In diesem Kapitel
- Einleitung
- Parsing
- Callback Interfaces
  - *ContentHandler*
  - *DefaultHandler Adapter*
- Verarbeiten mehrerer XML Dokumente
- Methoden
  - *characters*
  - *Attribute verarbeiten*
  - *Processing Instructions*
  - *Namespace Mappings*
  - *Ignorable Whitespaces*
  - *Locators*
- Einiges fehlt noch in heutigen Parsern
- Zusammenfassung

6. *SAX Parser*

6.1. Einleitung

Der SAX Parser besteht im Wesentlichen aus zwei Teilen:
- dem `XMLReader` Interface, dem eigentlichen Parser Interface und
- dem `ContentHandler`, der die Callbacks des SAX Parsers verarbeiten muss.

Im Folgenden beschreiben wir einige Features des Content Handlers und Grundoperationen der Readers. Im nächsten Kapitel werden wir uns dann den Reader genauer anschauen.

6.2. SAX Hintergrundinformationen

Peter Murry-Rust definierte das SAX Interface um 1997/1998 als leichte Alternative zu den riesigen API's des W3C's. Der eigentliche Parser wurde von Tim Bray und David Megginson gestartet und als offenes Projekt mit vielen Beiträgen „aus dem Publikum" realisiert.
SAX 1.0 wurde am 11. Mai 1998 veröffentlicht.

Grundsätzlich existiert SAX unabhängig vom konkreten Parser. SAX definiert Interfaces, welche mithilfe existierender Parser umgesetzt werden können.

SAX ist ein einfaches, effizientes XML API. DOM ist sicher viel umfassender, aber auch viel komplexer.

SAX steht in vielen / für viele Programmiersprachen zur Verfügung:
- Perl
- C++
- Visual Basic
- ...

Obschon SAX kein Standard ist, wenigstens nicht nach W3C oder einem andern Gremium, ist SAX ein de facto Standard, Open Source und wird von vielen Firmen unterstützt. Es besteht kein Copyright darauf, auch kein Patent.

Um 1999 begann man mit SAX 2, einer völlig neuen Implementierung und Umformulierung. Fast alle Klassen werden überarbeitet. Die Event-Idee bleibt aber der Kern von SAX.

Neue Features von SAX 2 (Mai 2000):
- Namensräume erkennen und verarbeiten
- DTD
- Filter
- Lexikalische Events

Heute (2003) unterstützen die meisten Parser SAX 2, als das nun wohl vollständigste API.

6.3. Parsing

Beim Parsen wird ein XML Dokument gelesen, auf wohlgeformtheit überprüft und an einen Client weiter geleitet. In SAX wird der Parser mithilfe einer Instanz des `XMLReader` Interfaces realisiert. Je nach Implementierung handelt es sich jeweils um eine andere Klasse:
- in Xerces: `org.apache.xerces.parsers.SAXParser`
- in Crimson: `org.apache.crimson.parser.XMLReaderImpl`

Im Anwendungsprogramm kreieren Sie typischerweise eine Instanz dieser Klasse mithilfe eine statische Factory Methode:
- `XMLReaderFactory.createXMLReader()`

Als nächstes wird ein `InputSource` Objekt, welches beispielsweise mit einem Input Stream verknüpft ist, an den Parser übergeben.

Der Parser liest das Dokument und wirft eine Exception, falls das Dokument nicht wohlgeformt ist.

Das folgende Beispiel zeigt, wie leicht die Wohlgeformtheit überprüft werden kann:
- kreieren eines `XMLReaders` / Parsers
- kreieren lesen des XML
- auswerten der Exceptions (falls keine auftritt ist das Dokument wohlgeformt).

Das Programm enthält zwei Lesevarianten:
- einmal mithilfe einer `InputSource` und einem `InputStream`.
  In diesem Fall wird überprüft, ob die XML Datei wirklich existiert.
- einmal direktes Lesen der Datei durch den Parser (`XMLReader`).
  In diesem Fall wird *nicht* überprüft, ob die XML Datei existiert.
  Falls die Datei nicht vorhanden ist, wird der Parser keine Exception werfen.
  Das (nicht vorhandene) XML Dokument ist also wohlgeformt!

Sie müssen also etwas vorsichtig mit den Exceptions und dem gesamten Handling von SAX umgehen.

```
import java.io.IOException;

import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

/**
 * @author jjoller
 *
 * Im folgenden wird ein XML Dokument gelesen.
 * Falls das Dokument wohlgeformt ist,
 * wird keine Exception geworfen
 * sonst wird die SAXException geworfen.
 */
public class SAXWohlgeformt {

  public static void main(String[] args) {
      try {
            String xml_doc = "XMLEingabe.xml";
            /*
             * Xerces
             */
            //System.setProperty("org.xml.sax.driver",
                  "org.apache.xerces.parsers.SAXParser");
             /*
              * Crimson
              */
            System.setProperty("org.xml.sax.driver",
                  "org.apache.crimson.parser.XMLReaderImpl");
            XMLReader parser =
                        XMLReaderFactory.createXMLReader();
            InputSource is = new InputSource();
            is.setByteStream(new FileInputStream(xml_doc));
            // URI
            //parser.parse("XMLEingabe.xml");
            parser.parse(is);
            System.out.println("Das Dokument "+xml_doc+" ist
                                                wohlgeformt");
      } catch (FileNotFoundException e) {
            System.err.println("Datei kann nicht gefunden
                                                      werden");
            e.printStackTrace();
      } catch (SAXException e) {
            System.err.println("Datei ist nicht wohlgeformt");
            e.printStackTrace();
      } catch (IOException e) {
            System.err.println("Genereller IO Fehler");
            e.printStackTrace();
      }
  }
}
```

Falls Sie die System Eigenschaft (`org.xml.sax.driver`) nicht setzen und die Klasse nicht voll qualifiziert verwenden (mit Package Name), wird der Parser nicht gefunden und eine entsprechende Fehlermeldung generiert:

```
org.xml.sax.SAXException: System property
org.xml.sax.driver not specified
      at org.xml.sax.helpers.XMLReaderFactory.createXMLReader(Unknown
Source)
```

Die Imports bleiben unverändert. Falls Sie Xerces aktivieren, müssen Sie die entsprechenden JAR Dateien in den CLASSPATH aufnehmen.

6.4. Callback Interfaces

SAX verwendet das Observer Design Pattern:

- der Parser teilt mit Callbacks dem Client mit, was im XML Dokument gefunden wird.

Im Client Programm (in den Callback Methoden) steckt die eigentliche Bearbeitung:

- Der Client implementiert ein Interface, beispielsweise einen Listener und registriert diesen bei einem Widget.
- Falls das zum Listener gehörende Event ausgelöst wird, wird die Listener Methode im Client aufgerufen.

Die einzelnen Rollen:

- Das Widget ist das Subjekt,
- das Listener Interface ist der Observer,
- die von Ihnen implementierte Methode ist der konkrete Observer.
-

Bei SAX werden die Tags analysiert und je nach Tag ein Event ausgelöst:

- der `XMLReader` ist das Subjekt
- das Interface `org.xml.sax.ContentHandler` ist der Observer

Ein Unterschied besteht jedoch zwischen diesen beiden Observer Pattern Implementierungen:

- im Falle der Widgets können mehrere Observer (Listener) registriert werden
- im Falle von SAX kann nur ein Handler angegeben werden. Im Handler selber verstecken sich aber viele Methoden, die genutzt werden können.

```
package org.xml.sax;

public interface ContentHandler {

  public void setDocumentLocator(Locator locator);
  public void startDocument() throws SAXException;
  public void endDocument() throws SAXException;
  public void startPrefixMapping(String prefix, String uri)
   throws SAXException;
  public void endPrefixMapping(String prefix)
   throws SAXException;
  public void startElement(String namespaceURI, String localName,
   String qualifiedName, Attributes atts) throws SAXException;
  public void endElement(String namespaceURI, String localName,
   String qualifiedName) throws SAXException;
  public void characters(char[] text, int start, int length)
   throws SAXException;
  public void ignorableWhitespace(char[] text, int start,
   int length) throws SAXException;
  public void processingInstruction(String target, String data)
   throws SAXException;
  public void skippedEntity(String name)
   throws SAXException;

}
```

Beim Lesen des XML Dokuments startet der Parser (`XMLReader`) diese Methoden, jeweils an der passenden Stelle:

- beim Lesen des Dokumentanfangs wird die Methode `startDocument()` gestartet
- beim finden eines Starttags (irgend eines Start Tags) wird die Methode `startElement()` gestartet. Sie müssen in der Methode abfragen, um welchen konkrete Tag es sich dabei handelt.
- falls der Tag einen Inhalt (in Form eines Textes: alles ist Text, auch int, boolean, ...) findet, wird die Methode `characters()` gestartet.

Die Methoden werden in der Reihenfolge des Auftretens der Tags im Dokument ausgelöst.

6.4.1. Implementieren eines `ContentHandler`

Als erstes bauen wir uns einen Handler, welcher den in den Tags vorhandenen Text ausgibt. Die relevante Methode ist in diesem Fall `characters()`.

```
package saxparser;

import java.io.IOException;
import java.io.Writer;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;

/**
 * @author jjoller
 *
 * Einfacher SAX Handler
 * Achtung:
 * 1) Sie dürfen den Writer nicht im Handler schliessen; er wird laufend
benötigt
 * 2) auch Leerzeichen werden als Text angesehen und ausgegeben
 * Sie können ihn sehr leicht in Eclipse generieren lassen:
 * Source->Override/Implement Methods generiert alle Methoden der
Oberklasse

 */

public class SAXCrimsonCDATAHandler implements ContentHandler {
      private Writer out;

      public SAXCrimsonCDATAHandler(Writer wout){
            this.out = wout;
      }
      public void characters(char[] ch, int start, int length)
            throws SAXException {
            try {
                  out.write(ch, start, length);
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
      public void endDocument() throws SAXException {
      }
      public void endElement(String namespaceURI, String localName,
            String qName)throws SAXException {
      }
```

```
public void endPrefixMapping(String prefix) throws SAXException {
}
public void ignorableWhitespace(char[] ch, int start, int length)
      throws SAXException {
}
public void processingInstruction(String target, String data)
      throws SAXException {
}
public void setDocumentLocator(Locator locator) {
}
public void skippedEntity(String name) throws SAXException {
}
public void startDocument() throws SAXException {
}
public void startElement(
      String namespaceURI,
      String localName,
      String qName,
      Attributes atts)
      throws SAXException {
}
public void startPrefixMapping(String prefix, String uri)
      throws SAXException {
}
```

In diesem Handler verwenden wir lediglich die vordefinierten Methoden. Sie könnten aber auch weitere, eigene Methoden (wie üblich) selber definieren und hinzufügen.

Die benötigte Methode ist characters(). Diese wird vom SAX Parser aufgerufen, wenn er auf Nicht-Tag-Informationen trifft, also beispielsweise ‚\n' und Leerzeichen, neben dem eigentlichen Inhalt der Tags:

```
<Tag>
      <NochEinTag>Taginhalt</NochEinTag>
          </Tag>
```

hat neben dem eigentlichen Taginhalt "`Taginhalt`" noch weitere `characters()` :
- einmal „`\n` „
- einmal „\n „ .

Wir werden noch sehen, wie sich dieses Problem einfach lösen lässt.

Falls Sie die `IOException` in der `characters()` Methode etwas besser weiter geben wollen, dann werfen Sie in dieser einfach noch die `SAXException()`, damit der Parser eine etwas sinnvollere Fehlermeldung produzieren kann:

```
public void characters(char[] ch, int start, int length)
      throws SAXException {
      try {
            out.write(ch, start, length);
      } catch (IOException e) {
            throw new SAXException(e);
      }
}
```

6.4.2. Einsatz des `ContentHandler`

Den Handler müssen wir jetzt noch beim Parser registrieren (wir können genau einen Handler registrieren). Das Testprogramm für den Handler finden Sie im Programm **`public class`** `SAXCrimsonCDATA.`

```
package saxparser;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;

import org.xml.sax.ContentHandler;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.XMLReaderFactory;

/**
 * @author jjoller
* Testprogramm für den SAX Handler
 */
public class SAXCrimsonCDATA {
      public static void main(String[] args) throws Exception  {
            // Lesen des XML Dokuments
            try {
                  String xml_doc = "Test.xml";
                  /* * Xerces */
                  //System.setProperty("org.xml.sax.driver",
                  "org.apache.xerces.parsers.SAXParser");
                   /* * Crimson */
                  System.setProperty("org.xml.sax.driver",
                        "org.apache.crimson.parser.XMLReaderImpl");
                  XMLReader parser = XMLReaderFactory.createXMLReader();
                  // Content Handler
                  Writer out = new OutputStreamWriter(System.out);
                  ContentHandler handler
                   = new SAXCrimsonCDATAHandler(out);
                  parser.setContentHandler(handler);

                  InputStream in = new FileInputStream(xml_doc);
                  InputSource source = new InputSource(in);
                  parser.parse(xml_doc); // oder (source);
                  System.out.println("Das Dokument "+xml_doc+
                                    " ist wohlgeformt!");
                  out.flush();
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
            } catch (SAXException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

Der Aufbau ist fast selbsterklärend:

- zuerst müssen wir den SAX Driver festlegen, über die System Property

- dann bilden wir eine Instanz des `XMLReader`
- dazu verwenden wir die `XMLReaderFactory`
- der Parser benötigt einen Handler. Diesen instanzieren wir.
- Den `InputStream` und die `InputSource` benötigen wir nur, falls wir das Problem mit der leeren XML-Quelle abfangen wollen.

Der `Writer` wird im Hauptprogramm definiert und für die Ausgabe aller Tag-Inhalte benutzt. Vor Programmende müssen wir den `Writer` leeren.

Für die XML Datei

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <methodCall>
  <methodName>primeServer.getPrime</methodName>
- <params>
- <param>
- <value>
  <int>123</int>
    </value>
    </param>
- <param>
- <value>
  <boolean>1</boolean>
    </value>
    </param>
    </params>
    </methodCall>
```

ergibt sich die Ausgabe:

```
Das Dokument Test.xml ist wohlgeformt!
primeServer.getPrime 123 1
```

6.4.3. Die `DefaultHandler` Adapter Klasse

Wie auch im GUI Bereich existiert neben den Interfaces ein Handler als Klasse. Mit deeren Hilfe kann man sehr schnell eine eigene Klasse definieren, als Erweiterung des Adapters. In dieser Klasse müssen Sie dann nur die benötigten Methoden überschreiben.

```
package saxparser;

import java.io.IOException;
import java.io.Writer;

import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

/**
 * @author jjoller
 *
 */

public class SAXCrimsonCDATAHandlerAdapter     extends DefaultHandler {

      private Writer out;

      public SAXCrimsonCDATAHandlerAdapter(Writer wout){
            this.out = wout;
      }
      public void characters(char[] ch, int start, int length)
            throws SAXException {
            try {
                  out.write(ch, start, length);
            } catch (IOException e) {
                  throw new SAXException(e);
            }
      }
}
```

6.5. Empfangen von mehreren XML Dokumenten

Wenn Sie die einzelnen Callback Methoden anschauen, werden Sie sich evtl. fragen, warum es eine startDocument() Methode gibt.

Der Grund ist der, dass die Parser so ausgelegt sind, dass gleich mehrere Dokumente, eines nach dem andern, verarbeitet werden können. In diesem Fall ist es natürlich wichtig mitgeteilt zu bekommen, wann ein neues Dokument analysiert wird bzw. ein Dokument abgeschlossen ist.

Das folgende Beispiel (`ContentHandler`) speichert den Text jedes gelesenen XML Dokuments in einen `StringBuffer` und serialisiert diesen in eine Zeichenkette, welche in einer Liste gespeichert wird.

```
package saxparser;

import java.util.List;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
```

```
/**
 * @author jjoller
 *
 * Dieser Handler liest mehrere XML Docs in einen StringBuffer
 * und speichert diesen serialisiert als String in einer Liste.
 */
public class SAXCrimsonMultiDocsHandler implements ContentHandler {

      /*
       * Liste mit den XML Dokumenten als String (Objekte)
       */
      private List documents;
      private StringBuffer currentDocument;

      public SAXCrimsonMultiDocsHandler(List documents) {
        if (documents == null) {
            throw new NullPointerException(
             "Leeres Dokument - Analyse unmöglich");
        }
        this.documents = documents;
      }

      /**
       * ein neues Dokument wird eingelesen
       * und in einen (neuen) StringBuffer eingelesen
       */
      public void startDocument() {
        currentDocument = new StringBuffer();
      }

      /**
       * Ende eines XML DOkuments :
       * - hinzufügen des Dokuments als Zeichenkette in die
       *   Dokumenteliste
       */
      public void endDocument() {
        String text = currentDocument.toString();
        documents.add(text);
      }

      /**
       * extrahieren des Textes
       */
      public void characters(char[] text, int start, int length) {
        currentDocument.append(text, start, length);
      }

      // leere Hanlder Methoden (Interface)
      public void setDocumentLocator(Locator locator) {}
      public void startPrefixMapping(String prefix, String uri) {}
      public void endPrefixMapping(String prefix) {}
      public void startElement(String namespaceURI, String localName,
       String qualifiedName, Attributes atts) {}
      public void endElement(String namespaceURI, String localName,
       String qualifiedName) {}
      public void ignorableWhitespace(char[] text, int start,
       int length) {}
      public void processingInstruction(String target,
       String data) {}
      public void skippedEntity(String name) {}
}
```

Im Hauptprogramm müssen die Dokumente angegeben und die Liste definiert werden:

```
String[] xml_docs = { "Test.xml","XMLEingabe.xml" };
System.setProperty(
      "org.xml.sax.driver",
      "org.apache.crimson.parser.XMLReaderImpl");
XMLReader parser = XMLReaderFactory.createXMLReader();
// Content Handler ab Interface
List docListe = new Vector();
ContentHandler handler = new
            SAXCrimsonMultiDocsHandler(docListe);
parser.setContentHandler(handler);

InputStream in;
InputSource source;
for (int i = 0; i < xml_docs.length; i++) {
      in = new FileInputStream(xml_docs[i]);
      source = new InputSource(in);
      parser.parse(xml_docs[i]);
      System.out.println(
            "Das Dokument " + xml_docs[i] +
                              " ist wohlgeformt!");
}
for (int i=0; i<xml_docs.length; i++) {
      System.out.println("Dokument "+xml_docs[i]);
      System.out.println(docListe.get(i));
}
```

6.6. Verarbeiten der Elemente

SAX zeigt, wie das XML Dokument aufgebaut ist; aber es berichtet bloss über Tags, nicht über Elemente und deren Inhalt. Falls auf ein Starttag kein passender Endtag folgt, wird eine SAX Exception geworfen. Aber der Programmierer ist für die eigentliche Verarbeitung des XML Elements verantwortlich.

In vielen Fällen kommt man dabei auch ohne DOM Baum aus. Das folgende Beispiel zeigt, wie ein XML Dokument als Baum dargestellt werden kann. Dazu müssen wir zuerst besser verstehen, was eine `startElement()` bzw. eine `endElement()` Methode liefert:

```
public void startElement(String namespaceURI, String localName,
      String qualifiedName, Attributes atts) throws SAXException;

public void endElement(String namespaceURI, String localName,
      String qualifiedName) throws SAXException;
```

Der erste Parameter enthält die URI, falls das Element qualifiziert ist. Sonst enthält dieses Element eine leere Zeichenkette (nicht das `null` Element).

Der zweite Parameter enthält den Namen des Elements *ohne* den Namensraum-Präfix:

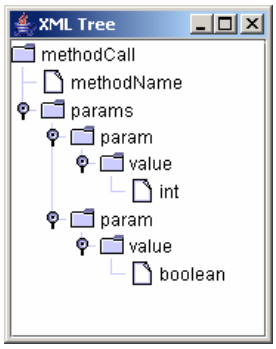
- falls das Element `SOAP-ENV:Envelope` ist
- dann wird `Envelope` übergeben

Der dritte Parameter enthält den voll qualifizierten Namen (im obigen Beispiel also `SOAP-ENV:Envelope`).

Das folgende Beispiel repräsentiert ein XML Dokument als `javax.swing.JTree`.

```
package saxparser;

import java.util.EmptyStackException;
import java.util.Stack;

import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.MutableTreeNode;
import javax.swing.tree.TreeNode;

import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

/**
 * @author jjoller
 * Einfache Baumausgabe eines XML Dokuments
 */
public class SAXCrimsonJTree extends DefaultHandler {

	private Stack nodes;

	/**
	 * startDocument() initialisiert pro XML Dokument einen Stack
	 * @throws SAXException
	 */
	public void startDocument() throws SAXException {
		nodes = new Stack();
	}

	/**
	 * Wurzelelement eines XML Dokuments
	 */
	private TreeNode root;

	/**
	 * Initialisierung pro neu eingelesenes Dokument
	 * @param namespaceURI
	 * @param localName
	 * @param qualifiedName
	 * @param atts
	 */
	public void startElement(
		String namespaceURI,
		String localName,
		String qualifiedName,
		Attributes atts) {

		String data;
		if (namespaceURI.equals(""))
			data = localName;
		else {
			data = '{' + namespaceURI + "} " + qualifiedName;
		}
		MutableTreeNode node = new DefaultMutableTreeNode(data);
```

```
			try {
				MutableTreeNode parent = (MutableTreeNode) nodes.peek();
				parent.insert(node, parent.getChildCount());
			} catch (EmptyStackException e) {
				root = node;
			}
			nodes.push(node);

		}
		/**
		 * Abschluss des XML Dokuments
		 * @param namespaceURI
		 * @param localName
		 * @param qualifiedName
		 */
		public void endElement(
			String namespaceURI,
			String localName,
			String qualifiedName) {
			nodes.pop();
		}

		/**
		 * Anzeige des Baumes pro Dokument
		 */
		public void endDocument() {

			JTree tree = new JTree(root);
			JScrollPane treeView = new JScrollPane(tree);
			JFrame f = new JFrame("XML Tree");

			f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
			f.getContentPane().add(treeView);
			f.pack();
			f.show();

		}

		public static void main(String[] args) {
			System.setProperty(
				"org.xml.sax.driver",
				"org.apache.crimson.parser.XMLReaderImpl");

			try {
				XMLReader parser =
					XMLReaderFactory.createXMLReader(
						"org.apache.crimson.parser.XMLReaderImpl");
				ContentHandler handler = new SAXCrimsonJTree();
				parser.setContentHandler(handler);
				String[] xml_docs = { "Test.xml", "XMLEingabe.xml" };
				for (int i = 0; i < xml_docs.length; i++) {
					parser.parse(xml_docs[i]);
				}

			} catch (Exception e) {
				System.err.println(e);
			}
		}
}
```



Die Ausgabe sieht je nach XML Dokument anders aus, beispielsweise wie im nebenstehenden Bild.

Eine komplexere Version des Programms finden Sie übrigens in dem Sample Programmen zu Xerces.

6.7. Die `characters` Methode

Die `characters(char[] ch, int start, int length)` Methode ist nicht unproblematisch:

- die meisten Parser können nur eine bestimmte, limitierte Anzahl Zeichen auf einmal erkennen und senden den `characters()` Callback öfters als erwartet
- im Zeichenarray werden eine Unzahl Zeichen übermittelt, viele nicht darstellbare.
- Falls Sie im Text Leerzeichen oder andere Sonderzeichen verwenden, zerlegt der Parser den Text (je nach Parser) in mehrere Teile.

Als Ausweichmassnahme bietet es sich an, die erhaltenen Zeichen in einen `StringBuffer` zu kopieren und erst nach Erhalt der Ende-Meldung den `StringBuffer` in einen `String` und schliesslich in den gewünschten Datentyp umzuwandeln.

Wie ein solches Beispiel aussehen könnte haben wir gerade gesehen.

6.8. Behandlung von Attributen

Attribute werden im `startElement()` Callback übermittelt. Es gibt keinen Attribut Callback. Zur Analyse steht das `Attributes` Interface zur Verfügung:

```
package org.xml.sax;
public interface Attributes {
  public int    getLength ();
  public String getQName(int index);
  public String getURI(int index);
  public String getLocalName(int index);
  public int    getIndex(String uri, String localPart);
  public int    getIndex(String qualifiedName);
  public String getType(String uri, String localName);
  public String getType(String qualifiedName);
  public String getType(int index);
  public String getValue(String uri, String localName);
  public String getValue(String qualifiedName);
  public String getValue(int index);
}
```

Damit lassen sich die Attribute eines Tags in der startElement() Callback-Methode analysieren und deren Werte extrahieren, wie das folgende Codefragment zeigt:

```
public void startElement(
      String uri,
      String localName,
      String qName,
```

```
            Attributes attributes)
            throws SAXException {
            System.out.println("URI : " + uri);
            System.out.println("localName : " + uri);
            System.out.println("qName : " + uri);

            Attributes attr = attributes;
            System.out.println("Attribut Analyse");
            int iAttr = attr.getLength();
            System.out.println("Anzahl " + iAttr);
            for (int i = 0; i < iAttr; i++) {
                  System.out.println("QName " + i + " : " +
                        attr.getQName(i));
                  System.out.println("Type " + i + " : " +
                        attr.getType(i));
                  System.out.println("URI " + i + " : " + attr.getURI(i));
                  System.out.println("Value " + i + " : " +
                        attr.getValue(i));

            }
      }
```

6.9. Processing Instruction Callbacks

Die Callback `processingInstruction()` wird nur für PI's geworfen, falls diese sich innerhalb der Wurzel des XML Dokuments befinden. Die XML Deklaration zu Beginn des Dokuments (`<?xml version="1.0"...?>`) wird nicht als PI betrachtet und wird demnach nicht ausgewertet. Ab SAX 2.x soll es möglich sein, diese Informationen zu bestimmen.

6.10. Namespace Mappings – Prefix/Postfix Mapping

Die Namensdeklarationen werden nicht als Attribute ausgegeben. Stattdessen werden vor dem Aufrufen der `startElement()` Methode die `startPrefixMapping()` Methode und direkt nach Aufruf der `endElement()` Methode die `endPrefixMapping()` Methode aufgerufen.

Typischerweise wird in der `startPrefixMapping()` Methode ein Stack angelegt und die Namensraum Information angespeichert; beim Aufruf von `endPrefixMapping()` wird sie wieder aus dem Stack entfernt. Falls Sie zu irgend einem Zeitpunkt wissen möchten, welcher Präfix zu welchem Namensraum gehört, können Sie einfach den Stack durchsuchen. Das ist zwar das allgemein eingesetzte Lösungsschema, obschon eine Hashtabelle eher sinnvoller erscheint. Die Logik mit dem Stack ist in der Hilfsklasse `org.xml.sax.helpers.NamespaceSupport` implementiert:

```
package org.xml.sax.helpers;
public class NamespaceSupport {
  public final static String XMLNS
   = "http://www.w3.org/XML/1998/namespace";
  public NamespaceSupport();
  public void        reset();
  public void        pushContext();
  public void        popContext();
  public boolean     declarePrefix(String prefix, String uri);
  public String[]    processName(String qualifedName,
   String parts[], boolean isAttribute);
  public String      getURI(String prefix);
  public Enumeration getPrefixes();
  public String      getPrefix(String uri);
```

```
  public Enumeration getPrefixes(String uri);
  public Enumeration getDeclaredPrefixes();
}
```

Ein Anwendungsbeispiel finden Sie im Programm `SAXCrimsonNamespaces`. Sie finden darin den Zugriff bzw. die Ausgabe der Attribute und Namensräume, sowie einen möglichen Einsatz der Helper Klasse.

6.11. Die `ignorableWhiteSpace` Methode

Falls Sie alle bisherigen Beispiele durchgetestet haben, werden Sie bereits festgestellt haben, dass alle Sonderzeichen, wie Leerzeichen, Zeilenumbruch,... vom Parser auch interpretiert werden, obschon Sie diese evtl. nur für die bessere Lesbarkeit eingefügt haben.

Betrachten wir ein Beispiel, eine DTD für ein XML Dokument, welches der besseren Lesbarkeit wegen eingerückt wurde.

```
<?xml version="1.0"?>
<!DOCTYPE methodCall [
  <!ELEMENT methodCall (methodName, params)>
  <!ELEMENT params (param+)>
  <!ELEMENT param (value)>
  <!ELEMENT value (string)>
  <!ELEMENT methodName (#PCDATA)>
  <!ELEMENT string (#PCDATA)>
]>
<methodCall>
  <methodName>translate</methodName>
  <params>
    <param>
      <value>
        <string>
          Sicherheitsfahrzeugfahrer
        </string>
      </value>
    </param>
  </params>
</methodCall>
```

Aufgrund der DTD wissen wir, dass es sich bei den Leerzeichen nicht um #PCDATA handelt, diese also ignoriert werden können. Ein validierender Parser wird diese Zeichen nicht an die Methode `characters()` senden, sondern an die Methode `ignorableWhiteSpace()`. Sie können dies mit dem Programm `SAXCrimsonNamespaces` testen, indem Sie einfach das zu parsende Dokument durch „`WhitespacesDTD.xml`“ ersetzen.

6.12. Locators

Beim Testen eines Programms wäre es sehr hilfreich zu wissen, wo, an welcher Stelle bestimmte Elemente vorkommen, gemäss Parser.

Dies ermöglicht das `Locator` Interface. Ein `Locator` Objekt weiss, wo im Dokument ein bestimmtes Element oder Item vorkommt.

```
package org.xml.sax;
public interface Locator {
  public String getPublicId();
  public String getSystemId();
  public int    getLineNumber();
  public int    getColumnNumber();

}
```

Das folgende Beispiel zeigt ein mögliches Abfragen der Zeilennummern:

```
	private Locator locator;

	public void setDocumentLocator(Locator locator) {
		this.locator = locator;
	}

	private void printLocation(String s) {

		int line = locator.getLineNumber();
		int column = locator.getColumnNumber();
		System.out.println(s + " in Zeile " + line +
			"; Spalte " + column);
	}
	// typischer Einsatz der printLocation() Methode
	public void startDocument() {
		printLocation("startDocument()");
	}
```

Hier ein Teil der Ausgabe des Programms `SAXCrimsonLocator` :

```
startDocument() in Zeile 1; Spalte -1
startElement() in Zeile 10; Spalte -1
ignorableWhitespace() in Zeile 11; Spalte -1
ignorableWhitespace() in Zeile 11; Spalte -1
ignorableWhitespace() in Zeile 11; Spalte -1
startElement() in Zeile 11; Spalte -1
characters() in Zeile 11; Spalte -1
endElement() in Zeile 11; Spalte -1
...
```

6.13. Was Sie alles vom Parser nicht erfahren

Der Content Handler ist zwar in der Lage das meiste eines XML Dokuments zu analysieren. Aber viele Teile des Dokuments werden Sie je nach Parser Version nie analysieren können:

- Kommentare, CDATA Sektionen
  Dafür steht Ihnen das `LexicalHandler` Interface zur Verfügung.
- Namen, public IDs, System IDs und Notations für nicht geparste Entities
  Dafür steht Ihnen das DTDHandler Interface zur Verfügung.
- `ELEMENT, ATTLIST` und geparste `ENTITY` Deklarationen aus der DTD
  Dafür steht das `DeclHandler` Interface zur Verfügung.
- Validierungsfehler oder nicht-fatale Fehler
  Dafür steht das `ErrorHandler` Interface zur Verfügung.

Völlig fehlen:

- Version, Encoding und Standalone Attribute der XML Deklaration (ab SAX2.1+)
- Whitespaces in Tags und vor und nach dem Wurzelelement.
- Attributreihenfolge
- Typus der Anführungszeichen
- Ob es sich um einen leeren Tag handelt (`<Name/>`)

6.14. Zusammenfassung

SAX ist ein einfaches XML API. Die zwei wichtigsten Interfaces von SAX sind

1) XMLReader, welcher den Parser repräsentiert
2) ContentHandler, mit dessen Hilfe der Parser mit der Client Anwendung kommuniziert.