Server-Side Java Mapping for Interfaces

The server-side mapping for interfaces provides an up-call API for the Ice run time: by implementing member functions in a servant class, you provide the hook that gets the thread of control from the Ice server-side run time into your application code.

On this page:

Skeleton Classes in Java

On the client side, interfaces map to proxy classes. On the server side, interfaces map to *skeleton* classes. A skeleton is a class that has a pure virtual member function for each operation on the corresponding interface. For example, consider our Slice definition for the `Node` interface:

Slice

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

The Slice compiler generates the following definition for this interface:

Java

```
package Filesystem;

public interface _NodeOperations
{
    String name(Ice.Current current);
}

public interface _NodeOperationsNC
{
    String name();
}

public interface Node extends Ice.Object,
                              _NodeOperations,
                              _NodeOperationsNC {}

public abstract class _NodeDisp extends Ice.ObjectImpl
                                implements Node
{
    // Mapping-internal code here...
}
```

The important points to note here are:

- As for the client side, Slice modules are mapped to Java packages with the same name, so the skeleton class definitions are part of the `Filesystem` package.
- For each Slice interface <*`interface-name`*>, the compiler generates Java interfaces `_`<*`interface-name`*>`Operations` and `_`<*`interface-name`*>`OperationsNC` (`_NodeOperations` and `_NodeOperationsNC` in this example). These interfaces contain a method for each operation in the Slice interface. (You can ignore the `Ice.Current` parameter for now.)

- For each Slice interface <*interface-name*>, the compiler generates a Java interface <*interface-name*> (`Node` in this example). That interface extends `Ice.Object` and the two operations interfaces.
- For each Slice interface <*interface-name*>, the compiler generates an abstract class _<*interface-name*>`Disp` (`_NodeDisp` in this example). This abstract class is the actual skeleton class; it is the base class from which you derive your servant class.

Servant Classes in Java

In order to provide an implementation for an Ice object, you must create a servant class that inherits from the corresponding skeleton class. For example, to create a servant for the `Node` interface, you could write:

Java

```
package Filesystem;

public final class NodeI extends _NodeDisp {

    public NodeI(String name)
    {
        _name = name;
    }

    public String name(Ice.Current current)
    {
        return _name;
    }

    private String _name;
}
```

By convention, servant classes have the name of their interface with an `I`-suffix, so the servant class for the `Node` interface is called `NodeI`. (This is a convention only: as far as the Ice run time is concerned, you can choose any name you prefer for your servant classes.) Note that `NodeI` extends _`NodeDisp`, that is, it derives from its skeleton class.

As far as Ice is concerned, the `NodeI` class must implement only a single method: the `name` method that it inherits from its skeleton. This makes the servant class a concrete class that can be instantiated. You can add other member functions and data members as you see fit to support your implementation. For example, in the preceding definition, we added a _`name` member and a constructor. (Obviously, the constructor initializes the _`name` member and the `name` function returns its value.)

Normal and `idempotent` Operations in Java

Whether an operation is an ordinary operation or an `idempotent` operation has no influence on the way the operation is mapped. To illustrate this, consider the following interface:

Slice

```
interface Example {
                void   normalOp();
    idempotent void   idempotentOp();
    idempotent string readonlyOp();
};
```

The operations class for this interface looks like this:

Java

```
public interface _ExampleOperations
{
    void normalOp(Ice.Current current);
    void idempotentOp(Ice.Current current);
```

```
    String readonlyOp(Ice.Current current);
}
```

Note that the signatures of the member functions are unaffected by the `idempotent` qualifier.

See Also

- Slice for a Simple File System
- Java Mapping for Interfaces
- Parameter Passing in Java
- Raising Exceptions in Java
- Tie Classes in Java
- The Current Object