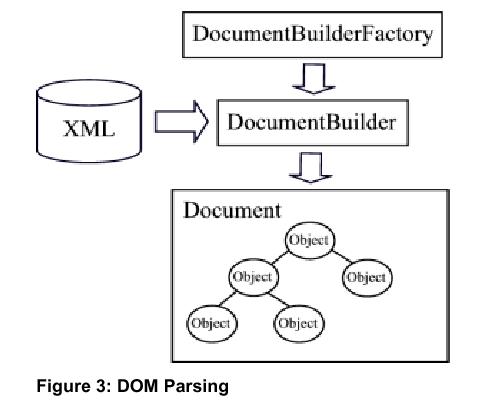
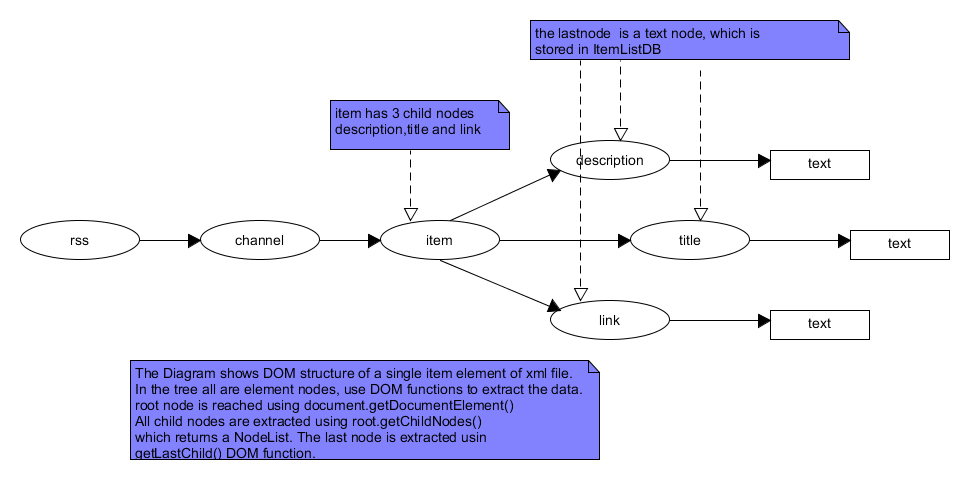
DOM:

The Document Object Model (DOM) is a set of interfaces that parse an XML document into a tree structure of objects. Each object, or Node, has a type that is represented by an interface in the package org.w3c.dom, such as Element, Attribute, Comment, and Text. This DOM tree object representation can then be manipulated just like any tree data structure. This allows for random access to particular pieces of data from the XML document and the ability to modify the XML document, which are not possible with a SAX parser.

The downside of using this API is that it is extremely memory and CPU intensive, since building the DOM requires that the entire XML structure be read and held in memory.



In our case, the DOM Parsing model is based on CBS RSS feeds. First, the user is asked about the type of news like World, Tech, Games, and Top Stories. The corresponding request is sent to the server <https://www.cbsnews.com/latest/rss/>, and the result is stored in newsrss.xml, which is parsed for the news feeds, these news feeds are stored in ItemListDB, which are displayed to the user, as shown in the following figure.

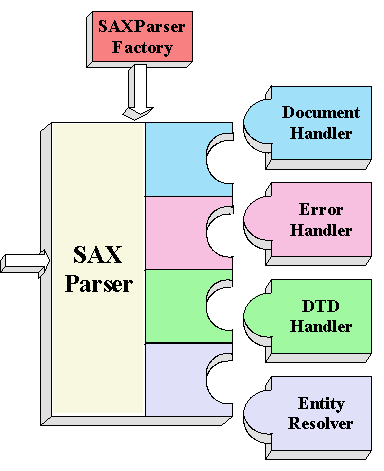


For details of the code flow, please refer to the example (Example\_DOM.txt) file.

**SAX Parser:**

DOM is an in-memory tree representation of the structure of an XML document and SAX is a standard for event-based XML parsing. XML parsers can also be used to modify or create a new XML document.

SAX is a push API i.e. you register listeners in the form of Handlers with call-back methods and the SAX parser sends (push) notifications about the XML document being processed an element, an attribute, at a time in sequential order starting at the top of the document, and ending with the closing of the ROOT element.



**DocumentHandler:**

Methods like startDocument, endDocument, startElement, and endElement are invoked when an XML tag is recognized. This interface also defines methods characters and processingInstruction, which are invoked when the parser encounters the text in an XML element or an inline processing instruction, respectively.

**ErrorHandler**

Methods error, fatalError, and warning are invoked in response to various parsing errors. The default error handler throws an exception for fatal errors and ignores other errors (including validation errors). To ensure the correct handling, you'll need to supply your own error handler to the parser.

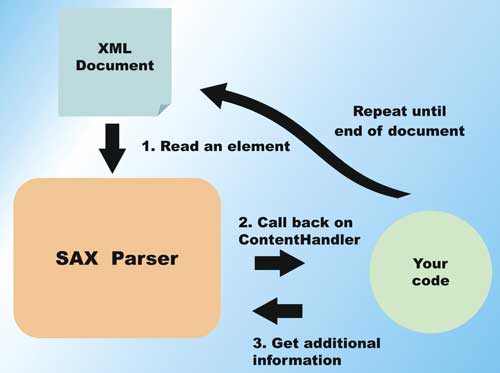
**DTDHandler**

Methods defined in this interface are invoked when processing definitions in a [DTD](http://gbengasesan.com/fyp/40/glossary.html#DTD).

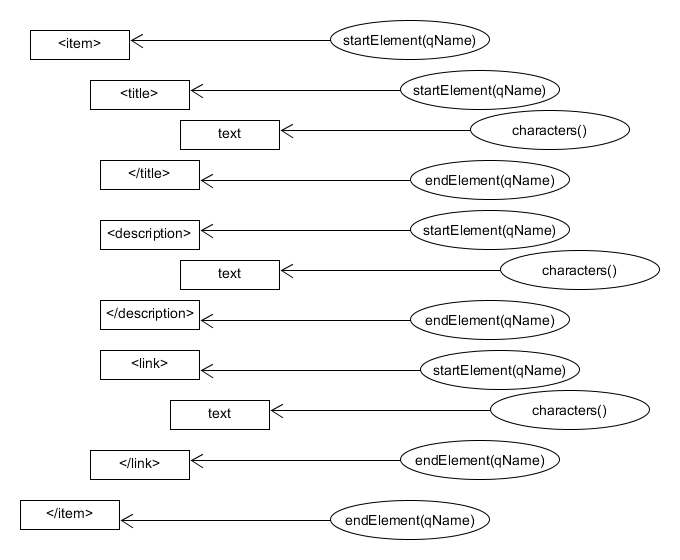
**EntityResolver**

The resolveEntity method is invoked when the parser must identify data identified by a [URI](http://gbengasesan.com/fyp/40/glossary.html#URI). In most cases, a URI is simply a [URL](http://gbengasesan.com/fyp/40/glossary.html#URL), which specifies the location of a document, but in some cases the document may be identified by a [URN](http://gbengasesan.com/fyp/40/glossary.html#URN) -- a public identifier, or name, that is unique in the web space. The public identifier may be specified in addition to the URL. The EntityResolver can then use the public identifier instead of the URL to find the document, for example to access a local copy of the document if one exists.

SAX Process is as shown below:



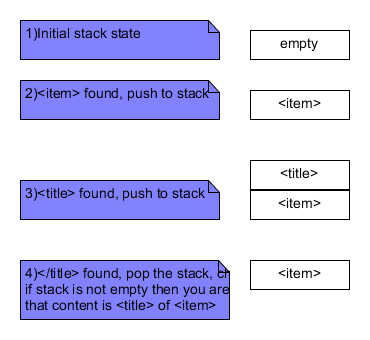
In our case, the SAX Parsing model is based on CBS RSS feeds. First, the user is asked about the type of news like World, Tech, Games, and Top Stories. The rss feeds received from server <https://www.cbsnews.com/latest/rss/>, and when the events are pushed by the parser, they are handled by the corresponding callback functions startElement(), endElement(), and characters(), defined SAXHandler (extends default handler), as shown the following figure, the data is stored in itermListDB, which are displayed to the user.

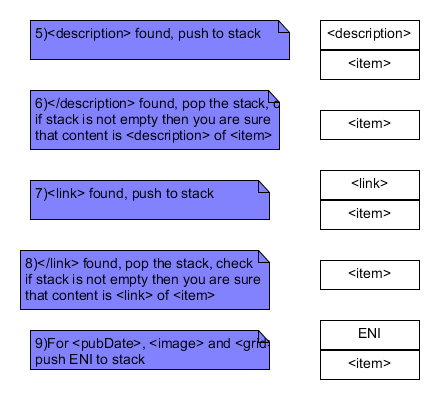


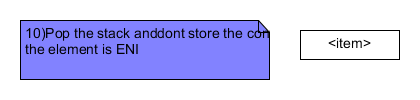
For details of the code flow, please refer to the example file(Example\_SAX.txt).

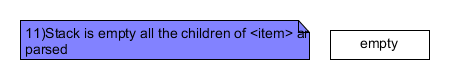
To extract the content of the <item>, a stack is used. As you can <item> of CBS RSS contains <description>, <title>, <link>, <pubDate>, <image>, <grid> elements, some of these elements are also present outside <item> also. So it is necessary to make sure that proper content of <item> are stored, and displayed to the user. The stack stores all children of <item> nodes, in our program <description>, <title>, <link> are useful and displayed to the user, and other elements <pubDate>, <image>, <grid> are not useful, hence they are marked as “ENI”, element not interested.

The stack flow of a single <item> is shown in the following figure:



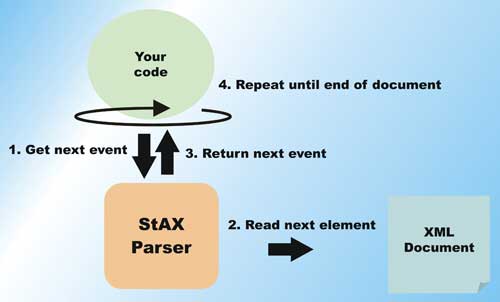


****

****

**StAX Parser:**

StAX is pull based parser, that means the client has to move the StAX parser from item to item in the XML, just like you do with a standard iterator.



Pull parsing provides several advantages over push parsing when working with XML streams:

* With pull parsing, the client controls the application thread, and can call methods on the parser when needed. By contrast, with push processing, the parser controls the application thread, and the client can only accept invocations from the parser.
* Pull parsing libraries can be much smaller and the client code to interact with those libraries much simpler than with push libraries, even for more complex documents.
* Pull clients can read multiple documents at one time with a single thread.
* A StAX pull parser can filter XML documents such that elements unnecessary to the client can be ignored, and it can support XML views of non-XML data.

The code flow is similar to SAX, except that, the code pulls the data, and parse it as an iterator

**XMLStreamReader readEvent = factory.createXMLStreamReader(is);**

**while(readEvent.hasNext()) {**

**int event; //event is <starttag>, <contents>, <endtag>**

**…**

**}**

For the code flow, check Example\_StAX.txt.