Java Mapping for Exceptions

Previous Next

On this page:

Java Mapping for User Exceptions

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

Slice

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

These exception definitions map as follows:

Java

```
public class GenericError extends com.zeroc.Ice.UserException
{
    public GenericError()
    {
        this.reason = "";
    }

    public GenericError(Throwable cause)
    {
        super(cause);
        this.reason = "";
    }

    public GenericError(String reason)
    {
        this.reason = reason;
    }

    public GenericError(String reason, Throwable cause)
    {
        super(cause);
        this.reason = reason;
    }

    public String ice_id()
    {
        return "::M::GenericError";
    }

    public String reason;

    ...
}
```

```
public class BadTimeVal extends GenericError
{
    public BadTimeVal()
    {
        super();
    }

    public BadTimeVal(Throwable cause)
    {
        super(cause);
    }

    public BadTimeVal(String reason)
    {
        super(reason);
    }

    public BadTimeVal(String reason, Throwable cause)
    {
        super(reason, cause);
    }

    public String ice_id()
    {
        return "::M::BadTimeVal";
    }
    ...
}

public class BadZoneName extends GenericError
{
    public BadZoneName()
    {
        super();
    }

    public BadZoneName(Throwable cause)
    {
        super(cause);
    }

    public BadZoneName(String reason)
    {
        super(reason);
    }

    public BadZoneName(String reason, Throwable cause)
    {
        super(reason, cause);
    }

    public String ice_id()
    {
        return "::M::BadZoneName";
    }


    ...
}
```

Each Slice exception is mapped to a Java class with the same name. For each data member, the corresponding class contains a public data member. (Obviously, because `BadTimeVal` and `BadZoneName` do not have members, the generated classes for these exceptions also do not have members.) A JavaBean-style API is used for optional data members, and you can customize the mapping to force required members to use this same API.

The inheritance structure of the Slice exceptions is preserved for the generated classes, so `BadTimeVal` and `BadZoneName` inherit from `GenericError`.

Each exception also defines an `ice_id` method, which returns the Slice type ID of the exception.

All user exceptions are derived from the base class `UserException`. This allows you to catch all user exceptions generically by installing a handler for `UserException`. `UserException`, in turn, derives from `java.lang.Exception`.

`UserException` implements a `clone` method that is inherited by its derived exceptions, so you can make member-wise shallow copies of exceptions.

Note that the generated exception classes contain other methods that are not shown. However, those methods are internal to the Java mapping and are not meant to be called by application code.

Back to Top ^

Java Constructors for User Exceptions

Exceptions have a default constructor that initializes data members as follows:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

The constructor won't explicitly initialize a data member if the default Java behavior for that type produces the desired results.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value instead.

If an exception declares or inherits any data members, the generated class provides a second constructor that accepts one parameter for each data member so that you can construct and initialize an instance in a single statement (instead of first having to construct the instance and then assign to its members). For a derived exception, this constructor accepts one argument for each base exception member, plus one argument for each derived exception member, in base-to-derived order.

The generated class may include an additional constructor if the exception declares or inherits any optional data members.

The Slice compiler generates overloaded versions of all constructors that accept a trailing `Throwable` argument for preserving an exception chain.

Back to Top ^

Java 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 `LocalException` (which, in turn, derives indirectly from `java.lang.RuntimeException`).

`LocalException` implements a `clone` method that is inherited by its derived exceptions, so you can make member-wise shallow copies of exceptions.

Recall the inheritance diagram for user and run-time exceptions. By catching exceptions at the appropriate point in the hierarchy, you can handle exceptions according to the category of error they indicate:

- `LocalException`
  This is the root of the inheritance tree for run-time exceptions.
- `UserException`
  This is the root of the inheritance tree for user exceptions.
- `TimeoutException`
  This is the base exception for both operation-invocation and connection-establishment timeouts.
- `ConnectTimeoutException`
  This exception is raised when the initial attempt to establish a connection to a server times out.

For example, a `ConnectTimeoutException` can be handled as `ConnectTimeoutException`, `TimeoutException`, `LocalException`, or `java.lang.Exception`.

You will probably have little need to catch run-time exceptions as their most-derived type and instead catch them as `LocalException`; 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 `FacetNotExistException` and `ObjectNotExistException`, respectively.

Back to Top ^

See Also

- User Exceptions
- Run-Time Exceptions
- Java Mapping for Optional Data Members
- JavaBean Mapping
- Versioning
- Object Life Cycle

Previous

Next