Objective-C Mapping for Interfaces

Previous Next

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 represents the remote object. This makes the mapping easy and intuitive to use because, for all intents and purposes (apart from error semantics), making a remote procedure call is no different from making a local procedure call.

On this page:

Proxy Classes and Proxy Protocols in Objective-C

On the client side, interfaces map to a protocol with member functions that correspond to the operations on those interfaces. Consider the following simple interface:

Slice

```
["objc:prefix:EX"]
module Example
{
    interface Simple
    {
        void op();
    }
}
```

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

Objective-C

```
@interface EXSimplePrx : ICEObjectPrx
// Mapping-internal methods here...
@end

@protocol EXSimplePrx <ICEObjectPrx>
-(void) op;
-(void) op:(ICEContext *)context;
@end;
```

As you can see, the compiler generates a proxy protocol `EXSimplePrx` and a proxy class `EXSimplePrx`. In general, the generated name for both protocol and class is <*`module-prefix`*><*`interface-name`*>`Prx`.

In the client's address space, an instance of `EXSimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a *proxy class 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 `EXSimplePrx` derives from `ICEObjectPrx`, and that `EXSimplePrx` adopts the `ICEObjectPrx` protocol. This reflects the fact that all Slice interfaces implicitly derive from `Ice::Object`. For each operation in the interface, the proxy protocol has two methods whose name is derived from the operation. For the preceding example, we find that the operation `op` is mapped to two methods, `op` and `op:`.

The second method has a trailing parameter of type `ICEContext`. This parameter is for use by the Ice run time to store information about how to deliver a request; normally, you do not need to supply a value here and can pretend that the trailing parameter does not exist. (We examine the `ICEContext` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Back to Top ^

Interface Inheritance in Objective-C

Inheritance relationships among Slice interfaces are maintained in the generated Objective-C code. For example:

Slice

```
["objc:prefix:EX"]
module Example
{
    interface A { ... }
    interface B { ... }
    interface C extends A, B { ... }
}
```

The generated code reflects the inheritance hierarchy:

Objective-C

```
@interface EXCPrx : ICEObjectPrx <EXCPrx>
    ...
@end

@protocol EXCPrx <EXAPrx, EXBPrx>
@end
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

Back to Top ^

Proxy Instantiation and Casting in Objective-C

Client-side application code never manipulates proxy class instances directly. In fact, you are not allowed to instantiate a proxy class directly. 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.

Proxies are immutable: once the run time has instantiated a proxy, that proxy continues to denote the same remote object and cannot be changed. `ICEObjectPrx` implements `NSCopying`. However, calling `copy` returns a reference on the target proxy.

Proxies are always passed and returned as type `id<<module-prefix><interface-name>Prx>`. For example, for the preceding `Simple` interface, the proxy type is `id<EXSimplePrx>`.

The `ICEObjectPrx` base class provides class methods that allow you to cast a proxy from one type to another, as described below.

Back to Top ^

Using a Checked Cast in Objective-C

A `checkedCast` tests whether the object denoted by a proxy implements the specified interface:

Objective-C

```
+(id) checkedCast:(id<ICEObjectPrx>)proxy;
```

If so, the cast returns a proxy to the specified interface; otherwise, if the object denoted by the proxy does not implement the specified interface, the cast returns `nil`. Checked casts are typically used to safely down-cast a proxy to a more derived interface. For example, assuming we have Slice interfaces `Base` and `Derived`, you can write the following:

Objective-C

```
id<EXBasePrx> base = ...;  // Initialize base proxy
id<EXDerivedPrx> derived = [EXDerivedPrx checkedCast:base];
if(derived != nil)
{
    // base implements run-time type Derived
    // use derived...
}
else
{
    // Base has some other, unrelated type
}
```

The expression [`EXDerivedPrx checkedCast:base`] tests whether `base` points at an object of type `Derived` (or an object with a type that is derived from `Derived`). If so, the cast succeeds and `derived` is set to point at the same object as `base`. Otherwise, the cast fails and `derived` is set to `nil`. (Proxies that "point nowhere" are represented by `nil`.)

Calling `checkedCast` on a proxy that is already of the desired proxy type returns immediately that proxy. Otherwise, `checkedCast` always calls `ice_isA` on the target object, and upon success, creates a new instance of the desired proxy class.

The message effectively asks the server "is the object denoted by this proxy of type `Derived`?" The reply from the server is communicated to the application code in form of a successful (non-`nil`) or unsuccessful (`nil`) result. Sending a remote message is necessary because, as a rule, there is no way for the client to find out what the actual run-time type of a proxy is without confirmation from the server. (For example, the server may replace the implementation of the object for an existing proxy with a more derived one.) This means that you have to be prepared for a `checkedCast` to fail. For example, if the server is not running, you will receive an `ICEConnectionRefusedException`; if the server is running, but the object denoted by the proxy no longer exists, you will receive an `ICEObjectNotExistException`.

Back to Top ^

Using an Unchecked Cast in Objective-C

In some cases, it is known that an object supports a more derived interface than the static type of its proxy. For such cases, you can use an unchecked down-cast:

Objective-C

```
+(id) uncheckedCast:(id<ICEObjectPrx>)proxy;
```

Here is an example:

Objective-C

```
id<EXBasePrx> base;
base = ...;  // Initialize base to point at a Derived
id<EXDerivedPrx> derived = [EXDerivedPrx uncheckedCast:base];
// Use derived...
```

An `uncheckedCast` provides a down-cast *without* consulting the server as to the actual run-time type of the object. You should use an `uncheckedCast` only if you are certain that the proxy indeed supports the more derived type: an `uncheckedCast`, as the name implies, is not checked in any way; it does not contact the object in the server and, if the proxy does not support the specified interface, the cast does not return null. If you use the proxy resulting from an incorrect `uncheckedCast` to invoke an operation, the behavior is undefined. Most likely, you will receive an `ICEOperationNotExistExceptior`, but, depending on the circumstances, the Ice run time may also report an exception indicating that unmarshaling has failed, or even silently return garbage results.

Despite its dangers, `uncheckedCast` is still useful because it avoids the cost of sending a message to the server. And, particularly during initialization, it is common to receive a proxy of type `id<ICEObjectPrx>`, but with a known run-time type. In such cases, an `uncheckedCast` saves the overhead of sending a remote message.

Note that an `uncheckedCast` is *not* the same as an ordinary cast. The following is incorrect and has undefined behavior:

Objective-C

```
id<EXDerivedPrx> derived = (id<EXDerivedPrx>)base; // Wrong!
```

When not using ARC, both `checkedCast` and `uncheckedCast` call `autorelease` on the proxy they return so, if you want to prevent the proxy from being deallocated once the enclosing autorelease pool is drained, you must call `retain` on the returned proxy.

Back to Top ^

Using Proxy Methods in Objective-C

The `ICEObjectPrx` provides 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 invocation timeout as shown below:

Objective-C

```
id<ICEObjectPrx> proxy = [communicator stringToProxy:...];
proxy = [proxy ice_invocationTimeout:10000];
```

A factory method returns a new (autoreleased) proxy object if the requested modification differs from the current proxy, otherwise it returns the original proxy. The returned proxy is always of the same type as the source proxy, except for 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.

Back to Top ^

Object Identity and Proxy Comparison in Objective-C

Proxy objects support comparison with `isEqual`. Note that `isEqual` 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 as well. In other words, comparison with `isEqual` tests for proxy identity, not object identity. A common mistake is to write code along the following lines:

Objective-C

```
id<ICEObjectPrx> p1 = ...;      // Get a proxy...
id<ICEObjectPrx> p2 = ...;      // Get another proxy...

if(![p1 isEqual: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 if, for example, `p1` and `p2` embed the same object identity, but use a different protocol to contact the target object. Similarly, the protocols might be the same, but could 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 `isEqual`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `isEqual`, 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 additional methods provided by proxies:

Objective-C

```
@protocol ICEObjectPrx <NSObject, NSCopying>
// ...
-(NSComparisonResult) compareIdentity:(id<ICEObjectPrx>)p;
-(NSComparisonResult) compareIdentityAndFacet:(id<ICEObjectPrx>)p;
@end
```

The `compareIdentity` method compares the object identities embedded in two proxies while ignoring other information, such as facet and transport information. To include the facet name in the comparison, use `compareIdentityAndFacet` instead.

`compareIdentity` and `compareIdentityAndFacet` allow you to correctly compare proxies for object identity. The example below demonstrates how to use `compareIdentity`:

Objective-C

```
id<ICEObjectPrx> p1 = ...;      // Get a proxy...
id<ICEObjectPrx> p2 = ...;      // Get another proxy...

if([p1 compareIdentity:p2] != NSOrderedSame)
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Back to Top ^

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Objective-C Mapping for Operations
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm

Previous Next