User Exceptions

On this page:

User Exception Syntax and Semantics

Looking at the `setTime` operation in the `Clock` interface, we find a potential problem: given that the `TimeOfDay` structure uses `short` as the type of each field, what will happen if a client invokes the `setTime` operation and passes a `TimeOfDay` value with meaningless field values, such as `-199` for the minute field, or `42` for the hour? Obviously, it would be nice to provide some indication to the caller that this is meaningless. Slice allows you to define user exceptions to indicate error conditions to the client. For example:

Slice

```
exception Error {}; // Empty exceptions are legal

exception RangeError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};
```

A user exception is much like a structure in that it contains a number of data members. However, unlike structures, exceptions can have zero data members, that is, be empty. Like classes, user exceptions support inheritance and may include optional data members.

Default Values for User Exception Members

You can specify a default value for an exception data member that has one of the following types:

- An integral type (`byte`, `short`, `int`, `long`)
- A floating point type (`float` or `double`)
- `string`
- `bool`
- `enum`

For example:

Slice

```
exception RangeError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
    string reason = "out of range";
};
```

The legal syntax for literal values is the same as for Slice constants, and you may also use a constant as a default value. The language mapping guarantees that data members are initialized to their declared default values using a language-specific mechanism.

Declaring User Exceptions in Operations

Exceptions allow you to return an arbitrary amount of error information to the client if an error condition arises in the implementation of an operation. Operations use an exception specification to indicate the exceptions that may be returned to the client:

Slice

```
interface Clock {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time)
        throws RangeError, Error;
};
```

This definition indicates that the `setTime` operation may throw either a `RangeError` or an `Error` user exception (and no other type of exception). If the client receives a `RangeError` exception, the exception contains the `TimeOfDay` value that was passed to `setTime` and caused the error (in the `errorTime` member), as well as the minimum and maximum time values that can be used (in the `minTime` and `maxTime` members). If `setTime` failed because of an error not caused by an illegal parameter value, it throws `Error`. Obviously, because `Error` does not have data members, the client will have no idea what exactly it was that went wrong — it simply knows that the operation did not work.

An operation can throw only those user exceptions that are listed in its exception specification. If, at run time, the implementation of an operation throws an exception that is not listed in its exception specification, the client receives a run-time exception) to indicate that the operation did something illegal. To indicate that an operation does not throw any user exception, simply omit the exception specification. (There is no empty exception specification in Slice.)

Restrictions for User Exceptions

Exceptions are not first-class data types and first-class data types are not exceptions:

- You cannot pass an exception as a parameter value.
- You cannot use an exception as the type of a data member.
- You cannot use an exception as the element type of a sequence.
- You cannot use an exception as the key or value type of a dictionary.
- You cannot throw a value of non-exception type (such as a value of type `int` or `string`).

The reason for these restrictions is that some implementation languages use a specific and separate type for exceptions (in the same way as Slice does). For such languages, it would be difficult to map exceptions if they could be used as an ordinary data type. (C++ is somewhat unusual among programming languages by allowing arbitrary types to be used as exceptions.)

User Exception Inheritance

Exceptions support inheritance. For example:

Slice

```
exception ErrorBase {
    string reason;
};

enum RTError {
    DivideByZero, NegativeRoot, IllegalNull /* ... */
};

exception RuntimeError extends ErrorBase {
    RTError err;
};

enum LError { ValueOutOfRange, ValuesInconsistent, /* ... */ };

exception LogicError extends ErrorBase {
    LError err;
};

exception RangeError extends LogicError {
    TimeOfDay errorTime;
    TimeOfDay minTime;
    TimeOfDay maxTime;
};
```

These definitions set up a simple exception hierarchy:

- `ErrorBase` is at the root of the tree and contains a string explaining the cause of the error.
- Derived from `ErrorBase` are `RuntimeError` and `LogicError`. Each of these exceptions contains an enumerated value that further categorizes the error.
- Finally, `RangeError` is derived from `LogicError` and reports the details of the specific error.

Setting up exception hierarchies such as this not only helps to create a more readable specification because errors are categorized, but also can be used at the language level to good advantage. For example, the Slice C++ mapping preserves the exception hierarchy so you can catch exceptions generically as a base exception, or set up exception handlers to deal with specific exceptions.

Looking at the exception hierarchy, it is not clear whether, at run time, the application will only throw most derived exceptions, such as `RangeError`, or if it will also throw base exceptions, such as `LogicError`, `RuntimeError`, and `ErrorBase`. If you want to indicate that a base exception, interface, or class is abstract (will not be instantiated), you can add a comment to that effect.

Note that, if the exception specification of an operation indicates a specific exception type, at run time, the implementation of the operation may also throw more derived exceptions. For example:

Slice

```
exception Base {
    // ...
};

exception Derived extends Base {
    // ...
};

interface Example {
    void op() throws Base;      // May throw Base or Derived
};
```

In this example, `op` may throw a `Base` or a `Derived` exception, that is, any exception that is compatible with the exception types listed in the exception specification can be thrown at run time.

As a system evolves, it is quite common for new, derived exceptions to be added to an existing hierarchy. Assume that we initially construct clients and server with the following definitions:

Slice

```
exception Error {
    // ...
};

interface Application {
    void doSomething() throws Error;
};
```

Also assume that a large number of clients are deployed in field, that is, when you upgrade the system, you cannot easily upgrade all the clients. As the application evolves, a new exception is added to the system and the server is redeployed with the new definition:

Slice

```
exception Error {
    // ...
};

exception FatalApplicationError extends Error {
    // ...
};

interface Application {
    void doSomething() throws Error;
};
```

This raises the question of what should happen if the server throws a `FatalApplicationError` from `doSomething`. The answer depends whether the client was built using the old or the updated definition:

- If the client was built using the same definition as the server, it simply receives a `FatalApplicationError`.
- If the client was built with the original definition, that client has no knowledge that `FatalApplicationError` even exists. In this case, the Ice run time automatically slices the exception to the most-derived type that is understood by the receiver (`Error`, in this case) and discards the information that is specific to the derived part of the exception. (This is exactly analogous to catching C++ exceptions by value — the exception is sliced to the type used in the `catch`-clause.)

Exceptions support single inheritance only. (Multiple inheritance would be difficult to map into many programming languages.)

See Also

- Constants and Literals
- Operations
- Run-Time Exceptions
- Proxies
- Interface Inheritance
- Optional Data Members