Understanding Objects and Exceptions

On page:

Distinguishing Objects and Servants

The term Ice object refers to a conceptual entity that is incarnated by a programming language object called a servant. An Ice object typically implements a Slice interface, but could implement a Slice class with operations instead.

In addition, applications can transfer instances of Slice classes *by value*, using classes as just another type in the data model, along with other Slice types such as structures and sequences. We refer to this usage as transferring *objects by value*, to distinguish it from the formal concept of *Ice objects*.

As a refresher on Slice syntax, consider the following definitions:

Slice

```
class ClassWithOperations
{
    int cost;

    void updateCost();
};

interface Remote
{
    ClassWithOperations getByValue();
    ClassWithOperations* getByProxy();
};
```

As its name implies, the operation `getByValue` returns a `ClassWithOperations` object by value, whereas the `getByProxy` operation returns a proxy to a remote Ice object that implements `ClassWithOperations`. Although their syntax is quite similar, the semantics of these two operations are very different. Calling `getByValue`, a client receives a *local* instance of `ClassWithOperations` on which it can access the `cost` member and call the `updateCost` method. With `getByProxy`, a client can use the returned proxy to call `updateCost` as a remote invocation but cannot access the `cost` member, which is local state for the remote servant.

Design Considerations for Exceptions

Using exceptions to report error conditions is a generally accepted idiom, given its wide support in programming languages and its convenience in simplifying a developer's error-handling responsibilities. The alternatives, such as using numeric error codes, can be unwieldy in practice and lack the flexibility that a well-designed exception hierarchy provides. Slice does not impose any particular error-handling style on developers, but if you choose to use Slice exceptions, you should understand their advantages and limitations.

Slice exception types support single inheritance, which allows you to compose hierarchies of related error conditions. Furthermore, exceptions may have any number of data members, meaning your errors can convey as little or as much detail as you like. One limitation of exceptions is that they are treated differently than other Slice data types: exceptions cannot be passed as parameters, and cannot be used as data members. The sole purpose of an exception type is to appear in the `throws` clause of an operation.

As an Ice developer, you are not expected to know the intricacies of how Ice transfers exceptions from server to client. However, it can be very useful for you to understand that exceptions are composed of slices, that multiple encoding formats are available, and that passing exceptions through intermediaries may require additional effort.

Design Considerations for Objects

The motivation for using objects by value is not as simple as for exceptions. If you wish to use the exception idiom, you have no choice but to use the Slice exception model. However, an application is not forced to use objects by value. Other user-defined Slice types, such as sequences, dictionaries, and structures, may satisfy the data model requirements of many applications. Classes and structures may seem similar at first glance because they can both be used as data containers, but there are also some significant differences between them. Be sure you understand the architectural implications of classes before using them in your application. Granted, if your data model requires the use of inheritance, polymorphism, or self-referential capabilities, you really have no other choice but to use classes.

One aspect of using objects by value that must not be overlooked is overhead, both in terms of the processing effort required to encode and decode them, as well as the space consumed in messages that use them. In Ice 3.4 and earlier, objects by value were relatively expensive, likely causing many developers to use structures when classes might have been the more convenient choice. In Ice 3.5, the new compact format makes objects by value much more competitive with structures. Ice 3.5 also makes it possible to forward objects of unknown types for situations where flexibility takes precedence over compactness.

Composition using Slices

The data encoding rules for classes and exceptions rely on the concept of *slices* in which each slice corresponds to a level in the type hierarchy and contains the encoded data members defined at that level. The low-level details of the encoding rules are not relevant here, but having a basic understanding of how Ice marshals and unmarshals classes and exceptions can help you with your design decisions. Consider this example:

Slice

```
class A
{
    int i;
};

class B extends A
{
    float f;
};

class C extends B
{
    string s;
};
```

The encoded form of an instance of class `C` is summarized below:

Slice	Contents
`C`	`string s`
`B`	`float f`
`A`	`int i`

Notice that the instance is encoded in order from most-derived to least-derived. For the sake of simplicity, we have omitted many of the encoding details. For instance, we have not shown that the initial (most-derived) slice includes a type ID string such as `"::MyModule::C"` that identifies the object's type.

Although we used a class hierarchy in the example above, the behavior for exceptions is very similar.

Let's add the following interface to our discussion:

Slice

```
interface I
{
    B getObject();
};
```

Now suppose a client invokes `getObject`. Let's also suppose that the servant actually returns an instance of class `C`, which is perfectly legal given that `C` derives from `B`. The Ice run time in the client knows that the formal return type of the `getObject` operation is `B` and unmarshals the object as follows:

1. Extract the type ID of the initial slice (`"::MyModule::C"`).
2. If this type is known to the client, instantiate the object, decode the state for each of its slices, and return the object.
3. If type `C` is *not* known to the client, which can occur when client and server are using different versions of the Slice definitions, the Ice run time discards the encoded data for slice `C` (also known as *slicing* the object), reads the type ID of the next slice, and repeats step 2.
4. If type `B` is also not known to the client then we have a problem. First, from a logical standpoint, the client *must* know type `B` because it is the statically-declared return type of the operation that the client just invoked. Second, we cannot slice this object any further because it would no longer be compatible with the formal signature of the operation; the returned object must at least be an instance of `B`, so we could not return an instance of `A`. In either case, the Ice run time would raise an exception.

Generally speaking, to unmarshal an instance of a class or exception, the Ice run time discards the slices of unknown types until it finds a type that it recognizes, exhausts all slices, or can no longer satisfy the formal type signature of the operation. This slicing feature allows the receiver, whose Slice definitions may be limited or outdated, to continue to function properly even when it does not recognize the most-derived type.

With this knowledge, we can now discuss new features introduced in Ice 3.5 that affect the run time's encoding rules for classes and exceptions. Namely, a choice of encoding formats and the ability to preserve the slices of unknown types. First, however, you should understand the need for object factories.

The Factory Requirement

For Slice classes that define data members but no operations, receiving an object by value is straightforward: the Ice run time examines the type ID e ncoded with the object, translates that (in a language-specific manner) into the corresponding generated class, and creates an instance of that class to represent the object. This happens without any help from the application. However, things get more complicated when a class defines operations. By definition, such a class is abstract, therefore Ice cannot allocate an instance of the generated abstract class. Only the application knows which derived class implements an abstract Slice class, so Ice requires the application to supply factories for translating type IDs into their programming language equivalent objects.

Ice does not require or support user-supplied factories for exceptions because exceptions cannot define operations.

The encoding for classes does not inform the receiver whether an object is abstract. As Ice processes each slice of an object, it first attempts to locate the corresponding generated class. If Ice cannot find the class, or finds the class but determines that it is abstract, the run time looks for a factory registered by the application for that type ID. If no factory is found to create the object, Ice may discard the slice as described above.

Here is some sample code to demonstrate the implementation and registration of a factory:

Java

```
private static class CWOFactory implements Ice.ObjectFactory
{
    public Ice.Object create(String typeId)
    {
        assert(typeId.equals(ClassWithOperations.ice_staticId()));
        return new ClassWithOperationsImpl();
    }

    public void destroy()
    {
    }
}

private static class ClassWithOperationsImpl extends ClassWithOperations
{
    public void updateCost()
    {
        // ...
    }
}

Ice.Communicator communicator = ...;
communicator.addObjectFactory(new CWOFactory(), ClassWithOperations.ice_staticId());
```

Our factory is only intended to create instances of the `ClassWithOperations` type we defined earlier, which is the reason for the assertion. In your own application, you can partition the factory responsibilities however you wish. For example, you can define a separate factory class for each Slice class, or use a single factory class that knows how to create multiple types. Whatever you decide, you must register a factory for each abstract class that the program has the potential to receive, and the registrations must occur before any remote invocations are made that might return one of these types.

You can also register a factory for a concrete Slice class if you wish to substitute a derived type, or simply because you want more control over the allocation of that type.

Slice Formats

Version 1.0 of the Ice encoding, which is used by all versions of Ice through 3.4, supports a single format for encoding the slices of classes and exceptions. The primary design goal for this format was the ability to support slicing at the expense of a somewhat larger encoding. The fact that the encoding embeds the type IDs of each type in an instance's type hierarchy means the size of the encoded data is affected by the symbol names and nesting depth of the Slice definitions. Consider this example:

Slice

```
module TopLevelModule
{
    module IntermediateModule
    {
        class Account { ... };
        class SubcontractorAccount extends Account { ... };
    };
};
```

The encoding for an instance of `SubcontractorAccount` includes the following type ID strings:

- `::TopLevelModule::IntermediateModule::SubcontractorAccount`
- `::TopLevelModule::IntermediateModule::Account`
- `::Ice::Object`

The last ID serves as a terminator to indicate the last slice and reflects the fact that all object types ultimately derive from the base `Object` type.

As you can see, there is a non-trivial amount of overhead for passing objects by value. Note however that the encoding includes a lookup table for type IDs to minimize this overhead as much as possible, meaning each unique type ID is only encoded once per message. While sending a single instance of `SubcontractorAccount` might be relatively expensive, sending a collection of `SubcontractorAccount` objects amortizes this cost over a number of instances. We can contrive situations where a message contains many instances of unrelated types, in which case the encoded type IDs would have a greater impact on the size of the resulting message, but those scenarios are less common.

The slice format for classes and exceptions is nearly identical. The main difference is that, for an exception, there is no need for a type ID lookup table because there can never be more than one instance of an exception in a message.

With version 1.1 of the encoding introduced in Ice 3.5, you now have two slice formats to choose from: *compact* and *sliced*.

Compact Format

The compact format in version 1.1 of the encoding sacrifices the slicing feature of classes and exceptions to reduce the encoded size of these types. The format still encodes the type ID of the most-derived type, and continues to use a lookup table to prevent duplication of type IDs in a message, but omits the type IDs for any intermediate types. Version 1.1 of the encoding also eliminates the need to encode the type ID for the base `Object` type of class instances and, together with other optimizations, makes the encoded size of classes in the compact format much more competitive with that of structures.

Ice 3.5 uses the compact format by default. Existing applications that rely on the slicing feature may begin receiving `NoObjectFactoryException` errors after migrating to Ice 3.5, an indication that the run time does not recognize the most-derived type and cannot slice to a less-derived type because the compact format omits the necessary type information. These applications must use the sliced format instead, which behaves similarly to the format in version 1.0 of the encoding.

Using our previous example, an instance of `SubcontractorAccount` still includes the type ID string `"::TopLevelModule::IntermediateModule::SubcontractorAccount"` in the compact format, but omits the type ID strings for `Account` and `Object`. Clearly we could reduce this overhead even further by using shorter symbol names and eliminating the nested module, or we could omit the type ID string altogether by using a compact type ID.

Sliced Format

The sliced format in version 1.1 of the encoding retains the slicing behavior of version 1.0: a receiver discards the slices of unknown types until it finds a type that it recognizes, exhausts all available slices, or slices so far that the result would violate the formal signature of the operation. The sliced format still encodes the type IDs of the most-derived type as well as all intermediate types. However, unlike the 1.0 encoding, Ice no longer encodes the type ID for the base `Object` type in class instances such that, along with other improvements in the 1.1 encoding, the sliced format is still more efficient than the 1.0 encoding.

Choosing a Format

Ice uses the compact format by default, so an application that needs slicing behavior must explicitly enable it as follows:

- Set the `Ice.Default.SlicedFormat` property to a non-zero value to force the Ice run time to use the sliced format by default.
- Annotate your Slice definitions with `format` metadata to selectively enable the sliced format for certain operations or interfaces.

For example, suppose an application can safely use the compact format most of the time, but still needs slicing in a few situations. In this case the application can use metadata to enable the sliced format where necessary:

Slice

```
interface Ledger
{
    Account getAccount(string id); // Uses compact format

    ["format:sliced"]
    Account importAccount(string source); // Uses sliced format
};
```

The semantics implied by these definitions state that the caller of `getAccount` assumes it will know every type that that might be returned, but the same cannot be said for `importAccount`. By enabling the sliced format here, we allow the client to "slice off" what it does not recognize, even if that means the client is left with only an instance of `Account` and not an instance of some derived type.

Now let's examine the opposite case: use the sliced format by default, and the compact format only in certain cases:

Slice

```
["format:sliced"]
interface Ledger
{
    ["format:compact"]
    Account getAccount(string id); // Uses compact format

    Account importAccount(string source); // Uses sliced format
};
```

Here we specify that all operations in `Ledger` use the sliced format unless overridden at the operation level, which we do for `getAccount`.

Keep the following points in mind when using the `format` metadata:

- The chosen format only controls how Ice encodes a value and has no effect on the decoding process.
- The format affects the input parameters, output parameters, and exceptions of an operation. Consider this example:

  Slice

  ```
  exception IncompatibleAccount { ... };

  interface Ledger
  {
      ["format:compact"]
      Account migrateAccount(Account oldAccount) throws IncompatibleAccount;
  };
  ```

  The metadata forces the client to use the compact format when marshaling the input parameter `oldAccount`, and forces the server to use the compact format when marshaling the return value or the exception. For a given operation, it is not possible to use one format for the parameters and a different format for the exceptions.
- If you decide to use the `Ice.Default.SlicedFormat` property, be aware that this property only affects the sender of a class or exception. For example, if you enable this property in the client but not the server, then all objects sent by the client use the sliced format by default, but all objects and exceptions returned by the server use the compact format by default.

By offering two alternative formats, Ice gives you a great deal of flexibility in designing your applications. The compact format is ideal for applications that place a greater emphasis on efficiency, while the sliced format is helpful when clients and servers evolve independently.

Optional Objects

Declaring a data member of type `class` to be optional does not eliminate the Ice run time's requirement for type information. Suppose a client uses the following Slice definitions:

Slice

```
class UserInfo
{
    string name;
    optional(1) string organization;
};
```

Also suppose that the server is using a newer version of the definitions:

Slice

```
class GroupInfo
{
    ...;
};

class UserInfo
{
    string name;
    optional(1) string organization;
    optional(2) GroupInfo group;
};
```

When receiving an instance of `UserInfo`, the Ice run time in the client ignores the `group` member because it does not recognize its tag, but Ice must still be able to unmarshal (or skip) the `GroupInfo` instance data in the message. If the server used the compact format for its reply, the client will receive an exception while attempting to decode the message. On the other hand, this scenario would succeed when using the sliced format because the format embeds enough information for the receiver to skip instances of unknown types.

The lesson here is that adding a new class type to your application can have unexpected repercussions, especially when using the compact format. Remember that the *sender* can change formats without necessarily needing to update the receiver. For example, after adding the `group` member to `UserInfo`, the server could begin using the sliced format for all operations that return `UserInfo` to ensure that any existing clients would be able to ignore this member.

Preserving Slices

The concept of slicing involves discarding the slices of unknown types when decoding an instance of a Slice class or exception. Here is a simple example:

Slice

```
class Base
{
    int b;
};

class Intermediate extends Base
{
    int i;
};

class Derived extends Intermediate
{
    int d;
};

interface Relay
{
    Base transform(Base b);
};
```

The server implementing the `Relay` object must know the type `Base` (because it is statically referenced in the interface definition), but may not know `Intermediate` or `Derived`. Suppose the implementation of `transform` involves forwarding the object to another back-end server for processing and returning the transformed object to the caller. In effect, the `Relay` server is an intermediary. If the `Relay` server does not know the types `Intermediate` and `Derived`, it will slice an instance to `Base` and discard the data members of any more-derived types, which is clearly not the intended result because the back-end server *does* know those types. The only way the `Relay` server could successfully forward these objects is by knowing all possible derived types, which makes the application more difficult to evolve over time because the intermediary must be updated each time a new derived type is added.

To address this limitation, Ice supports a new metadata directive that allows an instance of a Slice class or exception to be forwarded with all of its slices intact, even if the intermediary does not know one or more of the instance's derived types. The new directive, `preserve-slice`, is shown below:

Slice

```
["preserve-slice"]
class Base
{
    int b;
};

class Intermediate extends Base
{
    int i;
};

class Derived extends Intermediate
{
    int d;
};

["format:sliced"]
interface Relay
{
    Base transform(Base b);
};
```

With this change, all instances of `Base`, and types derived from `Base`, are encoded in a format that allows them to be forwarded intact. Also notice the addition of the `format:sliced` metadata on the `Relay` interface, which ensures that its operations use the sliced format and not the default compact format.

Unknown Sliced Objects

The combination of preserved slices and the sliced format means Ice applications now have a subtle new capability that was not available prior to Ice 3.5. Suppose we modify our `Relay` example as shown below:

Slice

```
["preserve-slice"]
class Base
{
    int b;
};

class Intermediate extends Base
{
    int i;
};

class Derived extends Intermediate
{
    int d;
};

["format:sliced"]
interface Relay
{
    Object transform(Object b);
};
```

The only difference here is the signature of the `transform` operation, which now uses the `Object` type. Technically, it is not necessary for the intermediary server to know *any* of the class types that might be relayed via this new definition of `transform` because the formal types in its signature do not impose any requirements. As long as any object types known by the intermediary are marked with `preserve-slice`, and the `transform` operation uses the sliced format, this intermediary is capable of relaying objects of any type.

If the Ice run time in the intermediary does not know any of the types in an object's inheritance hierarchy, and the formal type is `Object`, Ice uses an instance of `UnknownSlicedObject` to represent the object:

C++

```
namespace Ice {
class UnknownSlicedObject : public Object, ... {
public:

    const std::string& getUnknownTypeId() const;
    SlicedDataPtr getSlicedData() const;

    ...
};
}
```

The implementation of `transform` receives an instance of `UnknownSlicedObject` and can use that object as its return value. If necessary, the implementation can determine the most-derived type of the object by calling `getUnknownTypeId`. The actual encoded state of the object is encapsulated by the `SlicedData` class:

Slice

```
namespace Ice {
struct SliceInfo : public IceUtil::Shared {
    std::string typeId;
    std::vector<Byte> bytes;
    std::vector<ObjectPtr> objects;
    bool hasOptionalMembers;
    bool isLastSlice;
};


class SlicedData : public ... {
public:
    SlicedData(const SliceInfoSeq&);
    const SliceInfoSeq slices;

    ...
};
}
```

Applications do not normally need to use these types, but they provide all of the information the run time requires to forward an instance.

The definition of `UnknownSlicedObject`, `SlicedData` and `SliceInfo` are very similar in the other language mappings and are not shown here.

See Also

- Terminology
- Architectural Implications of Classes
- Classes Versus Structures
- Data Encoding for Classes
- Data Encoding for Exceptions
- Slice Metadata Directives
- Classes with Compact Type IDs