Operations

On this page:

Parameters and Return Values

An operation definition must contain a return type and zero or more parameter definitions. For example, in the `Clock` interface, the `getTime` operation has a return type of `TimeOfDay` and the `setTime` operation has a return type of `void`. You must use `void` to indicate that an operation returns no value — there is no default return type for Slice operations.

An operation can have one or more input parameters. For example, `setTime` accepts a single input parameter of type `TimeOfDay` called `time`. Of course, you can use multiple input parameters:

Slice

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);
    // ...
};
```

Note that the parameter name (as for Java) is mandatory. You cannot omit the parameter name, so the following is in error:

Slice

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay, TimeOfDay);  // Error!
    // ...
};
```

By default, parameters are sent from the client to the server, that is, they are input parameters. To pass a value from the server to the client, you can use an output parameter, indicated by the `out` keyword. For example, an alternative way to define the `getTime` operation in the `Clock` interface would be:

Slice

```
void getTime(out TimeOfDay time);
```

This achieves the same thing but uses an output parameter instead of the return value. As with input parameters, you can use multiple output parameters:

Slice

```
interface CircadianRhythm {
    void setSleepPeriod(TimeOfDay startTime, TimeOfDay stopTime);
    void getSleepPeriod(out TimeOfDay startTime, out TimeOfDay stopTime);
    // ...
};
```

If you have both input and output parameters for an operation, the output parameters must follow the input parameters:

Slice

```
void changeSleepPeriod(    TimeOfDay startTime,         TimeOfDay stopTime,     // OK
                       out TimeOfDay prevStartTime, out TimeOfDay prevStopTime);

void changeSleepPeriod(out TimeOfDay prevStartTime, out TimeOfDay prevStopTime, // Error
                           TimeOfDay startTime,         TimeOfDay stopTime);
```

Slice does not support parameters that are both input and output parameters (call by reference). The reason is that, for remote calls, reference parameters do not result in the same savings that one can obtain for call by reference in programming languages. (Data still needs to be copied in both directions and any gains in marshaling efficiency are negligible.) Also, reference (or input-output) parameters result in more complex language mappings, with concomitant increases in code size.

Style of Operation Definition

As you would expect, language mappings follow the style of operation definition you use in Slice: Slice return types map to programming language return types, and Slice parameters map to programming language parameters.

For operations that return only a single value, it is common to return the value from the operation instead of using an out-parameter. This style maps naturally into all programming languages. Note that, if you use an out-parameter instead, you impose a different API style on the client: most programming languages permit the return value of a function to be ignored whereas it is typically not possible to ignore an output parameter.

For operations that return multiple values, it is common to return all values as out-parameters and to use a return type of `void`. However, the rule is not all that clear-cut because operations with multiple output values can have one particular value that is considered more "important" than the remainder. A common example of this is an iterator operation that returns items from a collection one-by-one:

Slice

```
bool next(out RecordType r);
```

The `next` operation returns two values: the record that was retrieved and a Boolean to indicate the end-of-collection condition. (If the return value is `false`, the end of the collection has been reached and the parameter `r` has an undefined value.) This style of definition can be useful because it naturally fits into the way programmers write control structures. For example:

```
while (next(record))
    // Process record...

if (next(record))
    // Got a valid record...
```

Overloading Operations

Slice does not support any form of overloading of operations. For example:

Slice

```
interface CircadianRhythm {
    void modify(TimeOfDay startTime, TimeOfDay endTime);
    void modify(    TimeOfDay startTime,        // Error
                    TimeOfDay endTime,
                out TimeOfDay prevStartTime,
                out TimeOfDay prevEndTime);
};
```

Operations in the same interface must have different names, regardless of what type and number of parameters they have. This restriction exists because overloaded functions cannot sensibly be mapped to languages without built-in support for overloading.

Name mangling is not an option in this case: while it works fine for compilers, it is unacceptable to humans.

Idempotent Operations

Some operations, such as `getTime` in the `Clock` interface, do not modify the state of the object they operate on. They are the conceptual equivalent of C++ `const` member functions. Similary, `setTime` does modify the state of the object, but is idempotent. You can indicate this in Slice as follows:

Slice

```
interface Clock {
    idempotent TimeOfDay getTime();
    idempotent void setTime(TimeOfDay time);
};
```

This marks the `getTime` and `setTime` operations as idempotent. An operation is idempotent if two successive invocations of the operation have the same effect as a single invocation. For example, `x = 1;` is an idempotent operation because it does not matter whether it is executed once or twice — either way, `x` ends up with the value 1. On the other hand, `x += 1;` is not an idempotent operation because executing it twice results in a different value for `x` than executing it once. Obviously, any read-only operation is idempotent.

The `idempotent` keyword is useful because it allows the Ice run time to be more aggressive when performing automatic retries to recover from errors. Specifically, Ice guarantees *at-most-once* semantics for operation invocations:

- For normal (not idempotent) operations, the Ice run time has to be conservative about how it deals with errors. For example, if a client sends an operation invocation to a server and then loses connectivity, there is no way for the client-side run time to find out whether the request it sent actually made it to the server. This means that the run time cannot attempt to recover from the error by re-establishing a connection and sending the request a second time because that could cause the operation to be invoked a second time and violate at-most-once semantics; the run time has no option but to report the error to the application.
- For `idempotent` operations, on the other hand, the client-side run time can attempt to re-establish a connection to the server and safely send the failed request a second time. If the server can be reached on the second attempt, everything is fine and the application never notices the (temporary) failure. Only if the second attempt fails need the run time report the error back to the application. (The number of retries can be increased with an Ice configuration parameter.)

See Also

- Interfaces, Operations, and Exceptions
- User Exceptions
- Run-Time Exceptions
- Proxies
- Interface Inheritance
- Automatic Retries