Objective-C Mapping for Exceptions

This page describes the Objective-C mapping for exceptions.

On this page:

Exception Inheritance Hierarchy in Objective-C

Here again is a fragment of the Slice definition for our world time server:

Slice

```
exception GenericError {
    string reason;
};
exception BadTimeVal extends GenericError {};
exception BadZoneName extends GenericError {};
```

These exception definitions map as follows:

Objective-C

```
@interface EXGenericError : ICEUserException
{
@private
    NSString *reason_;
}

@property(nonatomic, retain) NSString *reason_;

// ...
@end

@interface EXBadTimeVal : EXGenericError
// ...
@end

@interface EXBadZoneName : EXGenericError
// ...
@end
```

Each Slice exception is mapped to an Objective-C class. For each exception member, the corresponding class contains a private instance variable and a property. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.)

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `EXBadTimeVal` and `EXBadZoneName` inherit from `EXGenericError`.

In turn, `EXGenericError` derives from `ICEUserException`:

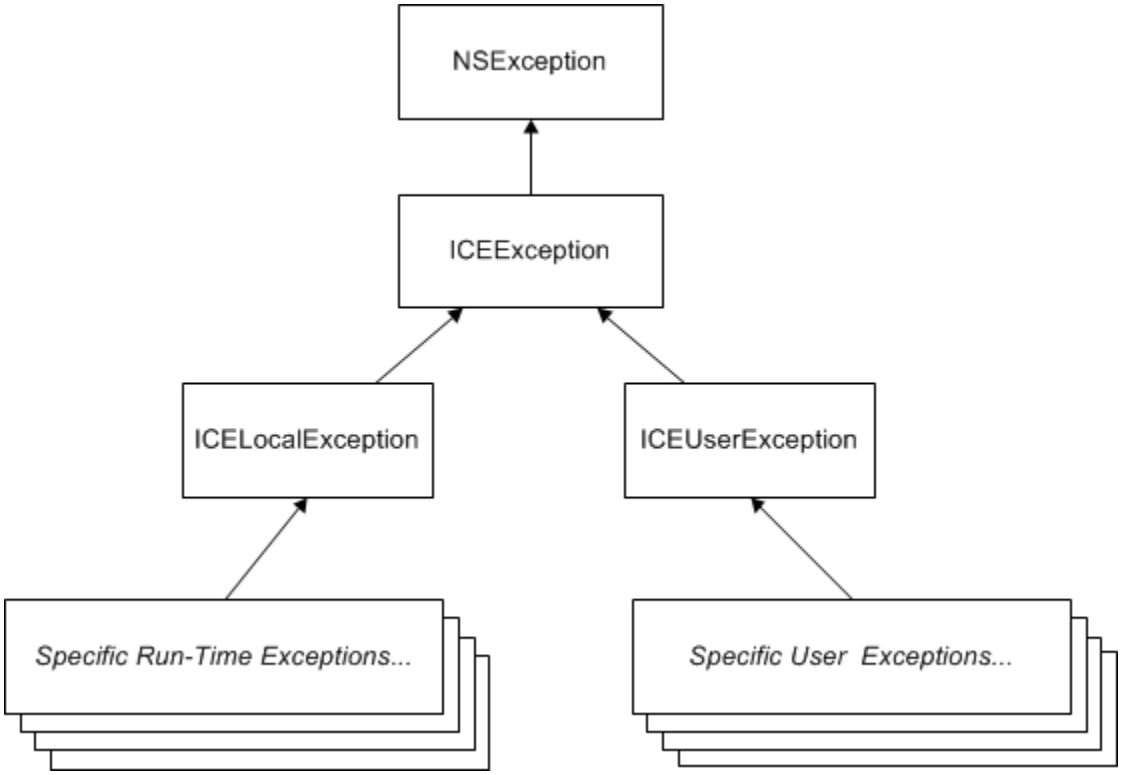
Objective-C

```
@interface ICEException : NSException
-(NSString* *)ice_name;
@end

@interface ICEUserException : ICEException
// ...
@end

@interface ICELocalException : ICEException
// ...
@end
```

Note that `ICEUserException` itself derives from `ICEException`, which derives from `NSException`. Similarly, run-time exceptions derive from a common base class `ICELocalException` that derives from `ICEException`, so we have the inheritance structure shown below:



Inheritance structure for exceptions.

`ICEException` provides a single method, `ice_name`, that returns the Slice type ID of the exception with the leading `::` omitted. For example, the return value of `ice_name` for our Slice `GenericError` is `Example::GenericError`.

Mapping for Exception Data Members in Objective-C

As we mentioned earlier, each data member of a Slice exception generates a corresponding Objective-C property. Here is an example that extends our `GenericError` with yet another exception:

Slice

```
exception GenericError {
    string reason;
};

exception FileError extends GenericError {
    string name;
    int errorCode;
};
```

The generated properties for these exceptions are as follows:

Objective-C

```
@interface EXGenericError : ICEUserException
{
@private
    NSString *reason_;
}

@property(nonatomic, retain) NSString *reason_;

// ...
@end

@interface EXFileError : EXGenericError
{
@private
    NSString *name_;
    ICEInt errorCode;
}

@property(nonatomic, retain) NSString *name_;
@property(nonatomic, assign) ICEInt errorCode;

// ...
@end
```

This is exactly the same mapping as for structure members, with one difference: the `name` and `reason` members map to `name_` and `reason_` properties, whereas — as for structures — `errorCode` maps to `errorCode`. The trailing underscore for `reason_` and `name_` prevents a name collision with the `name` and `reason` methods that are defined by `NSException`: if you call the `name` method, you receive the name that is stored by `NSException`; if you call the `name_` method, you receive the value of the `name_` instance variable of `EXFileError`:

Objective-C

```
@try {
    // Do something that can throw ExFileError...
}
@catch(EXFileError *ex)
{
    // Print the value of the Slice reason, name,
    // and errorCode members.
    printf("reason: %s, name: %s, errorCode: %d\n",
            [ex.reason_ UTF8String],
            [ex.name_ UTF8String],
            ex.errorCode);

    // Print the NSException name.
    printf("NSException name: %s\n", [[ex name] UTF8String]);
}
```

The same escape mechanism applies if you define exception data members named `callStackReturnAddresses`, `raise`, or `userInfo`.

Objective-C Mapping for User Exceptions

Initialization of exceptions follows the same pattern as for structures: each exception (apart from the inherited no-argument `init` method) provides an `init` method that accepts one argument for each data member of the exception, and two convenience constructors. For example, the generated methods for our `EXGenericError` exception look as follows:

Objective-C

```
@interface EXGenericError : ICEUserException
// ...

-(id) init:(NSString *)reason;
+(id) genericError;
+(id) genericError:(NSString *)reason;
@endif
```

If a user exception has no data members (and its base exceptions do not have data members either), only the inherited `init` method and the no-argument convenience constructor are generated.

If you declare default values in your Slice definition, the inherited `init` method and the no-argument convenience constructor initialize each data member with its declared value.

If an exception has a base exception with data members, its `init` method and convenience constructor accept one argument for each Slice data member, in base-to-derived order. For example, here are the methods for the `FileError` exception we defined above:

Objective-C

```
@interface EXFileError : EXGenericError
// ...

-(id) init:(NSString *)reason name_:(NSString *)name
                           errorCode:(ICEInt)errorCode;
+(id) fileError;
+(id) fileError:(NSString *)reason name_:(NSString *)name
                                 errorCode:(ICEInt)errorCode;
@end
```

Note that `init` and the second convenience constructor accept three arguments; the first initializes the `EXGenericError` base, and the remaining two initialize the instance variables of `EXFileError`.

Objective-C Mapping for Run-Time Exceptions

The Ice run time throws run-time exceptions for a number of pre-defined error conditions. All run-time exceptions directly or indirectly derive from `ICELocalException` which, in turn, derives from `ICEException`. (See the above illustration for an example of an inheritance diagram.)

By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `NSException`
  This is the root of the complete inheritance tree. Catching `NSException` catches all exceptions, whether they relate to Ice or the Cocoa framework.
- `ICEException`
  Catching `ICEException` catches both user and run-time exceptions.
- `ICEUserException`
  This is the root exception for all user exceptions. Catching `ICEUserException` catches all user exceptions (but not run-time exceptions).
- `ICELocalException`
  This is the root exception for all run-time exceptions. Catching `ICELocalException` catches all run-time exceptions (but not user exceptions).
- `ICETimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `ICEConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, an `ICEConnectTimeoutException` can be handled as `ICEConnectTimeoutException`, `ICETimeoutException`, `ICELocalException`, `ICEException`, or `NSException`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `ICELocalException`; the fine-grained error handling offered by the remainder of the hierarchy is of interest mainly in the implementation of the Ice run time. Exceptions to this rule are the exceptions related to facet and object life cycles, which you may want to catch explicitly. These exceptions are `ICEFacetNotExistException` and `ICEObjectNotExistException`, respectively.

Creating and Initializing Run-Time Exceptions in Objective-C

`ICELocalException` provides two properties that return the file name and line number at which an exception was raised:

Objective-C

```
@interface ICELocalException : ICEException
{
// ...

@property(nonatomic, readonly) NSString* file;
@property(nonatomic, readonly) int line;

-(id)init:(const char*)file line:(int)line;
+(id)localException:(const char*)file line:(int)line;
@end
```

The `init` method and the convenience constructor accept the file name and line number as arguments.

Concrete run-time exceptions that derived from `ICEException` provide a corresponding `init` method and convenience constructor. For example, here is the Slice definition of `ObjectNotExistException`:

Slice

```
local exception RequestFailedException {
    Identity id;
    string facet;
    string operation;
};

local exception ObjectNotExistException extends RequestFailedException {};
```

The Objective-C mapping for `ObjectNotExistException` is:

Objective-C

```
@interface ICEObjectNotExistException : ICERequestFailedException
// ...
-(id) init:(const char*)file__p line:(int)line__p;
-(id) init:(const char*)file__p
          line:(int)line__p
          id_:(ICEIdentity *)id_
          facet:(NSString *)facet
          operation:(NSString *)operation;
+(id) objectNotExistException:(const char*)file__p
                           line:(int)line__p;
+(id) objectNotExistException:(const char*)file__p
                           line:(int)line__p
                           id_:(ICEIdentity *)id_
                           facet:(NSString *)facet
                           operation:(NSString *)operation;
@end
```

In other words, as for user exceptions, run-time exceptions provide `init` methods and convenience constructors that accept arguments in base-to-derived order. This means that all run-time exceptions require a file name and line number when they are instantiated. For example, you can throw an `ICEObjectNotExistException` as follows:

Objective-C

```
@throw [ICEObjectNotExistException objectNotExistException:__FILE__ line:__LINE__];
```

If you throw this exception in the context of an executing operation on the server side, the `id_`, `facet`, and `operation` instance variables are automatically initialized by the Ice run time.

When you instantiate a run-time exception, the base `NSException` is initialized such that its `name` method returns the same string as `ice_name`; the `reason` and `userInfo` methods return `nil`.

Copying and Deallocating Exceptions in Objective-C

User exceptions and run-time exceptions implement the `NSCopying` protocol, so you can copy them. The semantics are the same as for structures.

Similarly, like structures, exceptions implement a `dealloc` method that takes care of deallocating the instance variables when an exception is released.

Exception Comparison and Hashing in Objective-C

Exceptions do not override `isEqual` or `hash`, so these methods have the behavior inherited from `NSObject`.

See Also

- User Exceptions