Objective-C Mapping for Operations

Previous Next

On this page:

Basic Objective-C Mapping for Operations

As we saw in the mapping for interfaces, for each operation on an interface, the proxy protocol contains two corresponding methods with the same name as the operation.

To invoke an operation, you call it via the proxy object. For example, here is part of the definitions for our file system:

Slice

```
["objc:prefix:FS"]
module Filesystem
{
    interface Node
    {
        idempotent string name();
    }
    // ...
}
```

The proxy protocol for the `Node` interface looks as follows:

Objective-C

```
@protocol FSNodePrx <ICEObjectPrx>
-(NSMutableString *) name;
-(NSMutableString *) name:(ICEContext *)context;
@end;
```

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

Objective-C

```
id<EXNodePrx> node = ...;       // Initialize proxy
NSString *name = [node name];   // Get name via RPC
```

The `name` method sends the operation invocation to the server, waits until the operation is complete, and then unmarshals the return value and returns it to the caller.

It is safe to ignore the return value even when not using ARC as the returned value is autoreleased. For example, the following code contains no memory leak:

Objective-C

```
id<EXNodePrx> node = ...;       // Initialize proxy
[node name];                    // Useless, but no leak
```

If you ignore the return value, no memory leak occurs because the next time the enclosing autorelease pool is drained, the memory will be reclaimed.

Back to Top ^

Normal and `idempotent` Operations in Objective-C

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

Slice

```
interface Ops
{
                string op1();
    idempotent  string op2();
    idempotent  void op3(string s);
}
```

The proxy protocol for this interface looks like this:

Objective-C

```
@protocol EXOpsPrx <ICEObjectPrx>
-(NSMutableString *) op1;
-(NSMutableString *) op1:(ICEContext *)context;
-(NSMutableString *) op2;
-(NSMutableString *) op2:(ICEContext *)context;
-(void) op3:(NSString *)s;
-(void) op3:(NSString *)s context:(ICEContext *)context;
@end;
```

For brevity, we will not show the methods with the additional trailing `context` parameter for the remainder of this discussion. Of course, the compiler generates the additional methods regardless.

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the mapping to be unaffected by the `idempotent` keyword.

Back to Top ^

Passing Parameters in Objective-C

In-Parameters in Objective-C

The parameter passing rules for the Objective-C mapping are very simple: value type parameters are passed by value and non-value type parameters are passed by pointer. Semantically, the two ways of passing parameters are identical: the Ice run time guarantees not to change the value of an in-parameter.

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 code for this definition:

Objective-C

```
@interface EXNumberAndString : NSObject <NSCopying>
// ...
@property(nonatomic, assign) ICEInt x;
@property(nonatomic, strong) NSString *str;
// ...
@end

typedef NSArray EXStringSeq;
typedef NSMutableArray EXMutableStringSeq;

typedef NSDictionary EXStringTable;
typedef NSMutableDictionary EXMutableStringTable;

@protocol EXClientToServerPrx <ICEObjectPrx>
-(void) op1:(ICEInt)i f:(ICEFloat)f b:(BOOL)b s:(NSString *)s;
-(void) op2:(EXNumberAndString *)ns ss:(EXStringSeq *)ss st:(NSDictionary *)st;
-(void) op3:(id<EXClientToServerPrx>)proxy;
@end;
```

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

Objective-C

```
id<EXClientToServerPrx> p = ...;          // Get proxy...

[p op1:42 f:3.14 b:YES s:@"Hello world!"]; // Pass literals

ICEInt i = 42;
ICEFloat f = 3.14;
BOOL b = YES;
NSString *s = @"Hello world!";

[p op1:i f:f b:b s:s];                    // Pass simple vars

EXNumberAndString *ns = [EXNumberAndString numberAndString:42 str:@"The Answer"];
EXMutableStringSeq *ss = [ExMutableStringSeq array];
[ss addObject:@"Hello world!"];
EXStringTable *st = [EXMutableStringTable dictionary];
[ss setObject:ss forKey:[NSNumber numberWithInt:0]];

[p op2:ns ss:ss st:st];                   // Pass complex vars

[p op3:p];                                // Pass proxy
```

You can pass either literals or variables to the various operations. The Ice run time simply marshals the value of the parameters to the server and leaves parameters otherwise untouched, so there are no memory-management issues to consider.

Note that the invocation of `op3` is somewhat unusual: the caller passes the proxy it uses to invoke the operation to the operation as a parameter. While unusual, this is legal (and no memory management issues arise from doing this.)

Back to Top ^

Passing `nil` and `NSNull` in Objective-C

The Slice language supports the concept of null ("points nowhere") for only two of its types: proxies and classes. For either type, `nil` represents a null proxy or class. For other Slice types, such as strings, the concept of a null string simply does not apply. (There is no such thing as a null string, only the empty string.) However, strings, structures, sequences, and dictionaries are all passed by pointer, which raises the question of how the Objective-C mapping deals with `nil` values.

As a convenience feature, the Objective-C mapping permits passing of `nil` as a parameter for the following types:

- Proxies (`nil` sends a null proxy.)
- Classes (`nil` sends a null class instance.)
- Strings (`nil` sends an empty string.)
- Structures (`nil` sends a default-initialized structure.)
- Sequences (`nil` sends an empty sequence.)
- Dictionaries (`nil` sends an empty dictionary.)

It is impossible to add `nil` to an `NSArray` or `NSDictionary`, so the mapping follows the usual convention that an `NSArray` element or `NSDictionary` value that is conceptually `nil` is represented by `NSNull`. For example, to send a sequence of proxies, some of which are null proxies, you must insert `NSNull` values into the sequence.

As a convenience feature, if you have a sequence with elements of type string, structure, sequence, or dictionary, you can use `NSNull` as the element value. For elements that are `NSNull`, the Ice run time marshals an empty string, default-initialized structure, empty sequence, or empty dictionary to the receiver.

Similarly, for dictionaries with value type string, structure, sequence, or dictionary, you can use `NSNull` as the value to send the corresponding empty value (or default-initialized value, in the case of structures).

Back to Top ^

Out-Parameters in Objective-C

The Objective-C mapping passes out-parameters by pointer (for value types) and by pointer-to-pointer (for non-value types). Here is the Slice definition once more, modified to pass all parameters 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 ClientToServer* proxy);
}
```

The Slice compiler generates the following code for this definition:

Objective-C

```
@protocol EXServerToClientPrx <ICEObjectPrx>
-(void) op1:(ICEInt *)i f:(ICEFloat *)f b:(BOOL *)b s:(NSMutableString **)s;
-(void) op2:(EXNumberAndString **)ns ss:(EXMutableStringSeq **)ss st:(EXMutableStringTable **)st;
-(void) op3:(id<EXClientToServerPrx> *)proxy;
@end
```

Note that, for types that come in immutable and mutable variants (strings, sequences, and dictionaries), the corresponding out-parameter uses the mutable variant.

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

Objective-C

```
id<EXServerToClientPrx> p = ...; // Get proxy...

ICEInt i;
ICEFloat f;
BOOL b;
NSMutableString *s;

[p op1:&i f:&f b:&b s:&s];
// i, f, b, and s contain updated values now

EXNumberAndString *ns;
EXStringSeq *ss;
EXStringTable *st;

[p op2:&ns ss:&ss st:&st];
// ns, ss, and st contain updated values now

[p op3:&p];
// p has changed now!
```

Again, there are no surprises in this code: the caller simply passes pointers to pointer variables to a method; once the operation completes, the values of those variables will have been set by the server.

Back to Top ^

Memory Management for Out-Parameters in Objective-C

When the Ice run time returns an out-parameter to the caller, it does not make any assumptions about the previous value of that parameter (if any). In other words, if you pass an initialized string as an out-parameter, the value you pass is simply discarded and the corresponding variable is assigned a new instance. As an example, consider the following operation:

Slice

```
void getString(out string s);
```

You could call this as follows:

Objective-C

```
NSMutableString *s = @"Hello";
[p getString:&s];
// s now points at the returned string.
```

When not using ARC, all out-parameters are autoreleased by the Ice run time before they are returned. With ARC, out parameters are implicitly qualified with `__autoreleasing` so the returned objects are automatically autoreleased. This is convenient because it does just the right thing with respect to memory management. For example, the following code does not leak memory:

Objective-C

```
NSMutableString *s = @"Hello";
[p getString:&s];
[p getString:&s]; // No leak here.
```

Beware however that when not using ARC, because the pointer value of out-parameters is simply assigned by the proxy method, you must be careful not to pass a variable as an out-parameter if that variable was not released or autoreleased:

Objective-C

```
NSMutableString *s = [[NSMutableString alloc] initWithString:@"Hello"];
[p getString:&s]; // Bad news when not using ARC!
```

This code leaks the initial string because the proxy method assigns the passed pointer without calling `release` on it first. (In practice, this is rarely a problem because there is no need to initialize out-parameters and, if an out-parameter was initialized by being passed as an out-parameter to an operation earlier, its value will have been autoreleased by the proxy method already.) When using ARC, this isn't an issue, the compiler will automatically release the string when the out-parameter is returned.

It is worth having another look at the final call of the code example we saw earlier:

Objective-C

```
[p op3:&p];
```

Here, p is the proxy that is used to dispatch the call. That same variable p is also passed as an out-parameter to the call, meaning that the server will set its value. In general, passing the same parameter as both an input and output parameter is safe (with the caveat we just discussed when not using ARC).

Back to Top ^

Receiving Return Values in Objective-C

The Objective-C mapping returns return values in much the same way as out-parameters: value types are returned by value, and non-value types are returned by pointer. As an example, consider the following operations:

Slice

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

interface Ops
{
    int getInt();
    string getString();
    NumberAndString getNumberAndString();
}
```

The proxy protocol looks as follows:

Objective-C

```
@protocol EXOpsPrx <ICEObjectPrx>
-(ICEInt) getInt;
-(NSMutableString *) getString;
-(EXNumberAndString *) getNumberAndString;
@end
```

Note that, for types with mutable and immutable variants (strings, sequences, and dictionaries), the formal return type is the mutable variant. As for out-parameters, when not using ARC, anything returned by pointer is autoreleased by the Ice run time. This means that the following code works fine and does not contain memory management errors whether or not you use ARC:

Objective-C

```
EXNumberAndString *ns = [NSNumberAndString numberAndString];
ns.x = [p getInt];
ns.str = [p getString]; // Autoreleased by getString and retained by ns.str when not using ARC
[p getNumberAndString]; // No leak here.
```

The return value of `getString` is autoreleased by the proxy method but, during the assignment to the property `str`, the generated code calls `retain`, so the structure keeps the returned string alive in memory, as it should. Similarly, ignoring the return value from an invocation is safe because the returned value is autoreleased and will be reclaimed when the enclosing autorelease pool is drained.

Back to Top ^

Chained Invocations in Objective-C

Consider the following simple interface containing two operations, one to set a value and one to get it:

Slice

```
interface Name
{
    string getName();
    void setName(string name);
}
```

Suppose we have two proxies to interfaces of type `Name`, `p1` and `p2`, and chain invocations as follows:

Objective-C

```
[p2 setName:[p1 getName]]; // No leak here.
```

This works exactly as intended: the value returned by `p1` is transferred to `p2`. There are no memory-management or exception safety issues with this code.

Back to Top ^

`nil` Out-Parameters and Return Values in Objective-C

If an out-parameter or return value is a proxy or class, and the operation returns a null proxy or class, the proxy method returns `nil`. If a proxy or class is returned as part of a sequence or dictionary, the corresponding entry is `NSNull`.

For strings, structures, sequences, and dictionaries, the Ice run time *never* returns `nil` or `NSNull` (even if the server passed `nil` or `NSNull` as the value). Instead, the unmarshaling code always instantiates an empty string, empty sequence, or empty dictionary, and it always initializes structure members during unmarshaling, so structures that are returned from an operation invocation never contain a `nil` instance variable (except for proxy and class instance variables).

Back to Top ^

Optional Parameters in Objective-C

The Objective-C mapping uses the `id` type for optional parameters. As a result, there's no compile time check for optional parameters. Instead, Ice performs a run-time type check and if the optional parameter does not match the expected type an `NSException` with the `NSInvalidArgumentExcepti` on name is raised. Slice types that map to an Objective-C class use the same mapping as required parameters. Slice built-in basic types (except string) are boxed into an `NSNumber` value. The `ICENone` singleton value can also be passed as the value of an optional parameter or return value.

Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string aString, optional(3) int anInt, out optional(4) float outFloat);
```

A client can invoke this operation as shown below:

Objective-C

```
id f;
id i;

i = [proxy execute:@"--file log.txt" anInt:@14 outFloat:&f]
i = [proxy execute:ICENone anInt:@14 outFloat:&f] // aString is unset

if(i == ICENone)
{
    NSLog(@"return value is not set");
}
else
{
    int v = [i intValue];
    NSLog(@"return value is set to %d", v);
}
```

Passing `nil` for an optional parameter is the same as passing `nil` for a required parameter, the optional parameter is considered to be set to `nil`. For Slice built-in basic types (except string), the optional parameter is considered to be set to `0` or `NO` for booleans.

A well-behaved program must not assume that an optional parameter always has a value. It should compare the value to `ICENone` to determine whether the optional parameter is set.

Back to Top ^

Exception Handling in Objective-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 Objective-C exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

Objective-C

```
id<EXChildPrx> child = ...;    // Get proxy...
@try
{
    [child askToCleanUp];      // Give it a try...
}
@catch(EXTantrum *t)
{
    printf("The child says: %s\n", 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 dealt with by exception handlers higher in the hierarchy. For example:

Objective-C

```
void run()
{
    id<EXChildPrx> child = ...;       // Get proxy...
    @try
    {
        [child askToCleanUp];         // Give it a try...
    }
    @catch(EXTantrum *t)
    {
        printf("The child says: %s\n", t.reason);
        [child scold];                // Recover from error...
    }
    [child praise];                   // Give positive feedback...
}

int
main(int argc, char* argv[])
{
    int status = 1;
    @try
    {
        // ...
        run();
        // ...
        status = 0;
    }
    @catch(ICEException *e)
    {
        printf("Unexpected run-time error: %s\n", [e ice_name]);
    }
    // ...
    return status;
}
```

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.)

Back to Top ^

Exceptions and Out-Parameters in Objective-C

If an operation throws an exception, the Ice run time makes no guarantee for the value of out-parameters. Individual out-parameters may have the old value, the new value, or a value that is indeterminate, such that parts of the out-parameter have been assigned and others have not. However, no matter what their state, the values will be "safe" for memory-management purposes, that is, any out-parameters that were successfully unmarshaled are autoreleased.

Back to Top ^

Exceptions and Return Values in Objective-C

For return values, the Objective-C mapping provides the guarantee that a variable receiving the return value of an operation will not be overwritten if an exception is thrown.

Back to Top ^

See Also

- Operations
- Hello World Application
- Objective-C Mapping for Interfaces
- Objective-C Mapping for Exceptions
- Request Contexts

Previous Next