Java Mapping for Operations

On this page:

Basic Java Mapping for Operations

As we saw in the mapping for interfaces, for each operation on an interface, the proxy class contains a corresponding member function with the same name. To invoke an operation, you call it via the proxy. For example, here is part of the definitions for our file system:

Slice

```
module Filesystem {
    interface Node {
        idempotent string name();
    };
    // ...
};
```

The `name` operation returns a value of type `string`. Given a proxy to an object of type `Node`, the client can invoke the operation as follows:

Java

```
NodePrx node = ...;             // Initialize proxy
String name = node.name();      // Get name via RPC
```

This illustrates the typical pattern for receiving return values: return values are returned by reference for complex types, and by value for simple types (such as `int` or `double`).

Normal and `idempotent` Operations in Java

You can add an `idempotent` qualifier to a Slice operation. As far as the signature for the corresponding proxy method is concerned, `idempotent` has no effect. For example, consider the following interface:

Slice

```
interface Example {
                string op1();
    idempotent  string op2();
};
```

The proxy interface for this is:

Java

```
public interface ExamplePrx extends Ice.ObjectPrx {
    public String op1();
    public String op2();
}
```

Because `idempotent` affects an aspect of call dispatch, not interface, it makes sense for the two methods to be mapped the same.

Passing Parameters in Java

In-Parameters in Java

The parameter passing rules for the Java mapping are very simple: parameters are passed either by value (for simple types) or by reference (for complex types and type `string`). Semantically, the two ways of passing parameters are identical: it is guaranteed that the value of a parameter will not be changed by the invocation (with some caveats — see Location Transparency).

Here is an interface with operations that pass parameters of various types from client to server:

Slice

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ClientToServer {
    void op1(int i, float f, bool b, string s);
    void op2(NumberAndString ns, StringSeq ss, StringTable st);
    void op3(ClientToServer* proxy);
};
```

The Slice compiler generates the following proxy for these definitions:

Java

```
public interface ClientToServerPrx extends Ice.ObjectPrx {
    public void op1(int i, float f, boolean b, String s);
    public void op2(NumberAndString ns, String[] ss, java.util.Map st);
    public void op3(ClientToServerPrx proxy);
}
```

Given a proxy to a `ClientToServer` interface, the client code can pass parameters as in the following example:

Java

```
ClientToServerPrx p = ...;              // Get proxy...

p.op1(42, 3.14f, true, "Hello world!"); // Pass simple literals

int i = 42;
float f = 3.14f;
boolean b = true;
String s = "Hello world!";
p.op1(i, f, b, s);                      // Pass simple variables

NumberAndString ns = new NumberAndString();
ns.x = 42;
ns.str = "The Answer";
String[] ss = { "Hello world!" };
java.util.HashMap st = new java.util.HashMap();
st.put(new Long(0), ns);
p.op2(ns, ss, st);                      // Pass complex variables

p.op3(p);                               // Pass proxy
```

Out-Parameters in Java

Java does not have pass-by-reference: parameters are always passed by value. For a function to modify one of its arguments, we must pass a reference (by value) to an object; the called function can then modify the object's contents via the passed reference.

To permit the called function to modify a parameter, the Java mapping uses *holder* classes. For example, for each of the built-in Slice types, such as `int` and `string`, the `Ice` package contains a corresponding holder class. Here are the definitions for the holder classes `Ice.IntHolder` and `Ice.StringHolder`:

Java

```
package Ice;

public final class IntHolder {
    public IntHolder() {}
    public IntHolder(int value) {
        this.value = value;
    }
    public int value;
}

public final class StringHolder {
    public StringHolder() {}
    public StringHolder(String value) {
        this.value = value;
    }
    public String value;
}
```

A holder class has a public `value` member that stores the value of the parameter; the called function can modify the value by assigning to that member. The class also has a default constructor and a constructor that accepts an initial value.

For user-defined types, such as structures, the Slice-to-Java compiler generates a corresponding holder type. For example, here is the generated holder type for the `NumberAndString` structure we defined earlier:

Java

```
public final class NumberAndStringHolder {
    public NumberAndStringHolder() {}

    public NumberAndStringHolder(NumberAndString value) {
        this.value = value;
    }

    public NumberAndString value;
}
```

This looks exactly like the holder classes for the built-in types: we get a default constructor, a constructor that accepts an initial value, and the public `value` member.

Note that holder classes are generated for *every* Slice type you define. For example, for sequences, such as the `FruitPlatter` sequence, the compiler does not generate a special Java `FruitPlatter` type because sequences map to Java arrays. However, the compiler does generate a `FruitPlatterHolder` class, so we can pass a `FruitPlatter` array as an out-parameter.

To pass an out-parameter to an operation, we simply pass an instance of a holder class and examine the `value` member of each out-parameter when the call completes. Here are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction:

Slice

```
struct NumberAndString {
    int x;
    string str;
};

sequence<string> StringSeq;

dictionary<long, StringSeq> StringTable;

interface ServerToClient {
    void op1(out int i, out float f, out bool b, out string s);
    void op2(out NumberAndString ns,
             out StringSeq ss,
             out StringTable st);
    void op3(out ServerToClient* proxy);
};
```

The Slice compiler generates the following code for these definitions:

Java

```
public interface ClientToServerPrx extends Ice.ObjectPrx {
    public void op1(Ice.IntHolder i, Ice.FloatHolder f,
                    Ice.BooleanHolder b, Ice.StringHolder s);
    public void op2(NumberAndStringHolder ns,
                    StringSeqHolder ss, StringTableHolder st);
    public void op3(ClientToServerPrxHolder proxy);
}
```

Given a proxy to a `ServerToClient` interface, the client code can pass parameters as in the following example:

Java

```
ClientToServerPrx p = ...;              // Get proxy...

Ice.IntHolder ih = new Ice.IntHolder();
Ice.FloatHolder fh = new Ice.FloatHolder();
Ice.BooleanHolder bh = new Ice.BooleanHolder();
Ice.StringHolder sh = new Ice.StringHolder();
p.op1(ih, fh, bh, sh);

NumberAndStringHolder nsh = new NumberAndString();
StringSeqHolder ssh = new StringSeqHolder();
StringTableHolder sth = new StringTableHolder();
p.op2(nsh, ssh, sth);

ServerToClientPrxHolder stcph = new ServerToClientPrxHolder();
p.op3(stch);

System.out.writeln(ih.value);   // Show one of the values
```

Again, there are no surprises in this code: the various holder instances contain values once the operation invocation completes and the `value` member of each instance provides access to those values.

Null Parameters in Java

Some Slice types naturally have "empty" or "not there" semantics. Specifically, sequences, dictionaries, and strings all can be `null`, but the corresponding Slice types do not have the concept of a null value. To make life with these types easier, whenever you pass `null` as a parameter or return value of type sequence, dictionary, or string, the Ice run time automatically sends an empty sequence, dictionary, or string to the receiver.

This behavior is useful as a convenience feature: especially for deeply-nested data types, members that are sequences, dictionaries, or strings automatically arrive as an empty value at the receiving end. This saves you having to explicitly initialize, for example, every string element in a large sequence before sending the sequence in order to avoid `NullPointerException`. Note that using null parameters in this way does *not* create null semantics for Slice sequences, dictionaries, or strings. As far as the object model is concerned, these do not exist (only *empty* sequences, dictionaries, and strings do). For example, whether you send a string as `null` or as an empty string makes no difference to the receiver: either way, the receiver sees an empty string.

Exception Handling in Java

Any operation invocation may throw a run-time exception and, if the operation has an exception specification, may also throw user exceptions. Suppose we have the following simple interface:

Slice

```
exception Tantrum {
    string reason;
};

interface Child {
    void askToCleanUp() throws Tantrum;
};
```

Slice exceptions are thrown as Java exceptions, so you can simply enclose one or more operation invocations in a `try-catch` block:

Java

```
ChildPrx child = ...;   // Get child proxy...

try {
    child.askToCleanUp();
} catch (Tantrum t) {
    System.out.write("The child says: ");
    System.out.writeln(t.reason);
}
```

Typically, you will catch only a few exceptions of specific interest around an operation invocation; other exceptions, such as unexpected run-time errors, will typically be handled by exception handlers higher in the hierarchy. For example:

Java

```
public class Client {
    static void run() {
        ChildPrx child = ...;   // Get child proxy...
        try {
            child.askToCleanUp();
        } catch (Tantrum t) {
            System.out.print("The child says: ");
            System.out.println(t.reason);
            child.scold();          // Recover from error...
        }
        child.praise();             // Give positive feedback...
    }

    public static void
    main(String[] args)
    {
        try {
            // ...
            run();
            // ...
        } catch (Ice.LocalException e) {
            e.printStackTrace();
        } catch (Ice.UserException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

This code handles a specific exception of local interest at the point of call and deals with other exceptions generically. (This is also the strategy we used for our first simple application.)

Exceptions and Out-Parameters

The Ice run time makes no guarantees about the state of out-parameters when an operation throws an exception: the parameter may still have its original value or may have been changed by the operation's implementation in the target object. In other words, for out-parameters, Ice provides the weak exception guarantee [1] but does not provide the strong exception guarantee.

> This is done for reasons of efficiency: providing the strong exception guarantee would require more overhead than can be justified.

See Also

- Operations
- Java Mapping for Exceptions
- Java Mapping for Sequences
- Java Mapping for Interfaces

- Location Transparency

References

1. Sutter, H. 1999. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Reading, MA: Addison-Wesley.