BRENDAN MACMILLAN
27 JANUARY 2004
VERSION 1.2

JSX

THE MISSING MANUAL

KNOWN ISSUES

As at: 27 January 2004

JSX has very few issues now; and implements both XML and JOS (Java Object Serialization) with a high degree of fidelity. Almost all outstanding issues have been rectified since the last edition of this manual.

XML

IMPLEMENTED: JSX recognizes apostrophes (') and quotes (") for attribute values, with the other being permitted within it without needing to escape it (in accordance with the XML spec). For example:

```
<string id='i0' value="everything in it's own time"/>
<string id='i1' value='or should that be "its"?'/>
```

Recognizing both styles is important for reading XML from another source. When writing, JSX itself always uses apostrophes (').

JOS, CORE METHODS

IMPLEMENTED: `writeUnshared` and `readUnshared` (introducted in Java 1.4) are not implemented

IMPLEMENTED: `readObjectNoData` (introduced in Java 1.4) is not implemented

IMPLEMENTED: `GetField.defaulted()`

NOT IMPLEMENTED: `GetField.getObjectStreamClass()` is not implemented. Internally, JSX does not use the ObjectStreamClass, and it never seems to be used in practice. However, if you need this, please submit a bug report, and we will add it in.

JOS, PROTECTED METHODS

Protected methods of ObjectInputStream and ObjectOutputStream are not used by objects that are being serialized or deserialized, and so are not required to fulfill the JSX promise of "handling all objects". These protected methods are call-backs, which subclasses of ObjectInputStream and ObjectOutputStream can override, to modify the serialization/deserialization behavior. No other class should call them. Some of these methods are only relevant for JOS's binary serialization; but others are also very useful for JSX, and these are implemented:

IMPLEMENTED: ObjectReader.resolveObject(Object obj)

IMPLEMENTED: ObjectReader.resolveClass(ObjectStreamClass v)

IMPLEMENTED: ObjectWriter.replaceObject(Object obj)

JSX relaxes some requirements of JOS: classes need not implement the java.io.Serializable interface; and classes need not define an SUID field for class evolution to work. JSX eliminates both these requirements to make serialization easier to work with.

OBJECTS TO XML AND BACK

Jsx enables you to save objects to a file, to be able to read that file back in and to recreate a copy of the original objects. This works very similarly to Java's built-in serialization mechanism (Java Object Serialization, or JOS). The difference is that Jsx saves the objects as XML, which is human readable and machine readable by non-Java tools, instead of the binary format of JOS.

DOWNLOAD

To start using Jsx, the first thing to do is to download it. The latest version is available from: http://jsx.org/download.html. At the time of writing, the latest version is "2.0.7.0", and so we could download it directly from http://jsx.org/JSX2.1.4.jar.

CLASSPATH

Next, you need to add it to your classpath. A quick way to get started is to specify it in your command line when compiling and running:

javac –classpath .;JSX2.1.4.jar YourTestCode.java

java –classpath .;JSX2.1.4.jar YourTestCode

(**Unix**: under unix, you need to use ":" instead of ";". For example, ".:JSX2.1.4.jar")

EXAMPLE CODE

For example, here is how to serialize an object to a file:

```
      ObjectWriter out = new ObjectWriter(new
FileWriter(fileName));
      out.writeObject(obj);
      out.close();
```

And to read it in again:

```
      ObjectReader in = new ObjectReader(new
FileReader(fileName));
      obj2 = in.readObject();
      in.close();
```

But there is slightly more to using JSX than that: when writing, Jsx may thrown an IOException so you need to catch that, or specify that your method throws it. When reading, Jsx may throw an IOException, or a ClassNotFoundException. Finally, IOException is in the java.io package, and the Jsx classes are in the JSX package – an easy way access these is to import java.io.* and import JSX.*.

And that is it.

Here is a full working example of FileXML:

```
import java.io.*;
import JSX.*;

public class FileXML {
   public static void main(String[] args) {
      String fileName = "my.xml";
      Object obj = "hello world";

      try {
         ObjectWriter out = new ObjectWriter(new
FileWriter(fileName));
         out.writeObject(obj);
         out.close();
      } catch (IOException io) { // handle it
      }

      Object obj2 = null;
      try {
         ObjectReader in = new ObjectReader(new
FileReader(fileName));
         obj2 = in.readObject();
         in.close();
      } catch (IOException io) { // handle it
      } catch (ClassNotFoundException noClass) { // handle it
      }

      System.err.println("Read in object: "+obj2);

   }
}
```

This writes the following XML:

```
<jsx major='1' minor='1' format='JSX.DataReader'>
 <string id='i0' value='hello world'/>
</jsx>
```

This may look like quite a lot of work to write out just "hello world" – but the beauty of serialization is that this same code will work for any object. It will serialize not just that object, but any other objects that it refers to (and any objects that they refer to etc, recursively). If any object is

referred to twice, the second and later references use the id to refer to it (eg "i0" in the above example). Circular references, where an object refers to itself (or to another object that refers to it etc) are handled the same way.

In other words, Jsx serializes a directed graph of objects, with the initial object being the root of the graph, and with the references from object to object being the directed arcs. Each object is a node, which may also have primitive data associated with it, as well as references to other objects. Deserialization creates a deep copy of that graph, with the same shape and primitive values, but using new objects.

This deep copy could exist in the same machine, at the same time as the original (as in the above example) – or it could exist much later in time, after the machine had been turned off and on again. It could also be recreated in a different machine altogether, after being streamed across the network via a socket.

Jsx works the same as JOS, with a few exceptions. One of them is that Jsx does not require object's classes to implement the "java.io.Serializable" interface – JOS requires this to prevent accidental serialization, but it is quite inconvenient, and so Jsx relaxes this restriction, and serializes objects of any class whether they implement this interface or not.

CUSTOMIZING AN OBJECT'S SERIAL FORM

In the previous chapter, we looked at how Jsx can be used from the outside – that is, how to serialize and deserialize an object graph to create a deep copy of the graph. In this chapter, we look at how Jsx can be used from the inside – that is, how to customize the serial form of a class.

There are a number of reasons why you might want to customize the serial form of a class – for this purpose, it is helpful to see the serial form of a class as a kind of a data API. This brings with it the problems of back-compatibility with previous versions of the API. It also suggests that not all information is appropriate to be exposed in an API, for logical and security reasons. Apart from this, there are the physical reasons of saving memory space and execution time by not writing out unnecessary information, and also for choosing an efficient format for the information that must be written.

TRANSIENT FIELDS

The default serial form is for all fields of each object to be serialized[1]. But fields marked as transient are not – so marking fields as "transient" is the simplest way to customize the serial form of a class. They are also not given a value upon deserialization, even if a field of that name is present in the XML (this matters if the field had evolved to become transient, and some objects had been persisted beforehand).

This can be very useful for hiding confidential information (eg a credit card number), for omitting information that it would not make sense to serialize (eg cached pre-calculations), or for cutting down the information written, to save time and save – because all references are followed, just one unnecessary reference to an object can drag in a huge graph of other objects (eg one reference to a GUI object).

"WHOLE" OBJECT VS CLASS "SLICE"

In understanding the more complex ways of customizing the serial form, it is important to recognize the structure of objects due to inheritance, and to distinguish between the "whole" objects, and a particular data "slice". The "slice" is the data contributed from one particular class or superclass; whereas the "whole" object is the data from all slices, taken as a single unit. For example, if Sub extends Super, then an instance of Sub will have data "slices" from both Sub and Super; and an instance of Super will have only a data "slice" from super – which will be the "whole" object. The

[1] Like JOS, it omits static fields, as these belong to the class, not each object. If each object did save them, there would be multiple copies of the one value, and if they differed, this class value would change depending on the last object deserialized. This doesn't make sense. However, it would make sense if the class values were save as a sort of special object in its own right – but JOS does not do this, and so Jsx does not either. Note that although Class objects can be serialized, they do not contain this information.

upshot is that just talking about a class (like Super) does not tell us whether we mean a slice or the whole – indeed, Super can be both, depending on which object it is in.

The "whole" vs "slice" distinction is important for custom serial forms, because some approaches work at the whole level, and some work at the slice level. Specifically, writeExternal/readExternal and writeReplace/readResolve operate on the "whole" object; and writeObject/readObject, PutFields/GetFields and serialPersistentFields are "per slice".

For example, the transient keyword above can only affect the field it is applied to, which is of course declared in a particular class - it can have no effect on inherited fields. That is, the transient keyword operates only one class "slice" of an object.

CLASS "SLICE"

WRITEOBJECT AND READOBJECT

As an alternative to the default serial form, an object may define `writeObject` and `readObject` methods that write and read its data, when called by Jsx (or JOS). These callback methods deal with only the information in that class - it makes no difference whether they are defined in a superclass or subclass. Both take one argument, and use this to write or read their data (see example below). When Jsx needs to serialize each class "slice" of an object, it invokes the writeObject of the class if it has one; during deserialization, it does the same for readObject. Don't the names "`writeObject`" and "`readObject`" confuse you! These callback methods are quite different from those used to *start* serialization and deserialization, even though they have the same name.

One advantage of these methods is that they allow you to process the data before writing it and after reading it. For example, here is CustomEg1, which keeps a total of days internally, but writes it out as a number of weeks and remainder days:

```
import JSX.*;
import java.io.*;
public class CustomEg1 {
   int days = 22;
   private void writeObject(ObjectOutputStream oos) throws
IOException {
      oos.writeInt(days/7); // weeks
      oos.writeInt(days%7); // remainder days
   }
   private void readObject(ObjectInputStream ois) throws
IOException, ClassNotFoundException {
      days = ois.readInt();
      days += ois.readInt();
   }
}
```

Note that both `writeObject` and `readObject` are **`private`**. This is a necessary part of their signature for serialization, restricts their operation to a particular class "slice" of an object – they are not inherited.

The example produces this XML:

```
<jsx major='1' minor='1' format='JSX.DataReader'>
 <object id='i0' class='CustomEg1'>
  <declaredClass class='CustomEg1'>
   <primitive type='int' value='3'/>
   <primitive type='int' value='1'/>
  </declaredClass>
 </object>
</jsx>
```

Note that the <object> tag refers to the "whole" object, but the <declaredClass> tag refers only to one class "slice" of the object.

To produce the XML, we can modify the FileXML example of chapter One, changing the seventh line from:

```
      Object obj = "hello world";
```

to

```
      Object obj = new CustomEg1();
```

The above source code use only writeInt() and readInt(), but the API has similar methods for all the primitive types and for objects.

In additional, it is also possible to combine the default serial form with writeObject and readObject callbacks. This can be useful for making a class back-compatible with a previous version, and also to add more data. This uses the `defaultWriteObject()` and `defaultReadObject()` methods, and here is what it looks like:

```
import JSX.*;
import java.io.*;
public class CustomEg2 {
   int days = 22;
   private void writeObject(ObjectOutputStream oos) throws
IOException {
      oos.defaultWriteObject();
      oos.writeInt(days/7); // weeks
      oos.writeInt(days%7); // remainder days
   }
   private void readObject(ObjectInputStream ois) throws
IOException, ClassNotFoundException {
      ois.defaultReadObject();
      days = ois.readInt();
      days += ois.readInt();
   }
}
```

Another use for these default calls is for pre-processing it before writing and post-processing it after reading. However, it is important that the default occur before any other data is written to or read from the stream – for example, the writeInt() and readInt() must come after the default calls.
And the XML:

```
<jsx major='1' minor='1' format='JSX.DataReader'>
 <object id='i0' class='CustomEg2'>
  <declaredClass class='CustomEg2'>
   <default>
    <primitive field='days' type='int' value='22'/>
   </default>
   <primitive type='int' value='3'/>
   <primitive type='int' value='1'/>
  </declaredClass>
 </object>
</jsx>
```

The `writeObject()` and `readObject()` methods provide a radical alternative to the default mechanism. Their format is user defined, in that it relies on the user to read the data correctly, to know how many items there are to deserialize, and to know the order and their types. Unlike fields, this data is not named (with a fieldname), and it is not typed at compile time.

PUTFIELD AND GETFIELD

The `PutField` and `GetField` classes are another part of JOS that Jsx implements. They are used within `writeObject()` and `readObject()`, and address some of the short-comings of customization by direct writing and reading. The PutField and GetField classes enable objects to customize their serial form using the same format as fields. Thus, they have a field name and a type, and can be back-compatible with previously serialized objects. This is especially useful when a class has evolved significantly, yet you need to be able to deserialize instances stored with the old version of the class.

Here is an example of using it:

```
import JSX.*;
import java.io.*;
public class CustomEg3 {
   int days = 22;
   private static final ObjectStreamField[]
serialPersistentFields = {
      new ObjectStreamField("weeks", int.class),
      new ObjectStreamField("remainder_days", int.class)
   };
   private void writeObject(ObjectOutputStream oos) throws
IOException {
      ObjectOutputStream.PutField pf = oos.putFields();
      pf.put("weeks", days/7);
      pf.put("remainder_days", days%7);
      oos.writeFields();
   }
   private void readObject(ObjectInputStream ois) throws
IOException, ClassNotFoundException {
      ObjectInputStream.GetField gf = ois.readFields();
      days = gf.get("weeks", 0) * 7;
      days += gf.get("remainder_days", 0);
   }
}
```

This API is not the easiest to use, despite its conceptual elegance. As in JOS, only serializable fields can be written (that is, an ordinary non-static, non-transient field); however, it is also possible to define virtual serializable fields using the serialPersistentFields array. These virtual fields are necessary for writing; but any field that is present in the stream can be read, regardless of whether it is a serializable field or not (this is important when a class has evolved and serializable fields have been removed - so that a serializable field that was defined at the time the instance was written no longer is defined when the serialized representation of that instance is read).

For writing, we first need to obtain an instance of `PutField` from the stream. We can then `put()` our virtual serial fields and their values into that instance. The type of the field is determined by the type of the value argument - `put` is overloaded on each of the primitive types. In this example, the type is `int`. But this has not yet actually written out the fields; we need to explicitly do so with `writeFields()`, a method from the stream, not from PutField (the reason for this is because JOS writes fields in a canonical order, and so needs to know them all before writing).

For reading, obtaining the GetField object and reading the data values requires only the one call to `readFields()`. The second argument in the `get` methods is the default value (to bes returned if that field is not present in the stream), but it also serves the role of selecting the correct overloaded method, based on its type. This can result in awkward code to select the correct method, for example:

```
      atoms = gf.get("universeSize", (long)0);
```

The XML from the example code follows – note that it is indistinguishable from a class that really did have `weeks` and `remainder_days` fields:

```
<jsx major='1' minor='1' format='JSX.DataReader'>
 <object id='i0' class='CustomEg3'>
  <declaredClass class='CustomEg3'>
   <default>
    <primitive field='weeks' type='int' value='3'/>
    <primitive field='remainder_days' type='int' value='1'/>
   </default>
  </declaredClass>
 </object>
</jsx>
```

A downside of PutField and GetField is that you have to explicitly write the mapping for each field, and lose the automatic operation of the default methods. This can be unfortunate if you only need PutField or GetField for one field out of many, because you will need to use it for each of them. Finally, because PutField and GetField play the same role as defaultWriteObject and defaultReadObject, they cannot be used together, or else Jsx will expect to see the same data twice.

full GetField API:

SERIALPERSISTENTFIELDS

The serialPersistentFields information is a field which lists the serializable fields of the class. As it works per "slice", it is private. When used with `defaultWriteObject` and `defaultReadObject` (either when called explicitly or by default), it acts as an alternative to the transient keyword: by not listing a field, you prevent it from being serialized or deserialized. Any mismatch of fieldnames and/or types, causes an exception to be thrown at runtime.

When used with `PutField`, it does not need to correspond to actual fields in the class, but the field names and types `put` need to correspond to fields listed in it, matching in name and type.
Unlike the previous use, not all of them need be written. For `GetField`, fields in the stream can still be read even if they are not listed – but if they are listed, the types must match.

The actual `serialPersistentFields` itself is an array of `ObjectStreamField` objects, for example:

```
   private static final ObjectStreamField[]
serialPersistentFields = {
     new ObjectStreamField("weeks", int.class),
     new ObjectStreamField("remainder_days", int.class)
   };
```

Note the modifiers, `private static final`; and the type `ObjectStreamField`.

WRITEEXTERNAL AND READEXTERNAL

The writeExternal and readExternal are defined in the `Externalizable` interface, and must be public. Unlike many other serialization methods, these follow conventional object oritented rules, as they operate at the "whole" object level, and so the methods are public and can be inherited. On writing, writeExternal is called with one ObjectOutput argument – in fact, the object passed in the same as that passed in for writeObject and readObject, but the ObjectOutput restricts the allowable methods to be called (eg defaultWriteObject and the PutField methods are not available, as these are "slice" based).

On reading, the public no-argument constructor is invoked to create the object, and then its readExternal is called, with one ObjectInput argument – again, the object passed is actually the same as in readObject, but the ObjecctInput interface constricts the allowable methods (eg defaultReadObject cannot be called).

```
import JSX.*;
import java.io.*;
public class CustomEg5 implements Externalizable {
   int days = 22;
public CustomEg5() {
   }
   public void writeExternal(ObjectOutput oos) throws IOException
{
      oos.writeInt(days/7); // weeks
      oos.writeInt(days%7); // remainder days
   }
   public void readExternal(ObjectInput ois) throws IOException,
ClassNotFoundException {
      days = ois.readInt();
      days += ois.readInt();
   }
}
```

Notes:

1. **public no-argument constructor** - the no-argument constructor is explicitly "public" – the default no-argument constructor will be public only if the class itself is public; in other cases, the constructor must be explicitly given in order for externalization to be able to recreate the object.

2. `writeExternal` and `readExternal` are both **public**,

3. arguments are of interface type **`ObjectOutput`** and **`ObjectInput`**, thus restricting the API available.

Following is the resultant XML. Compare it to the XML from CustomEg1 above. Note that while data *content* is the same (the two ints), here it is for the "whole" object, instead of one "slice". Also note the lack of fieldnames.

```
<jsx major='1' minor='1' format='JSX.DataReader'>
 <object id='i0' class='CustomEg5'>
  <primitive type='int' value='3'/>
  <primitive type='int' value='1'/>
 </object>
</jsx>
```

WRITEREPLACE AND READRESOLVE

These two methods, `writeReplace` and `readResolve`, do not really customize the serial form of a class, but enable it to be represented by a different object altogether, possibly of a different class. They both nominate a replacement object, and so operate at the "whole" object level, rather like `writeExternal` and `readExternal`.

When writing, an object's `writeReplace` method is called before serializing the object, and the object it return is serialized instead. After deserializing an object, its `readResolve` is called and the object returned is placed into the object graph instead of the deserialized object. Because of this, the new object must be of a type that can be assigned to the fields that held the original object. For example, if an object of type A is held in a field of type A, you can't `readResolve` it to a String, because that cannot be assigned to a field of type A.

These methods do not have to be used together: a class can have `writeReplace` without a `readResolve`, and it can have a `readResolve` without a `writeReplace`.

Another use of `writeReplace` and `readResolve` is to process the data or to validate it, when writing and reading. For example, `writeReplace` could return the same object, but have prepared the data for serialization in some way; similarly, `readResolve` can do the same after reading. The same read post-processing effect can be achieved within a readObject method (defaultReadObject followed by the processing), and the write pre-processing in a writeObject method (the processing followed by `defaultWriteObject`).

Something very strange about `writeReplace` and `readResolve` is that they can have any access-modifier: private, protected, public or the default package-private access modifier. In other words, they can be inherited. If they are visible in a subclass (unless overridden, public and protected are, package-private is if in the same package, and private never is), they will be invoked. The result is that although the effect is on the "whole" object, as the whole object is replaced or resolved and not just one class slice, the method itself might reside in a superclass.

In the following example, the `writeReplace` takes us to the Replacement object; and when this Replacement object is read, it has a `readResolve`, which takes us back to the original object. It is a little like two node pointing at one another.

```
import JSX.*;
import java.io.*;
public class CustomEg6 {
   int days = 22;
   public CustomEg6() {}
   public CustomEg6(int days) {
      this.days = days;
   }
   public Object writeReplace() throws ObjectStreamException {
      return new Replacement(days);
   }
}

class Replacemen
   int weeks;
   int remainder_days;
   public Replacement(int days) {
      weeks = days/7;
      r
   }
   p  lic Object readResolve() throws Objec
      int days = weeks*7 + remainder
      r
```

And here is the resulting XML:

```
<jsx major='1' minor='1' format='JSX.DataReader'>
 <object id='i0' class='Replacement'>
  <declaredCl
   <default>
    <primitive field='weeks' type='int' value='3'/>
    <primitive
   </default>
  </declare
 </obje
```

One powerful feature of this approach is that the XML is completely separated from the specific class – it is not possible to tell from which class it came, or to which class it is resolved.

One downside is that this is a "whole" object technique, as mentioned. This means that a subclass's information would not automatically be represented in the replacement object, but must be handled specifically. The downside is that this subclass would also have to handle the superclass's information explicitly, as it must deal with the "whole" object and not just a class "slice".

EVOLUTION

One of the main reasons for customizing a class's serial form is for evolution. For example, if a class has evolved, but still needs to read in data from the old class. It is possible to change a class so that the changes are "backwards compatible" in this way, by not changing fields but only adding more information.

A more sophisticated approach is to separate the serial form from the class itself, with a layer between the two that maps from one to the other. In this way, when the class evolves, only the mapping need to be changed. However, if information is added in this course of evolution, then the serial form will also need to have information added to it, so that it can be retrieved.

Evolution can occur through changes to fields, classes and inheritance:

1. **field** changes
    a. meaning (semantics)
    b. name
    c. type
    d. addition
    e. deletion
2. **class** changes
    a. name
    b. addition
    c. deletion
3. inheritance changes
    a. different superclasses

For example, several "refactorings" affect the data model of classes in this way.

There are standard ways to cope with some of these changes: if a field has been deleted, then extra information upon deserialization is simply discarded. If a field has been added (and so the information is not present in the serialized data), it is set to the default value for its type (ie false, 0 or null). Type changes to fields are coped with gracefully, provided they are assignment compatible with the actual value (note that JOS does this for objects, but not for primitive types, which must match exactly). If a class's name changes, JSX will not be able to find it, and will throw a ClassNotFoundException. If the inheritance hierarchy of a class is reordered, JSX will still cope, but there is the risk of subclass readObject methods etc being executed before all superclasses are initialized.

BACKWARD COMPATIBILITY

As mentioned, backwards compatibility can be maintained very simply by only ever adding extra fields, and never changing the existing fields (or their meaning). The only complication is that when old instances are deserialized, these new values will not be set, but take on their default values – eg 0 for ints, and null for objects. There needs to be some way to distinguish this from an actual value of 0 or null. If the field could never naturally have such a value, then this is quite simple – your code need only test for zero or null, and initialize it appropriately. Things could get complicated if the class has passed through several versions, and the fields need to be initialized in different ways depending on the version.

The `GetField` class addresses the default issue with a `boolean defaulted(String fieldname)` method to tell you if the field has a value, or has merely taken on the default value. However, you will still need to track versions explicitly if they need to be initialized differently or if fields change their meanings.

The `writeReplace` and `readResolve` methods suggest an interesting alternative, by having a different representation class for each version. A problem with this solution is that the methods operate on the whole object rather than a slice.

FORWARD COMPATIBILTY

Forward compatibility is when old versions of the class can read data written by the new version – the opposite of backward compatibility. This can occur, for example, when different versions of a class need to communicate back and forth over the network.

For this to work, the only allowable changes is information adding – for if information is removed from a later version, or changed, then it will not be available for the old version that expects it.

A MODEST PROPOSAL: THE MEMENTO

The memento concept is not yet implemented in Jsx (and is not part of JOS), but will be implemented in due course.

Its operation is similar to the example given above for `writeReplace` and `readResolve`, but operates at the "slice" level rather the "whole" object level – in this respect, it is also similar to `PutField` and `GetField`.

It works by representing each "slice" of an object with another object, which also contains the code for mapping to and from it. Unlike readResolve, the mapping from XML to object does not involve an object being created, but only the filling in of a slice. The mapping to the object may be done via public methods, or directly to the fields if the memento has that access (one way to provide this is for the memento to be an inner class of the target object).

Memento will be written in the same format exactly as a class (so may be backward and forwards compatible with this), even though it is an object. The name of the memento class therefore serves as the name of that version of the serial format.

Mementos can be extended by inheritance, but when this happens, all the fields will be treated as if they were at the one level, instead of being sliced themselves. This means that shadowing will not work with mementos. Inherited mementos will need to chain to their inherited mapping methods, in order for all fields to be set correctly.

The memento may provide a very flexible mapping layer between objet and serial form. For example:

mapping between an old field and new field in the one class is straight-forward

mapping to a new class altogether cannot be done, because that is an operation at the "whole" object level, rather than the "slice" level.

mapping to different superclasses is automatic – the field data will be mapped to the appropriate class even if the order has changed. However, it may break the rule that superclasses are initialized before subclasses, and so any code run during deserialization that that depend on superclass being already initialized may fail.

In summary, it is difficult to provide power and safety at the same time. The best course is to restrict freedom, in such a way that there is enough power to do everything (perhaps with more complication), but to always be safe and robust. This is the java approach: sacrifice naked power for robust power.

Another approach is to delegate explicitly, within Jsx mechanism itself, in a way that is totally transparent to the class itself, and so we have complete power over the data as a whole, do not need to have access to the original source nor recompile nor risk breaking it. We can still use

```
public class Memento {
   int x;
   int y;
   public Memento(A a) {
      x = a.i;
      y = a.j;
   }
   public void mapInternal(A a) {
      a.i = x
      a.j = y;
   }
}
```

XML AND XSLT

One of the main uses of XSLT is to transform XML from one format into another. Because JSX writes XML, this opens up the opportunity of transforming object data, as a means of coping with evolution of classes (and upgrading information between application versions) and also as a means of loosely coupling different components together – regardless of the classes at either end, XSLT can transform between them. Taking it a step further, XSLT can be used to glue JSX output into arbitrary XML formats that you might need to work with already.

In other words, XSLT can be used as a kind of universal glue – and JSX is a means of preparing and cleaning the surfaces so they are as easy to glue as possible. The two surfaces can be old and new versions of a class, in the case of evolution; or two completely different classes, in the case of component integration.

EVOLUTION

Classes can evolve is some very complex ways, that involve fields migrating to different classes, inheritance hierarchies being overturned and the meanings of constellations of fields changing, requiring that their values be transformed. In the extreme, evolution is no difference from integration – it is just that usually, evolutionary changes are quite small, and it is these that we look at here.

The two most common and simple evolutions are the change of a fieldname, and the change of a classname.

The first step in writing an XSLT stylesheet for transforming XML from JSX, is to understand the grammar of JSX's format. Basically, each object is represented with an <object> element, which contains a <declaredClass> element for each class comprising it. This in turn contains the fields within a <default> element, and can be other objects, primitives, arrays, nulls and references. For example:

```
import JSX.*;
import java.io.*;
public class Evolve1 {
   Object[] array = {"red", "green", "blue", null, new Datum()};
   int days = 22;
}

class Datum {
   String name = "fred";
   int age = 24;
}
```

The objects created by this class are represented by the following XML:

```
<jsx major='1' minor='1' format='JSX.DataReader'>
 <object id='i0' class='Evolve1'>
  <declaredClass class='Evolve1'>
   <default>
    <array field='array' id='i1' base='java.lang.Object' dim='1'
length='5'>
     <string id='i2' value='red'/>
     <string id='i3' value='green'/>
     <string id='i4' value='blue'/>
     <null/>
     <object id='i5' class='Datum'>
      <declaredClass class='Datum'>
       <default>
        <string field='name' id='i6' value='fred'/>
        <primitive field='age' type='int' value='24'/>
       </default>
      </declaredClass>
     </object>
    </array>
    <primitive field='days' type='int' value='22'/>
   </default>
  </declaredClass>
 </object>
</jsx>
```

FIELD NAME EVOLUTION

For our first transformation, of fieldnames, it is important to note that every object that can have a fieldname uses the attribute "field" for it – this is true for objects, arrays, primitives, strings and so on. So our XSLT stylesheet will need to recognize this attribute and transform it.

The second thing is we need to take the class into account, for different classes can have fields of the same name.

The third thing is that the declaredClass element has the classname in a `class` attribute, and contains a default element, which contains one of several elements, all of which have the fieldname in a `field` attribute.

For example, the path to the fieldname is therefore must start from a `declaredClass` element, with a `class` of 'Evolve1', through a `default` element, through some other element (which could be object, array, primitive etc), and finally to a `field` attribute, with a value of 'array':

```
declaredClass[@class='classname']/default/*/@field[.='fieldname']
```

The "." stands for the current node, which is the field attribute in this case. Upon detecting this attribute, we need to write it out with its new value, like so:

```
   <xsl:attribute name="field">newfieldname</xsl:attribute>
```

It is important to realize how XSLT works: it first creates a fully formed representation of the XML in memory (as a DOM tree), and so there is no need to worry about the order that the XML is processed in, before all the information is available at the same time. You can think of this as a tree diagram of the XML document.

XSLT enables you to write templates that will match various parts of the tree – and you can specify what it writes out as it encounters each match. What we are doing here is matching the desired fields, and writing out the new field information.

But of course, we also want the rest of the XML to be written as well – for this we use an identity transformation, which simply traverses the entire tree, writing it out as we it goes along. This is based on very general templates, and the rule for XSLT processing is that it will prefer a more specific match if one is available – which is why our field matches are followed, when they do match. Here is the "identity transformation":

```
<xsl:template match="*|@*">
   <xsl:copy>
      <xsl:apply-templates select="*|@*"/"
   </xsl:copy>
</xsl:template>
```

The "*" means to match any element but for an attribute – the "@* means to match any attribute. The "apply-templates" is like a method call, that continues the recursion. Finally, the xsl is the conventional abbreviation of the full namespace for XSLT, which is "http://www.w3.org/1999/XSL/Transform".

There needs to be some preliminary guff to make all this work, which is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <xsl:output method="xml" version="1.0" encoding="UTF-8"
indent="yes"/>
```

To test this, let's change several fields at once., based on the above example XML:

1. rename the "array" field of Evolve1, to "things"
2. rename the "age" field of Datum, to "years"
3. rename the "name" field of Datum, to "firstName"

This leads to a complete document of:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<xsl:stylesheet version=”1.0”
   xmlns:xsl=“http://www.w3.org/1999/XSL/Transform”>
   <xsl:output method=”xml” version=”1.0” encoding=”UTF-8”
indent=”yes”/>

   <xsl:template match=”*|@*”>
      <xsl:copy>
         <xsl:apply-templates select=”*|@*”/>
      </xsl:copy>
   </xsl:template>

   <xsl:template
match="declaredClass[@class='Evolve1']/default/*/@field[.='array'
]">
      <xsl:attribute name=”field”>things</xsl:attribute>
   </xsl:template>

   <xsl:template
match="declaredClass[@class='Datum']/default/*/@field[.='age']">
      <xsl:attribute name=”field”>years</xsl:attribute>
   </xsl:template>

   <xsl:template
match="declaredClass[@class='Datum']/default/*/@field[.='name']">
      <xsl:attribute name=”field”>firstName</xsl:attribute>
   </xsl:template>

</xsl:stylesheet>
```

When applied to the previously given example, this XML results:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsx major="1" minor="1" format="JSX.DataReader">
   <object id="i0" class="Evolve1">
      <declaredClass class="Evolve1">
         <default>
            <array field="things" id="i1" base="java.lang.Object"
dim="1" length="5">
            <string id="i2" value="red"/>
            <string id="i3" value="green"/>
            <string id="i4" value="blue"/>
            <null/>
            <object id="i5" class="Datum">
               <declaredClass class="Datum">
                  <default>
                     <string field="firstName" id="i6"
value="fred"/>
                     <primitive field="years" type="int"
value="24"/>
                     </default>
                  </declaredClass>
               </object>
            </array>
            <primitive field="days" type="int" value="22"/>
         </default>
      </declaredClass>
   </object>
</jsx>
```

Note: as at the time of writing, the above XML cannot be deserialized by JSX, because the XML parser only recognizes apostrophes (‘), and not quotes (“) for attribute values. Secondly, of course, new fields of these names need to be created within the relevant classes, or they have no where to be read into, and so will be discarded.

CLASS NAME EVOLUTION

The other simple transformation we mentioned was changing of a class name. By this we mean that it is exactly the same class, but only the name has changed. Class names appear in both the object element and the declared class element, and so we need to change it in both places. But as the attribute class only ever occurs in those two places, we can ignore the element name, and use a very simple path to match on:

```
@class[.='classname']
```

As in field name evolution, the “.” stands for the current node, which is the field attribute in this case. Upon detecting this attribute, we need to write it out with its new value, like so:

```
   <xsl:attribute name=”class”>newclassname</xsl:attribute>
```

For example, we could change the name of the “Datum” class in the above XML, to “Datum2”, by adding this to our stylesheet:

```
   <xsl:template match="@class[.='Datum']">
      <xsl:attribute name="class">Datum2</xsl:attribute>
   </xsl:template>
```

INTEGRATION

As we suggested earlier, the distinction between evolution and integration is historical, in that if two classes differ because one evolved for the other, it is "evolution"; but if those same two classes came about by independent development in different components, then it is "integration". In fact, both are integrations of classes with different definitions, and that is all.

Accordingly, we can provide an example of integration by reusing the XSLT stylesheet developed above, by simply creating a Datum2 class, with years and firstName fields:

```
public class Datum2 {
   int years;
   String firstName;
}
```

Of course, because integration is usually between unrelated classes, there will be usually a greater gap to cross than for evolution. Very likely, the values of the fields will also need to be changed as well as the names. Although this can be done within XSLT, it can be awkward as XSLT was not designed as a general purpose language, and it is probably much easier to handle this translation within java itself, as in the previous chapter on "customizing an object's serial form".