Objective-C Mapping for Classes

On this page:

Basic Objective-C Mapping for Classes

A Slice class is mapped similar to a structure and exception.

The generated class contains an instance variable and a property for each Slice data member. Consider the following class definition:

Slice

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler generates the following code for this definition:

Objective-C

```
@interface EXTimeOfDay : ICEObject
{
    ICEShort hour;
    ICEShort minute;
    ICEShort second;
}

@property(nonatomic, assign) ICEShort hour;
@property(nonatomic, assign) ICEShort minute;
@property(nonatomic, assign) ICEShort second;

-(id) init:(ICEShort)hour minute:(ICEShort)minute second:(ICEShort)second;
+(id) timeOfDay;
+(id) timeOfDay:(ICEShort)hour minute:(ICEShort)minute second:(ICEShort)second;
@end
```

There are a number of things to note about the generated code:

1. The generated class `EXTimeOfDay` derives from `ICEObject`, which is the parent of all classes. Note that `ICEObject` is *not* the same as `ICEObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The generated class contains a property for each Slice data member.
3. The generated class provides an `init` method that accepts one argument for each data member, and it provides the same two convenience constructors as structures and exceptions.

Derivation from `ICEObject` in Objective-C

All classes ultimately derive from a common base class, `ICEObject`. Note that this is not the same as implementing the `ICEObjectPrx` protocol (which is implemented by proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`ICEObject` defines a number of methods:

Objective-C

```
@protocol ICEObject <NSObject>
-(BOOL) ice_isA:(NSString*)typeId current:(ICECurrent*)current;
-(void) ice_ping:(ICECurrent*)current;
-(NSString*) ice_id:(ICECurrent*)current;
-(NSArray*) ice_ids:(ICECurrent*)current;
@end

@interface ICEObject NSObject <ICEObject, NSCopying>
-(BOOL) ice_isA:(NSString*)typeId;
-(void) ice_ping;
-(NSString*) ice_id;
-(NSArray*) ice_ids;
+(NSString*) ice_staticId;
-(void) ice_preMarshal;
-(void) ice_postUnmarshal;
-(BOOL) ice_dispatch:(id<ICERequest>)request;
-(id) initWithDelegate:(id)delegate;
+(id) objectWithDelegate:(id)delegate;
@end
```

The methods are split between the `ICEObject` protocol and class because classes can be servants.

The methods of `ICEObject` behave as follows:

- `ice_isA`
  This function returns YES if the object supports the given type ID, and `NO` otherwise.
- `ice_ping`
  `ice_ping` provides a basic reachability test for the class. If it completes without raising an exception, the class exists and is reachable. Note that `ice_ping` is normally only invoked on the proxy for a class that might be remote because a class instance that is local (in the caller's address space) can always be reached.
- `ice_ids`
  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`
  This function returns the actual run-time type ID for a class. If you call `ice_id` via a pointer to a base instance, the returned type ID is the actual (possibly more derived) type ID of the instance.
- `ice_staticId`
  This function returns the static type ID of a class.
- `ice_preMarshal`
  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_dispatch`
  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.
- `initWithDelegate`
  These constructors enable the implementation of servants with a delegate.

Class Data Members in Objective-C

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding property.

Class Constructors in Objective-C

Classes provide the usual `init` method and a parameter-less convenience constructor that perform default initialization of the class's instance variables. If you declare default values in your Slice definition, the `init` method and convenience constructor initialize each data member with its declared value.

In addition, if a class has data members, it provides an `init` method and a convenience constructor that accept one argument for each data member. This allows you to allocate and initialize a class instance in a single statement (instead of first having to allocate and default-initialize the instance and then assign to its properties).

For derived classes, the `init` method and the convenience constructor have one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order. For example:

Slice

```
class Base {
    int i;
};

class Derived extends Base {
    string s;
};
```

This generates:

Objective-C

```
@interface EXBase : ICEObject
// ...

@property(nonatomic, assign) ICEInt i;

-(id) init:(ICEInt)i;
+(id) base;
+(id) base:(ICEInt)i;
@end

@interface EXDerived : EXBase
// ...

@property(nonatomic, retain) NSString *s;

-(id) init:(ICEInt)i s:(NSString *)s;
+(id) derived;
+(id) derived:(ICEInt)i s:(NSString *)s;
@end
```

Derived Classes in Objective-C

Note that, in the preceding example, the derivation of the Slice definitions is preserved for the generated classes: `EXBase` derives from `ICEObject`, and `EXDerived` derives from `EXBase`. This allows you to treat and pass classes polymorphically: you can always pass an `EXDerived` instance where an `EXBase` instance is expected.

Passing Classes as Parameters in Objective-C

Classes are passed by pointer, like any other Objective-C object. For example, here is an operation that accepts a `Base` as an in-parameter and returns a `Derived`:

Slice

```
Derived getDerived(Base d);
```

The corresponding proxy method looks as follows:

Objective-C

```
-(EXDerived *) getDerived:(EXBase *)d;
```

To pass a null instance, you simply pass `nil`.

Operations of Classes in Objective-C

If you look back at the code that is generated for the `EXTimeOfDay` class, you will notice that there is no indication at all that the class has a `format` operation. As opposed to proxies, classes do not implement any protocol that would define which operations are available. This means that you can partially implement the operations of a class. For example, you might have a Slice class with five operations that is returned from a server to a client. If the client uses only one of the five operations, the client-side code needs to implement only that one operation and can leave the remaining four operations without implementation. (If the class were to implement a mandatory protocol, the client-side code would have to implement all operations in order to avoid a compiler warning.)

Of course, you must implement those operations that you actually intend to call. The mapping of operations for classes follows the *server-side* mapping for operations on interfaces: parameter types and labels are exactly the same. (See Parameter Passing in Objective-C for details.) In a nutshell, the server-side mapping is the same as the client-side mapping except that, for types that have mutable and immutable variants, they map to the immutable variant where the client-side mapping uses the mutable variant, and vice versa.

For example, here is how we could implement the `format` operation of our `TimeOfDay` class:

Objective-C

```
@interface TimeOfDayI : EXTimeOfDay
@end

@implementation TimeOfDayI
-(NSString *) format
{
    return [NSString stringWithFormat:@"%.2d:%.2d:%.2d", self.hour, self.minute, self.second];
}
@end
```

By convention, the implementation of classes with operations has the same name as the Slice class with an `I`-suffix. Doing this is not mandatory — you can call your implementation class anything you like. However, if you do not want to use the `I`-suffix naming, we recommend that you adopt another naming convention and follow it consistently.

Note that `TimeOfDayI` derives from `EXTimeOfDay`. This is because, as we will see in a moment, the Ice run time will instantiate a `TimeOfDayI` instance whenever it receives a `TimeOfDay` instance over the wire and expects that instance to provide the properties of `EXTimeOfDay`.

Class Factories in Objective-C

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

Slice

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDayI` class. However, unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements a `format` method. To allow the Ice run time to instantiate the correct object, we must provide a factory that knows that the Slice `TimeOfDay` class is implemented by our `TimeOfDayI` class. The `Ice::Communicator` interface provides us with the necessary operations:

Slice

```
["objc:prefix:ICE"]
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

Slice

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };
};
```

The object factory's `create` operation is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` operation is called by the Ice run time when its communicator is destroyed. A possible implementation of our object factory is:

Objective-C

```
@interface ObjectFactory<ICEObjectFactory>
@end

@implementation ObjectFactory
-(ICEObject*) create:(NSString *)type
{
    NSAssert([type isEqualToString:@"::Example::TimeOfDay"]);
    return [[TimeOfDayI alloc] init];
}
@end
```

The `create` method is passed the type ID of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::Example::TimeOfDay"`. Our implementation of `create` checks the type ID: if it is `"::Example::TimeOfDay"`, it instantiates and returns a `TimeOfDayI` object. For other type IDs, it asserts because it does not know how to instantiate other types of objects.

Note that your factory *must not* autorelease the returned instance. The Ice run time takes care of the necessary memory management activities on your behalf.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

Objective-C

```
id<ICECommunicator> ice = ...;
ObjectFactory *factory = [[[ObjectFactory alloc] init] autorelease];
[ic addObjectFactory:factory sliceId:@"::Example::TimeOfDay"];
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::Example::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator — if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, the Ice run time may make concurrent calls to `create`.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not (but can) create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

Using a Category to Implement Operations in Objective-C

An alternative to registering a class factory is to use an Objective-C category to implement operations. For example, we could have implemented our `format` method using a category instead:

Objective-C

```
@interface EXTimeOfDay (TimeOfDayI)
@end

@implementation EXTimeOfDay (TimeOfDayI)
-(NSString *) format
{
    return [NSString stringWithFormat:@"%.2d:%.2d:%.2d", self.hour, self.minute, self.second];
}
@end
```

In this case, there is no need to derive from the generated `EXTimeOfDay` class because we provide the format implementation as a category. There is also no need to register a class factory: the Ice run time instantiates an `EXTimeOfDay` instance when a `TimeOfDay` instance arrives over the wire, and the `format` method is found at run time when it is actually called.

This is a viable alternative approach to implement class operations. However, keep in mind that, if the operation implementation requires use of instance variables that are not defined as part of the Slice definitions of a class, you cannot use this approach because Objective-C categories do not permit you to add instance variables to a class.

Copying of Classes in Objective-C

Classes implement `NSCopying`. The behavior is the same as for structures: instance variables of value type are copied by assignment, instance variables of pointer type are copied by calling `retain`, that is, the copy is shallow. To illustrate this, consider the following class definition:

Slice

```
class Node {
    int i;
    string s;
    Node next;
};
```

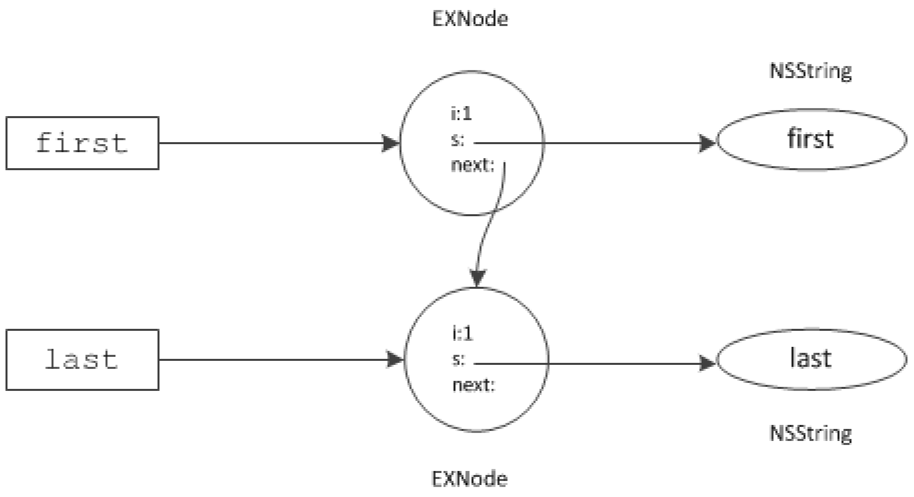
We can initialize two instances of type `EXNode` as follows:

Objective-C

```
NSString lastString = [NSString stringWithString:@"last"];
EXNode *last = [EXNode node:99 s:lastString next:nil];

NSString firstString = [NSString stringWithString:@"first"];
EXNode *first = [EXNode node:1 s:firstString next:last];
```

This creates the situation shown below:



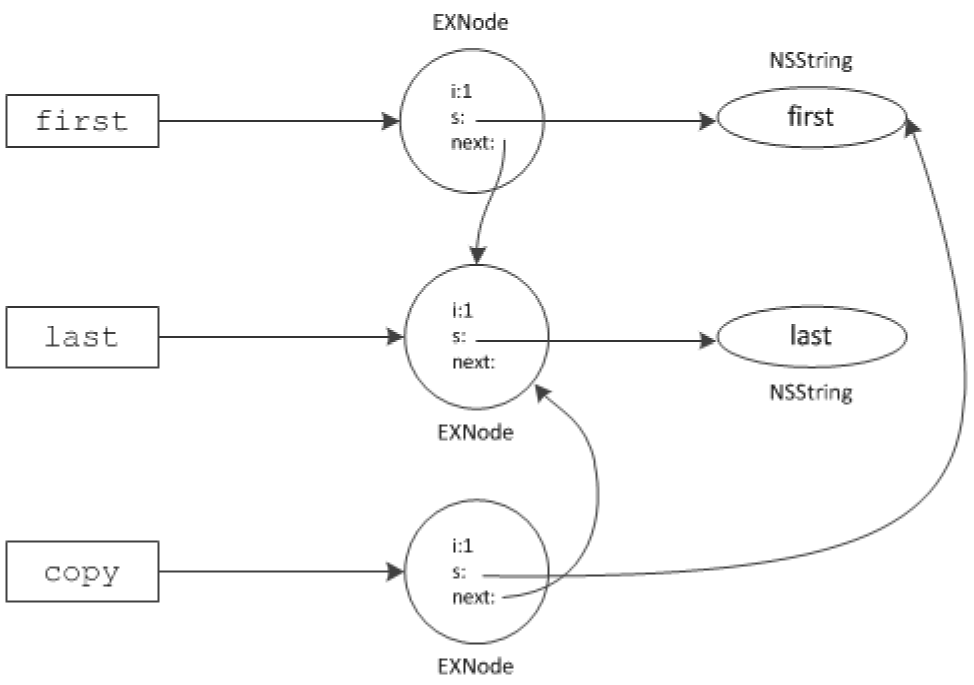
Two instances of type EXNode.

Now we create a copy of the first node by calling `copy`:

Objective-C

```
EXNode *copy = [[first copy] autorelease];
```

This creates the situation shown here:



`EXNode` instances after calling `copy` on `first`.

As you can see, the first node is copied, but the last node (pointed at by the `next` instance variable of the first node) is not copied; instead, `first` and `copy` now both have their `next` instance variable point at the same last node, and both point at the same string.

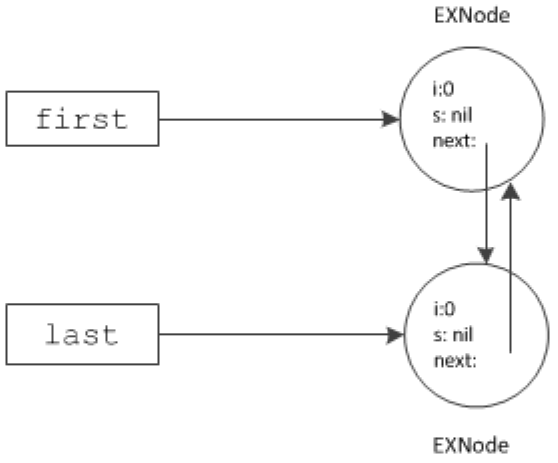
Cyclic References in Objective-C

One thing to be aware of are cyclic references among classes. As an example, we can easily create a cycle by executing the following statements:

Objective-C

```
EXNode *first = [EXNode node];
ExNode *last = [EXNode node];
first.next = last;
last.next = first;
```

This makes the `next` instance variable of the two classes point at each other, creating the cycle shown below:



Two nodes with cyclic references.

There is no problem with sending this class graph as a parameter. For example, you could pass either `first` or `last` as a parameter to an operation and, in the server, the Ice run time will faithfully rebuild the corresponding graph, preserving the cycle. However, if a server returns such a graph from an operation invocation as the return value or as an out-parameter, all class instances that are part of a cycle are leaked. The same is true on the client side: if you receive such a graph from an operation invocation and do not explicitly break the cycle, you will leak all instances that form part of the cycle.

Because it is difficult to break cycles manually (and, on the server side, for return values and out-parameters, it is impossible to break them), we recommend that you avoid cyclic references among classes.

A future version of the Objective-C run time may provide a garbage collector similar to the one used by Ice for C++.

See Also

- Simple Classes
- Objective-C Mapping for Classes]
- Server-Side Objective-C Mapping for Interfaces
- Parameter Passing in Objective-C
- Dispatch Interceptors