Create Meta Object Adapters

Table of contents

1. Introduction

MetaObject adapters are the fundamental building blocks of all things meta. They are responsible for adapting different types of regular objects to the MetaObject interface. Meta-JB provides built in adapters for Java beans, standard hashmaps, etc. but it is frequently necessary to adapt additional object types.

In this How-to I will write a MetaObject implementation for the Java preferences API. This will involve creating a PreferencesMetaObject to implement the MetaObject interface and a PreferencesMetaKit to provide more general access to the adapter layer.

2. PreferencesMetaObject

The PreferencesMetaObject will pass any calls to MetaObject.setProperty() and MetaObject.getProperty() through to the underlying java.util.prefs.Preferences object that it wraps. The MetaClass will be used as a template for how the types should be stored and retrieved.

2.1. Things to Consider

The preferences API primarily deals with strings. MetaObjects allow any type of object to be stored. To properly adapt things we'll need to come up with a way of translating these values in enough cases to make the utility useful.

Another issue will be nested meta-objects. When a MetaObject has a property that is also a MetaObject, the underlying values are wrapped and unwrapped as they are retrieved from or stored into the property. A problem can occur when a MetaClass cannot be easily retrieved for a raw unwrapped object but it is being stored in a more generic type property. For example, if Foo extends Bar but the property type can take any Foo. We won't know to create another Bar when pulling the value back out because the Preferences object doesn't point us back to the Bar meta-class in any way. An object like a Java bean usually has a direct relationship to a MetaClass based on its Java type. The only thing we know about a Preferences object is its location and tying class to location would limit the way the preferences API could be used through the meta-object.

This is a pretty common issue and usually a combination of object caching and specific typing solves it. Fortunately, AbstractMetaObject handles most of the caching for us. Any object that is set into the MetaObject and subsequently unwrapped will be remembered.

Since preferences are persistent there is the possibility that the PreferencesMetaObject will need to wrap a Preferences node that it has never seen before. When specific type information within the containing MetaObject isn't sufficient, we can provide some location based meta-class resolution using regular expressions or something. Which is perhaps beyond the scope of this article.

For this article, we'll assume that specific property typing is a sufficient solution.

2.2. AbstractMetaObject

For almost every meta-object implementation, AbstractMetaObject will be the way to go. For all but the most complicated implementations, it is sufficient to simply override the abstract setPropertyValue() and getPropertyValue() methods to perform the proper access to the underlying objects.

2.3. String Properties

To make things simple, we'll start with String based properties since they require no translation. We'll also skip worrying about the MetaKit implementation at the moment. More on that later.

So, with that in mind our basic PreferencesMetaObject would look like:

```
public class PreferencesMetaObject extends AbstractMetaObject
{
    private Preferences node;

    public PreferencesMetaObject( Preferences node, MetaClass mClass )
    {
        super( mClass, null );

        this.node = node;
    }

    protected Object setPropertyValue( String name, Object value )
    {
        // Get the old value to return when we're done
        Object oldValue = node.get( name, null );

        node.put( name, String.valueOf(value) );

        return( oldValue );
    }

    protected Object getPropertyValue( String name )
    {
```

```
        String value = node.get( name, null );

        return( value );
    }

    public String toString()
    {
        return( getClass().getName() + "[" + node + "]" );
    }
}
```

And that's it. If all you wanted to do was access very flat data structures from a Preferences node then we would be done. However, we want to really exploit everything we can.

2.4. Non-String Properties

The preferences API supports several additional types of attributes stored on a node:

- Boolean
- Byte Array
- Double
- Float
- Int
- Long

Other than Byte Array these values are pretty easy to support natively since the preferences API ultimately stores them as strings anyway. This means that we only have to beef up our getPropertyValue() method a bit.

```
    protected Object getPropertyValue( String name )
    {
        // Find out what the raw type is
        PropertyType type = getMetaClass().getPropertyType( name );
        Class base = type.getBaseClass();

        // Make sure we handle primitive types as their
        // wrapper cousins.  int -> Integer, etc..
        base = Inspector.translateFromPrimitive( base );

        // We use the raw string value to see if the property
        // even exists
        String value = node.get( name, null );
        if( value == null )
            return( null );

        if( Boolean.class.isAssignableFrom( base ) )
            return( Boolean.valueOf( node.getBoolean( name, false ) ) );
        else if( Double.class.isAssignableFrom( base ) )
            return( new Double( node.getDouble( name, 0.0 ) ) );
```

```
        else if( Float.class.isAssignableFrom( base ) )
            return( new Float( node.getFloat( name, 0.0f ) ) );
        else if( Integer.class.isAssignableFrom( base ) )
            return( new Integer( node.getInt( name, 0 ) ) );
        else if( Long.class.isAssignableFrom( base ) )
            return( new Long( node.getLong( name, 0 ) ) );
        else if( String.class.isAssignableFrom( base ) )
            return( value );

        throw new UnsupportedOperationException( "Cannot convert string
to:" + base );
    }
```

And with that we handle all types that a normal preferences node can handle as its attributes. I'll discuss ways to potentially handle even more complex base types in the PreferencesMetaKit section. For now, let's figure out how to handle nested meta-object properties.

2.5. MetaObject-based Properties

A Preferences node can have nested nodes. It would be convenient if we could automatically wrap these nested nodes in an appropriate meta-class if they represent types that we know about. It's easy to conceive of how getPropertyValue() might work in this setting... indeed AbstractMetaObject will even take care of the wrapping for us if we've built a proper meta-kit.

It's the set operation that is a little tricky in this case. Because of the way that preferences are accessed, it really isn't possible for a caller to setup a PreferencesMetaObject and then pass it directly to us... because for it to have a proper Preferences node it would already be our child.

If we're getting a MetaObject from some other implementation layer then it is possible that we could copy all of the values into a local version. I think this goes beyond the scope of this particular discussion. Besides, AbstractMetaObject should be handling most of that for us.

In the interest of illustration, I will put some stubs into the setPropertyValue() method to show how one might deal with this in the normal case.

The stubbed setPropertyValue() method:

```
    protected Object setPropertyValue( String name, Object value )
    {
        PropertyType type = getMetaClass().getPropertyType( name );
        if( type instanceof MetaClassPropertyType )
            {
            // The object passed in is the already unwrapped
            // internal value... in this case a Preferences object.
            // Since we can't graft a new node onto the tree then
```

```
            // there is really nothing that we can do easily so
            // we'll ignore it.
            return;
            }

        // Get the old value to return when we're done
        Object oldValue = node.get( name, null );

        node.put( name, String.valueOf(value) );

        return( oldValue );
    }
```

The getPropertyValue() method modified to handle nested meta-objects:

```
    protected Object getPropertyValue( String name )
    {
        // Find out what the raw type is
        PropertyType type = getMetaClass().getPropertyType( name );

        if( type instanceof MetaClassPropertyType )
            {
            // AbstractMetaObject will wrap this for us so our job
            // is easy
            return( node.node( name ) );
            }

        Class base = type.getBaseClass();

        // Make sure we handle primitive types as their
        // wrapper cousins.  int -> Integer, etc..
        base = Inspector.translateFromPrimitive( base );

        // We use the raw string value to see if the property
        // even exists
        String value = node.get( name, null );
        if( value == null )
            return( null );

        if( Boolean.class.isAssignableFrom( base ) )
            return( Boolean.valueOf( node.getBoolean( name, false ) ) );
        else if( Double.class.isAssignableFrom( base ) )
            return( new Double( node.getDouble( name, 0.0 ) ) );
        else if( Float.class.isAssignableFrom( base ) )
            return( new Float( node.getFloat( name, 0.0f ) ) );
        else if( Integer.class.isAssignableFrom( base ) )
            return( new Integer( node.getInt( name, 0 ) ) );
        else if( Long.class.isAssignableFrom( base ) )
            return( new Long( node.getLong( name, 0 ) ) );
        else if( String.class.isAssignableFrom( base ) )
            return( value );

        throw new UnsupportedOperationException( "Cannot convert string
to:" + base );
```

```
    }
```

And that's all there is to it. Well, except that is will throw a NullPointerException when actually attempting to access meta-object typed properties. Here is where our null MetaKit short-cut is coming back to haunt us. The meta-kit is crucial for automatically wrapping and unwrapping MetaObjects of a particular implementation.

3. PreferencesMetaKit

A MetaKit is what provides access to the non-MetaObject specific parts of an adapter layer. Today, this mostly includes the ability to wrap and unwrap object values and to otherwise automatically determine certain things about an object in relation to an adapter layer.

MetaKit method summary:

- getInternalObject() - Unwraps a meta-object and returns its internal real object. In our case, this would be a Preferences node.
- getMetaClassForObject() - Attempts to find an appropriate MetaClass for a given real object. Without additional functionality, our meta-kit cannot really answer these questions.
- wrapObject() - The opposite of getInternalObject(), given a real object it will wrap it in a MetaObject. Because the MetaClass is provided, our meta-kit can handle this method even if the result may not be 100% accurate in all cases. (ie: The superclass problem mentioned before.)
- getMetaObjectFactory() - Provides a factory that can create new instances of a MetaObject for a given MetaClass. Our meta-kit cannot service these requests.

So, basically, we have to implement the getInternalObject() and wrapObject() methods. I'll also show how we can partially implement the getMetaClassForObject() method since we'll have to do some caching of other things anyway.

MetaKits are encouraged to return the same wrapper object for specific real object instances. In other words, when our PreferencesMetaKit is asked to wrap a Preferences node that we've wrapped before then we should return the pre-existing MetaObject... assuming the application is still holding a reference to it somewhere. However, this is not a requirement and does complicate the code slightly by incorporating a wrapper cache 1. For simplicity, this meta-kit will not perform that resolution.

Our basic PreferencesMetaKit:

```
public class PreferencesMetaKit implements MetaKit, Serializable
{
    public static final PreferencesMetaKit DEFAULT_KIT = new
```

```
PreferencesMetaKit();

    public PreferencesMetaKit()
    {
    }

    public Object getInternalObject( MetaObject object )
    {
        if( !(object instanceof PreferencesMetaObject) )
            throw new RuntimeException( "Object is not a
PreferencesMetaObject." );

        return( ((PreferencesMetaObject)object).getPreferencesNode() );
    }

    public MetaClass getMetaClassForObject( Object object,
MetaClassRegistry classRegistry )
    {
        // We can't determine a type just from the object
        return( null );
    }

    public MetaClass getMetaClassForObject( Object object )
    {
        // We can't determine a type just from the object
        return( null );
    }

    public MetaObject wrapObject( Object object, MetaClass mClass )
    {
        if( !(object instanceof Preferences) )
            throw new RuntimeException( "Unabled to wrap non-Preferences
object:" + object );

        return( new PreferencesMetaObject( (Preferences)object, mClass ) );
    }

    public MetaObjectFactory getMetaObjectFactory()
    {
        return( new PreferencesMetaObjectFactory() );
    }

    private class PreferencesMetaObjectFactory implements MetaObjectFactory
    {
        public MetaObject createMetaObject( MetaClass type )
        {
            throw new UnsupportedOperationException( "PreferenceMetaObjects
instantiation not supported." );
        }
    }
}
```

It's pretty straight-forward. With a few modifications to our PreferencesMetaObject to support referencing the meta-kit and to allow the kit to pull the preferences node:

```
    public PreferencesMetaObject( Preferences node, MetaClass mClass )
    {
        super( mClass, PreferencesMetaKit.DEFAULT_KIT );
        this.node = node;
    }

    protected Preferences getPreferencesNode()
    {
        return( node );
    }
```

And we're in business. Now a MetaObject hierarchy can be imposed upon a preferences tree which makes it available to all of the other Meta-JB features.

4. Advanced Types

Here is where I would discuss extending the above to handle list based properties. I'm going to wait a bit for that.

5. Additional Future Features

- Add an alternate constructor that allows the caller to specify a key in the preferences tree. This might make it easier to instantiate PreferencesMetaObjects right from XML.
- Support for PropertyInfo default values. It would be easy to retrieve these and pass the value to the call to the preferences node.
- A key to MetaClass mapping registry for the PreferencesMetaKit so that it could guess at MetaClasses for different preferences nodes.

6. Footnotes

1:

A MetaKit, when possible, should attempt to return the same MetaObject instance for the same real object instance. In other words, repeated calls to wrapObject() should return the same MetaObject instance. The reason for this is simple.

In some underlying implementations the real objects can reference one another. If Object A references Object C and Object B also references Object C then when possible a similar MetaObject instance relationship should be maintained so that Object C is always wrapped by the same MetaObject instance.

In the case of PreferencesMetaKit, it would seem like it isn't specifically necessary. There

will only be one way to access any given Preferences node from another node. However, if Object A references Object B which then references Object C then I should get the same MetaObject instance if I get A.B.C as if I get B.C... even if A and B are actually instantiated separately and are therefore different MetaObject instances.

Yes, it's a little confusing but is easily taken care of with a self-cleaning cache. See the implementation of MapMetaKit for details.