPrx lhs, ObjectPrx rhs);
    public static int proxyIdentityAndFacetCompare(ObjectPrx lhs, ObjectPrx rhs);
// ...
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

C#

```
Ice.ObjectPrx p1 = ...;        // Get a proxy...
Ice.ObjectPrx p2 = ...;        // Get another proxy...

if (Ice.Util.proxyIdentityCompare(p1, p2) != 0) {
    // p1 and p2 denote different objects       // Correct
} else {
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, `-1` if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

The C# mapping also provides two helper classes in the `Ice` namespace that allow you to insert proxies into hashtables or ordered collections, based on the identity, or the identity plus the facet name:

C#

```
public class ProxyIdentityKey
    : System.Collections.IHashCodeProvider,
      System.Collections.IComparer {

    public int GetHashCode(object obj);
    public int Compare(object obj1, object obj2);
}

public class ProxyIdentityFacetKey
    : System.Collections.IHashCodeProvider,
      System.Collections.IComparer {

    public int GetHashCode(object obj);
    public int Compare(object obj1, object obj2);
}
```

Note these classes derive from `IHashCodeProvider` and `IComparer`, so they can be used for both hash tables and ordered collections.

See Also

- Interfaces, Operations, and Exceptions
- Proxies
- C-Sharp Mapping for Operations
- Operations on Object
- Proxy Methods
- Facets and Versioning
- IceStorm

Copyright © 2017, ZeroC, Inc.