Tie Classes in Java

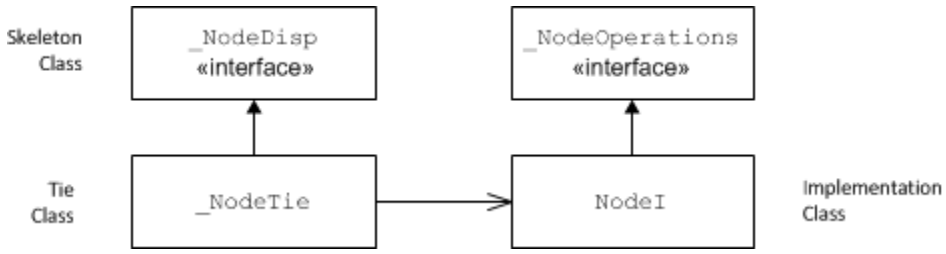
The mapping to skeleton classes requires the servant class to inherit from its skeleton class. Occasionally, this creates a problem: some class libraries require you to inherit from a base class in order to access functionality provided by the library; because Java does not support multiple implementation inheritance, this means that you cannot use such a class library to implement your servants because your servants cannot inherit from both the library class and the skeleton class simultaneously.

To allow you to still use such class libraries, Ice provides a way to write servants that replaces inheritance with delegation. This approach is supported by *tie classes*. The idea is that, instead of inheriting from the skeleton class, you simply create a class (known as an *implementation class* or *delegate class*) that contains methods corresponding to the operations of an interface. You use the `--tie` option with the `slice2java` compiler to create a tie class. For example, the `--tie` option causes the compiler to create exactly the same code for the Node interface as we saw previously, but to also emit an additional tie class. For an interface `<`*`interface-name`*`>`, the generated tie class has the name `_<`*`interface-name`*`>Tie`:

Java

```
package Filesystem;

public class _NodeTie extends _NodeDisp implements Ice.TieBase {

    public _NodeTie() {}

    public
    _NodeTie(_NodeOperations delegate)
    {
        _ice_delegate = delegate;
    }

    public java.lang.Object
    ice_delegate()
    {
        return _ice_delegate;
    }

    public void
    ice_delegate(java.lang.Object delegate)
    {
        _ice_delegate = (_NodeOperations)delegate;
    }

    public boolean
    equals(java.lang.Object rhs)
    {
        if (this == rhs)
        {
            return true;
        }
        if (!(rhs instanceof _NodeTie))
        {
            return false;
        }

        return _ice_delegate.equals(((_NodeTie)rhs)._ice_delegate);
    }

    public int
    hashCode()
    {
        return _ice_delegate.hashCode();
    }

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

    private _NodeOperations _ice_delegate;
}
```

This looks a lot worse than it is: in essence, the generated tie class is simply a servant class (it extends `_NodeDisp`) that delegates to your implementation class each invocation of a method corresponding to a Slice operation:



A skeleton class, tie class, and implementation class.

The generated tie class also implements the `Ice.TieBase` interface, which defines methods for obtaining and changing the delegate object:

Java

```
package Ice;

public interface TieBase {
    java.lang.Object ice_delegate();
    void ice_delegate(java.lang.Object delegate);
}
```

The delegate has type `java.lang.Object` in these methods in order to allow a tie object's delegate to be manipulated without knowing its actual type. However, the `ice_delegate` modifier raises `ClassCastException` if the given delegate object is not of the correct type.

Given this machinery, we can create an implementation class for our `Node` interface as follows:

Java

```
package Filesystem;

public final class NodeI implements _NodeOperations {

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

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

    private String _name;
}
```

Note that this class is identical to our previous implementation, except that it implements the `_NodeOperations` interface and does not extend `_NodeDisp` (which means that you are now free to extend any other class to support your implementation).

To create a servant, you instantiate your implementation class and the tie class, passing a reference to the implementation instance to the tie constructor:

Java

```
NodeI fred = new NodeI("Fred");         // Create implementation
_NodeTie servant = new _NodeTie(fred);  // Create tie
```

Alternatively, you can also default-construct the tie class and later set its delegate instance by calling `ice_delegate`:

Java

```
_NodeTie servant = new _NodeTie();      // Create tie
// ...
NodeI fred = new NodeI("Fred");         // Create implementation
// ...
servant.ice_delegate(fred);             // Set delegate
```

When using tie classes, it is important to remember that the tie instance is the servant, not your delegate. Furthermore, you must not use a tie instance to incarnate an Ice object until the tie has a delegate. Once you have set the delegate, you must not change it for the lifetime of the tie; otherwise, undefined behavior results.

You should use the tie approach only when necessary, that is, if you need to extend some base class in order to implement your servants: using the tie approach is more costly in terms of memory because each Ice object is incarnated by two Java objects (the tie and the delegate) instead of just one. In addition, call dispatch for ties is marginally slower than for ordinary servants because the tie forwards each operation to the delegate, that is, each operation invocation requires two function calls instead of one.

Also note that, unless you arrange for it, there is no way to get from the delegate back to the tie. If you need to navigate back to the tie from the delegate, you can store a reference to the tie in a member of the delegate. (The reference can, for example, be initialized by the constructor of the delegate.)

See Also

- Server-Side Java Mapping for Interfaces
- Parameter Passing in Java
- Raising Exceptions in Java
- Object Incarnation in Java