The syntax parser

1. The syntax parser

JaxMeXS mainly consists of three parts: A generic parser, which is by no means restricted to XML schema, a syntax parser, which is is dedicated to the syntactical aspects of XML schema, and a structure parser, which understands the logic. Extending JaxMeXS will always imply extending the syntax parser. In the most cases this will even be sufficient: Who's interested in creating a language as complex as XML schema?

The syntax parser is an application of the generic parser. This means, that it converts any element in the XML schema into a Java bean. The attributes and elements are mapped to bean properties. Attributes have simple values like strings or integers, but a child element is yet another bean. In what follows, we'll discuss the following aspects:

1. Using the syntax parser.
2. Making the schema parser to use your own beans.
3. Adding new attributes to an existing schema element.
4. Handling different namespaces.
5. Adding new child elements.

2. Using the syntax parser

To use the syntax parser, instantiate the class XSParser and invoke its `parse()` method, for example like this:

```
import java.io.FileInputStream;
import org.xml.sax.InputSource;
import org.apache.ws.jaxme.xs.XSParser:

public class MyParser {
  public static void main(String[] args) throws Exception {
    FileInputStream fs = new FileInputStream(args[0]);
    InputSource is = new InputSource(fs);
    is.setSystemId(fs.toURL().toString()); // This is important, if you use a DTD,
                                           // external entities, schema validation, .
                                           // in other words: Other files
    XSParser parser = new XSParser();
    XsESchema schema = parser.parseSyntax(is);
  }
}
```

That's it! The returned instance of XsESchema is a standard schema with methods like `getTargetNamespace()` or `getChilds()`. For example, the list of global types can be retrieved as follows:

```
Object[] childs = schema.getChilds();
for (int i = 0;  i < childs.length;  i++) {
  Object o = childs[i];
  if (o instanceof XsETopLevelSimpleType) {
    XsETopLevelSimpleType t = (XsETopLevelSimpleType) o;
    System.out.println("Simple type: " + t.getName());
  } else if (o instanceof XsTComplexType) {
    XsTComplexType t = (XsTComplexType) o;
    System.out.println("Complex type: " + t.getName());
  }
}
```

This is not very comfortable, but the intent of the syntax parser is simplicity, and not comfort.

3. Forcing the schema parser to use your own beans

The beans returned by the schema parser are instances of XsObjectImpl, implementing the interface XsObject. The `XsObject` interface allows access to the SAX location.

However, you might replace these completely with own implementations: The object factory makes it possible.

Any of the standard XML schema beans is created by the object factory. For example, the method `newXSESchema()` is invoked to create the schema bean. We take this bean as an example and change the behaviour of its attribute `targetNamespace`. For compatibility reasons we want to replace the namespace `http://company.com/namespaces/version1` with `http://company.com/namespaces/version2`. To achieve that, we have to create three classes:

1. A subclass of XsESchemaImpl with a modified `setTargetNamespace()` method,
2. an updated object factory, that doesn't create an instance of the base class, but an instance of our subclass,
3. and, finally, a parser that uses our own object factory.

Let's begin with the first task:

```
import org.xml.sax.Locator;
import org.apache.ws.jaxme.xs.xml.XsObjectFactory;
import org.apache.ws.jaxme.xs.xml.impl.XsESchemaImpl;
```

```
public class MySchema extends XsESchemaImpl {
  protected MySchema(XsObjectFactory pFactory, Locator pLocator) {
    super(pFactory, pLocator);
  }
  public void setTargetNamespace(XsAnyURI pURI) {
    if (pURI.equals("http://company.com/namespaces/version1")) {
      pURI = "http://company.com/namespaces/version2";
    }
    super.setTargetNamespace(pURI);
  }
}
```

Neat, isn't it? Now, here's the object factory:

```
import org.apache.ws.jaxme.xs.xml.XsESchema;
import org.apache.ws.jaxme.xs.xml.impl.XsObjectFactoryImpl;

public class MyXsObjectFactory extends XsObjectFactoryImpl {
  public XsESchema newXsESchema() {
    return new MySchema(this, getLocator());
  }
}
```

You probably already guess how the parser looks like:

```
import org.apache.ws.jaxme.xs.XSParser;

public class MyParser extends XSParser {
  public MyParser() {
    getData().setXsObjectFactory(new MyXsObjectFactory());
  }
}
```

Now we have a parser, which does no longer distinguish between `http://company.com/namespaces/version2` and `http://company.com/namespaces/version1` in the target namespace.

4. Adding new attributes to an existing schema element.

We already know how to extend the parser.This knowledge will be applied in the following example: We'll have an an additional attribute "ignore" in the element definition. It ought to have a boolean value. (For example, the attribute might indicate that a custom program should ignore the element.)

The attribute is introduced by adding a property "ignore" to the "element" bean. This might look like this:

```
import org.xml.sax.Locator;
import org.apache.ws.jaxme.xs.xml.XsObjectFactory;
import org.apache.ws.jaxme.xs.xml.impl.XsTElementImpl;

public class MyElement extends XsTElementImpl {
  private boolean ignoreMe;

  protected MyElement(XsObjectFactory pFactory, Locator pLocator) {
    super(pFactory, pLocator);
  }
  public void setIgnore(boolean pIgnore) {
    ignoreMe = pIgnore;
  }
  public boolean getIgnore() {
    return ignoreMe;
  }
}
```

We are not yet done. As we are using an own bean, we have to extend the object factory to return this bean, if the method `newXsTElementImpl()` is invoked. We also have to extend the parser to use the extended object factory. These steps have already been described in the previous section, so we omit it here.

5. Handling different namespaces

What we did in the previous section on adding attributes, wasn't really conforming to XML Schema. Our attribute had the default namespace, as the standard XML Schema attributes do. Any other namespace had been a better choice. XML Schema allows to include arbitrary attributes into a schema, as long as they do not have the XML schema namespace. Surprisingly, the default namespace isn't implicitly forbidden. Anyways, such behaviour cannot be recommended.

To support attributes from other namespaces, we'll have to add another method to our bean. The method is called

```
public boolean setAttribute(String pQName, String pNamespaceURI,
                            String pLocalName, String pValue)
  throws SAXException;
```

The boolean return value allows the method a decision to handle an attribute (for example, if it is defined in a particular additional namespace) by returning `true` or to leave the attribute to the standard mechanisms by returning `false`. A typical implementation might look like this:

```
  if (!"http://company.com/namespaces/mynamespace".equals(pNamespaceURI)) {
    return false;
```

```
    }
    if ("ignore".equals(pLocalName)) {
      setIgnore(Boolean.valueOf(pValue).booleanValue());
    } else {
      throw new SaxParseException("Invalid attribute: " + pValue, getLocation());
    }
```

The meaning is obvious: We feel responsible for the namespace `http://company.com/namespaces/mynamespace`. If the attributes namespace is different, we simply return false. If the namespace matches, we accept the attribute "ignore", and refuse all others by throwing a `SAXException`.

6. Adding new child elements

The handling of a new child is no more complex than the handling of attributes. In fact, it works quite the same. Basically one creates a new bean and adds a bean property to the parent element, as in the following example:

```
MyChildBean childBean;

public MyChildBean createMyChild() {
  if (childBean != null) {
    throw new IllegalStateException("Multiple 'myChild' elements are forbidden.");
  }
  childBean = new MyChildBean();
}

public MyChildBean getMyChild() {
  return childBean;
}
```

This code is added to the parent bean. For example, if we want to have a new element `xs:schema/xs:myChild`, we could create a new subclass of XsESchemaImpl with the above code. By extending the object factory to use our updated schema bean and extending the parser to use our private object factory, we would be done. (The latter steps are as in the first example section on using our own beans.)

There are two possible reasons, why the above code might be insufficient: First of all, the example obviously doesn't care for namespaces. Second, there's a chance that we do not want to create a simple bean. For example, the standard behaviour of XsEAppinfo is to convert child elements into DOM documents.

Both becomes possible by the following example:

```
import org.xml.sax.ContentHandler;
import org.apache.ws.jaxme.xs.parser.impl.XsSAXParserImpl;
```

```
MyChildBean childBean;

public ContentHandler getChildHandler(String pQName, String pNamespaceURI,
                                      String pLocalName) throws SAXException {
  if (!"http://company.com/namespaces/mynamespace".equals(pNamespaceURI)) {
    return null;
  }
  if ("myChild".equals(pLocalName)) {
        if (childBean != null) {
          throw new IllegalStateException("Multiple 'myChild' child elements are forb
        }
        childBean = new MyChildBean();
        return new XsSAXParserImpl(childBean);
  } else {
    throw new IllegalStateException("Unknown child element: " + pQName);
  }
}

public MyChildBean getMyChild() {
  return childBean;
}
```

Besides the different namespace, the example is functionally equivalent to the previous example.