Java Mapping for Operations

Previous Next

On this page:

- Basic Java Mapping for Operations
- Normal and idempotent Operations in Java
- Passing Parameters in Java
  - In Parameters in Java
  - Out Parameters in Java
  - Null Parameters in Java
  - Optional Parameters in Java
- Exception Handling in Java
  - Exceptions and Out-Parameters

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`).

Back to Top ^

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 ObjectPrx
{
    String op1();
    String op2();
}
```

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

Back to Top ^

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.

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 ObjectPrx
{
    void op1(int i, float f, boolean b, String s);
    void op2(NumberAndString ns, String[] ss, java.util.Map<Long, String[]> st);
    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.Map<Long, String[]> st = new java.util.HashMap<Long, String[]>();
st.put(0, ns);
p.op2(ns, ss, st);                      // Pass complex variables

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

Back to Top ^

Out Parameters in Java

The mapping for an operation depends on how many values it returns, including out parameters and a non-`void` return value:

- Zero values
  The corresponding Java method returns `void`. For the purposes of this discussion, we're not interested in these operations.

- One value
  The corresponding Java method returns the mapped type, regardless of whether the Slice definition of the operation declared it as a return value or as an out parameter. Consider this example:

  Slice

  ```
  interface I
  {
      string op1();
      void op2(out string name);
  }
  ```

  The mapping generates corresponding methods with identical signatures:

  Java

  ```
  interface IPrx extends ObjectPrx
  {
      String op1();
      String op2();
  }
  ```

- Multiple values
  The Slice-to-Java translator generates an extra nested class to hold the results of an operation that returns multiple values. The class is nested in the mapped interface (not the proxy interface) and has the name *Op*`Result`, where *Op* represents the name of the operation. The leading character of the class name for a "result class" is always capitalized. The values of out parameters are provided in corresponding data members of the same names. If the operation declares a return value, its value is provided in the data member named `returnValue`. If an out parameter is also named `returnValue`, the data member to hold the operation's return value is named `_returnValue` instead. The result class defines an empty constructor as well as a "one-shot" constructor that accepts and assigns a value for each of its data members. The corresponding Java method returns the result class type.

Here are the same Slice definitions we saw earlier, but this time with all parameters being passed in the `out` direction, along with one additional operation to better demonstrate the mapping:

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);
    StringSeq op4(out string returnValue);
}
```

The Slice compiler generates the following code for these definitions:

Java

```
public interface ServerToClientPrx extends ObjectPrx
{
    ServerToClient.Op1Result op1();
    ServerToClient.Op2Result op2();
    ServerToClientPrx op3();
    ServerToClient.Op4Result op4();
}

public interface ServerToClient extends ...
{
    public static class Op1Result
    {
        public Op1Result() {}
        public Op1Result(int i, float f, boolean b, String s)
        {
            this.i = i;
            this.f = f;
            this.b = b;
            this.s = s;
        }
        public int i;
        public float f;
        public boolean b;
        public String s;
    }

    public static class Op2Result
    {
        public Op2Result() {}
        public Op2Result(NumberAndString ns, String[] ss, java.util.Map<java.lang.Long, String[]> st)
        {
            this.ns = ns;
            this.ss = ss;
            this.st = st;
        }
        public NumberAndString ns;
        public String[] ss;
        public java.util.Map<java.lang.Long, String[]> st;
    }

    public static class Op4Result
    {
        public Op4Result() {}
        public Op4Result(String[] _returnValue, String returnValue)
        {
            this._returnValue = _returnValue;
            this.returnValue = returnValue;
        }
        public String[] _returnValue;
        public String returnValue;
    }
}
```

We need to point out several things here:

- Result classes are generated for `op1`, `op2` and `op4` because they return multiple values
- The result classes are generated as nested classes of interface `ServerToClient`, and **not** `ServerToClientPrx`
- `op4` declares an out parameter named `returnValue`, therefore `Op4Result` declares a data member named `_returnValue` to hold the operation's return value
- No result class is necessary for `op3` because it only returns one value; the mapped Java method declares a return type of `ServerToClientPrx` even though the Slice operation declared it as an out parameter

Back to Top ^

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.

Back to Top ^

Optional Parameters in Java

The mapping uses standard Java types to encapsulate optional parameters:

- `java.util.OptionalDouble`
  The mapped type for an optional `double`.
- `java.util.OptionalInt`
  The mapped type for an optional `int`.
- `java.util.OptionalLong`
  The mapped type for an optional `long`.
- `java.util.Optional<T>`
  The mapped type for all other Slice types.

Optional return values and output parameters are mapped to instances of the above classes, depending on their types. For operations with optional in para meters, the proxy provides a set of overloaded methods that accept them as optional values, and another set of methods that accept them as required values. Consider the following operation:

Slice

```
optional(1) int execute(optional(2) string params);
```

The mapping for this operation is shown below:

Java

```
java.util.OptionalInt execute(String params);
java.util.OptionalInt execute(java.util.Optional<String> params);
```

For cases where you are passing values for all of the optional in parameters, it is more efficient to use the required mapping and avoid creating temporary optional values.

A client can invoke `execute` as shown below:

Java

```
java.util.OptionalInt i;

i = proxy.execute("--file log.txt");                        // required mapping
i = proxy.execute(java.util.Optional.of("--file log.txt")); // optional mapping
i = proxy.execute(java.util.Optional.empty());              // params is unset

if(i.isPresent())
{
    System.out.println("value = " + i.get());
}
```

Passing `null` where an optional value is expected is equivalent to passing an instance whose value is unset.

Java's optional classes do not consider `null` to be a legal value. Consider this example:

Slice

```
class Data
{
    ...
}

interface Repository
{
    void addOptional(optional(1) Data d);
    void addRequired(Data d);
}
```

The Ice encoding allows `null` values for class instances, so you can pass `null` to `addRequired` and the server will receive it as `null`. However, there's no way to pass an "optional value set to null" in the Java mapping. Passing `null` to `addOptional` is equivalent to passing the value of `java.util.Optional.ofNullable((T)null)`, which is equivalent to passing the value of `java.util.Optional.empty()`. In either case, the server will receive it as an optional whose value is not present.

A well-behaved program must not assume that an optional parameter always has a value. Calling `get` on an optional instance for which no value is set raises `java.util.NoSuchElementException`.

Back to Top ^

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(LocalException e)
        {
            e.printStackTrace();
        }
        catch(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.)

Back to Top ^

Exceptions and Out-Parameters

For the Java Compat mapping, 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.

Back to Top ^

See Also

- Operations
- Java Mapping for Exceptions
- Java Mapping for Interfaces
- Java Mapping for Optional Data Members
- Collocated Invocation and Dispatch

References

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

Previous

Next