Parameter Passing in Java

For each parameter of a Slice operation, the Java mapping generates a corresponding parameter for the method in the `_<interface-name>Operations` interface. In addition, every operation has a trailing parameter of type `Ice.Current`. For example, the `name` operation of the `Node` interface has no parameters, but the `name` method of the `_NodeOperations` interface has a single parameter of type `Ice.Current`. We will ignore this parameter for now.

Passing Required Parameters in Java

Parameter passing on the server side follows the rules for the client side. To illustrate the rules, consider the following interface that passes string parameters in all possible directions:

Slice

```
module M {
    interface Example {
        string op(string sin, out string sout);
    };
};
```

The generated skeleton class for this interface looks as follows:

Java

```
public interface _ExampleOperations
{
    String op(String sin, Ice.StringHolder sout, Ice.Current current);
}
```

As you can see, there are no surprises here. For example, we could implement op as follows:

Java

```
public final class ExampleI extends M._ExampleDisp {

    public String op(String sin, Ice.StringHolder sout, Ice.Current current)
    {
        System.out.println(sin);     // In params are initialized
        sout.value = "Hello World!"; // Assign out param
        return "Done";
    }
}
```

This code is in no way different from what you would normally write if you were to pass strings to and from a function; the fact that remote procedure calls are involved does not impact on your code in any way. The same is true for parameters of other types, such as proxies, classes, or dictionaries: the parameter passing conventions follow normal Java rules and do not require special-purpose API calls.

Passing Optional Parameters in Java

Suppose we modify the example above to use optional parameters:

Slice

```
module M {
    interface Example {
        optional(1) string op(optional(2) string sin, out optional(3) string sout);
    };
};
```

The generated skeleton now looks like this:

Java

```
public interface _ExampleOperations
{
    String op(Ice.Optional<String> sin, Ice.StringHolder sout, Ice.Current current);
}
```

For the sake of performance, the default mapping treats optional out parameters and return values as if they are required. If your servant needs the ability to return an optional value, you must add the `java:optional` metadata tag:

Slice

```
module M {
    interface Example {
        ["java:optional"]
        optional(1) string op(optional(2) string sin, out optional(3) string sout);
    };
};
```

This tag can be applied to an interface if you want to use the optional mapping for all of the operations in that interface, or to individual operations as shown here. With this change, the mapping now returns optional values:

Java

```
public interface _ExampleOperations
{
    Ice.Optional<String> op(Ice.Optional<String> sin, Ice.Optional<String> sout, Ice.Current current);
}
```

The `java:optional` tag affects the return value and all out parameters; it is not possible to modify the mapping only for certain parameters.

See Also

- Server-Side Java Mapping for Interfaces
- Java Mapping for Operations
- Raising Exceptions in Java
- Tie Classes in Java
- The Current Object