<JSTORM>

JDOM을 이용한 즐거운 JAVA/XML 프로그래밍<1부>

*Revision <**1.0**>*

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	JDOM을 이용한 즐거운 Java/XML프로그래밍 1부
Document file name:	jdom1_jinho.pdf
Revision number:	<1.0>
Issued by:	<최진호>(csecau@orgio.net)
Issued Date:	<2001/02/08>
Status:	Final

Content Information

Audience:	중급, 고급
Abstract:	Java로 XML프로그래밍을 해봤다면 그 불필요하게 복잡한 코딩이 짜증나지 않았는지? 아무리 SAX와 DOM이 표준이지만 자바는 자바만의 특성을 가진 API가 있어야 하지 않을까? 여기 JDOM이라는 매력적인 도구가 당신을 앞으로 즐거운 XML의 세계로 인도할 것이다. DOM보다 빠르고 경량이며 SAX보다 편리한 JDOM의 세계로 당신을 초대한다.
Reference:	1)http://www.javaworld.com/javaworld/jw-05-2000/jw-0518-jdom.html 2)http://jdom.org/ 3)http://www.w3.org/DOM/
참고 사항	본 문서는 자바월드의 상기 레퍼런스 1번의 문서를 번역한 문서임을 밝힌다.

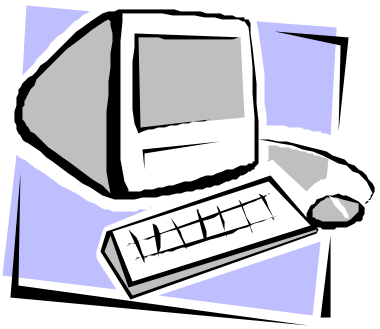
Document Approvals

박지훈		
고문	Signature	date
지위	Signature	date

Revision History

Revision	Date	Author	Description of change

Table of Contents



1부

JDOM을 이용해서 즐겁게 java/xml 프로그래밍하기

JDOM은 자바로 XML을 읽고 쓰고 제어하는 새로운 API입니다. 2000년 4월에 Jason Hunter와 Brett McLaughlin이 만들었구요. 보다 직관적이고 간단하며 생산적으로 XML을 요리해보자는 취지에서 만들어진 것이죠. 본 문서에서는 1,2부로 나눠서 어떻게 JDOM을 사용해서 XML을 읽을 것인가하는 주제를 1부에 담을 것이고 2부에서는 JDOM을 이용해서 XML데이타를 만들고 수정하는 방법을 담을 것입니다.

저자: Jason Hunter ,Brett McLaughlin
번역자: 최진호 (csecau@orgio.net)

자바 개발자들이 보다 직관적이고 간단하게 XML문서를 표현하고자 만들어진 오픈소스 API가 바로 JDOM입니다. 이름에서 풍기는 바와 같이 JDOM은은 자바에서 완벽하게 돌아가죠. 마치 자바에서 기본제공하는 collection들처럼 사용하며 따라서 XML로 어떤 일을 함에 있어 두려워 하거나 부담을 가질 이유가 없습니다.
JDOM은 특히 SAX나 DOM같은 기존의 표준과 같이 잘 돌아갑니다. 즉 그런 API들 위에 살짝 추상화를 실현한 것이죠. 존재하던 많은 API들로부터 개념들을 잘 끄집어 내어 새로운 형태의 클래스들과 인터페이스를 제공합니다. JDOM을 사용해본 어느 사용자의 평을 옮겨보자면.. '내가 처음 org.w3c.dom 패키지를 볼 때 기대했던 인터페이스가 바로 JDOM이었다.' JDOM은 DOM이나 SAX로 구성된 소스에서 데이타를 끄집어 올 수도 있을 뿐 아니라 DOM이나 SAX 형태로 결과를 줄 수가 있습니다. 따라서 이미 SAX나 DOM으로 만들어진 기존 프로그램의 컴포넌트와 전혀 이상없이 작동할 수가 있죠.

JDOM은 현재 아파치 스타일의 오픈소스 라이센스를 가지고 있습니다. 즉, 최소한의 소프트웨어 라이센스규정을 가지고 있죠. 자세한 것은 사이트에 가서 살펴보세요.(이런 건 번역하기 짜

증나죠. 필요하면 가서 보시길 바랍니다.역자주)

JDOM의 철학(?)

무조건 JDOM은 자바 개발자들에게 친숙하고 간단해야한다는 모토로 개발되었습니다. 다른 XML API들이 언어간 호환성을 생각해서 규정된 반면에 JDOM은 리플렉션이나 콜렉션, 오버로딩 메소드같은 자바의 특징을 사용합니다.

그 간단함을 위해서는 API가 프로그래머들이 기대하는 방식대로 document를 표현해야합니다. 예를 들어다음과 같은 element의 텍스트 내용을 가져올 때 자바 프로그래머는 어떤 것을 기대할까요?

<element>이것은 나의 텍스트내용이다.</element>

어떤 API에서는 element의 텍스트 내용은 오직 Element의 자식 Node일 때에만 가져올 수 있죠. 뭐 물론 기술적으로는 맞는 얘기입니다. 아래에 나오는 대로 코딩을 해야겠지요.

```
String content = element.getFirstChild().getValue();
```

그러나 JDOM은 텍스트 내용을 좀더 간단한 방법으로 가져올 수 있습니다.

```
String text = element.getText();
```

어때요? 놀랍죠? 매력적이죠?

JDOM의 두번째 철학은 빠르고 가벼워지자는 것입니다. docuement를 로드하고 제어하는 것은 정말 빨라야합니다. 또한 메모리점유율도 적어야겠죠. JDOM의 디자인은 확실하게 그것들을 지원합니다. 예를들면 정렬이나 문법이 엉성한 어떤 문서같은 경우는 DOM보다도 빠른 성능을 발휘합니다.

JDOM이 꼭 필요한가요?

좋은 질문입니다. 이미 존재하는 표준이 있으니까요. 그럼에도 불구하고 다른 걸 쓰는 게 뭐 좋겠느냐 말이죠. 하지만 존재하는 표준이 풀지 못한 문제를 JDOM이 해결합니다.

DOM은 docuement 트리를 메모리에 모두 적재하는 방식이죠. 그것은 XML 작업으로 생각되는 모든 일들을 수행해 내도록 디자인 된 거대한 API입니다. 게다가 어떤 언어이든지 사용될 수 있도록 제정된 표준 API입니다. 이런 제약때문에 DOM은 전형적인 자바 능력들 즉, 메소드 오버로딩이나 자바 표준 객체 타입들 그리고 set,get메소드들을 사용하는 자바 개발자들에게 어색하게 다가오는 것도 사실입니다. 또한 DOM은 많은 양의 프로세싱 파워와 메모리를 필요로 하죠. 따라서 경량급의 웹 어플리케이션이나 프로그램들에게 맞지 않는 점이 있습니다.

SAX는 document 트리를 메모리에 적재하지는 않죠. 대신에 document의 view를 마치 이벤트의 흐름으로서 표현해내고 있습니다. 예를들어 항상 시작 태그와 끝나는 태그를 만날 때의 모든 시점을 감지하죠. 이런 API는 빠른 입력을 위해서는 상당히 가벼운 API이지요. 그러나 document의 전체 이벤트 view라는 방식은 자바개발자들에게 별로 직관적인 구조가 아니죠. 또한 SAX는 document의 수정을 지원하지 않습니다.

JDOM은 앞서 언급한 DOM과 SAX의 장점만을 흡수한 것이죠. 적은 메모리를 가지고 빠르게 수행될 수 있는 경량급의 API입니다. 놀랍게도 JDOM은 document의 전체 view를 제공함에도 불구하고 document를 모두 메모리에 적재하지 않습니다. 이외에도 JDOM에는 장점이 수도 없이 많습니다(이외에도 침이 마르도록 칭찬한 내용이 몇 문장 있는데 지겨워서 그만 옮깁니다. 이 정도면 충분히 그 매력에 감동받거나 지겨워지셨을 것이므로..)

document를 가져오기

JDOM은 document를 org.jdom.Document의 인스턴스로 표현합니다. Document는 Comment 객체들이나 root 요소, multiple ProcessingInstruction 객체들, DocType을 제어할 수 있습니다. 이제 어떤 Factory도 필요없이 간단하게 Document를 만들어보자구요.

```
Document doc=new Document(new Element("rootElement"));
```

이제 JDOM을 사용해서 XML구조를 만드는 게 얼마나 쉬운 것인가를 보여드리죠. URL혹은 stream 아니면 file로부터 Document 객체를 만드는 방법입니다.

```
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(url);
```

위의 예제뿐 아니라 org.jdom.input 패키지에서 찾을 수 있는 builder클래스를 사용하면 어떤 데이터 소스로부터든지 Document 객체를 만들 수 있습니다. 현재 SAXBuilder와 DOMBuilder 두 개의 Builder가 있습니다. SAXBuilder는 메모리에 있는 Document객체를 실시간으로 대응하면서 SAX 이벤트를 통해 데이타를 받아들입니다. 따라서 매우 빠르죠. 대부분의 경우 이 SAXBuilder를 XML데이타 읽는데 사용하는 것을 추천합니다. 반면에 DOMBuilder는 대체로 이미 존재하는 org.w3c.dom.Document객체로 부터 Document를 생성할 때 사용되지요. JDOM은 그래서 DOM 트리 툴과 쉽게 연동될 수 있는 거죠.

JDOM의 속도는 builder가 XML데이타 소스를 스캔하는 능력에 따라 많이 좌우하지만 실제 데이터 요청이 들어오기 전에 모든 요소들을 파싱하지는 않습니다. 예를들어 element속성은 그 값이 요청되기전에는 파싱하지 않는거죠.

이 Builder인터페이스는 SQL쿼리나 LDAP쿼리 혹은 기타 어떤 데이타 포맷이든간에 JDOM Document로 만들 수 있도록 유연하게 되어있습니다. 그래서 한번 메모리에 들어가면 그것들의 build툴과의 연결고리는 없죠.

SAXBuilder와 DOMBuilder의 생성자에는 사용자가 실제 파싱할 때 유효성을 검사할 것인지 아닌지를 선택할 수 있게 되어있습니다.

```
public SAXBuilder(String parserClass, boolean validation);
public DOMBuilder(String adapterClass, boolean validation);
```

아파치의 오픈소스인 Xerces파서를 사용한다면 validation을 false로 하고 사용하세요.

DOMBuilder의 첫번째 인자가 parserClass가 아니고 adapterClass인 이유는 모든 DOM파서가 같은 API를 가진게 아니기 때문이죠. 사용자가 임의로 파서들을 붙일 수 있도록 JDOM은 모든 DOM파서가 사용될 수 있는 공통된 API를 가진 adapter클래스를 사용합니다. 이 adapter는 아파치의 Xerces나 Crimson, IBM의 XML4J 썬의 X프로젝트 그리고 오라클의 V1,V2파서등의 유명한 DOM파서들을 지원합니다.

Document 출력

여러가지 제공하는 output툴를 이용해서 Document를 출력할 수 있습니다. org.jdom.output.XMLOutputter 툴은 아마 가장 자주 쓰이게 될 겁니다. 그것은 Document객체를 특정 OutputStream에 출력하지요.

SAXOutputter툴은 좀 특별한 툴입니다. 이것은 JDOM Document 객체에 기반한 SAX이벤트를 발생합니다. 따라서 SAX이벤트를 받는 어떤 애플리케이션 컴포넌트에게 이벤트를 전달할 수 있겠지요. 이와 유사한 방법으로 DOMOutputter는 DOM Document를 생성하는데 DOM객체를 받는 애플리케이션 컴포넌트에게 넘겨줄 수 있겠지요. Document 객체를 출력하는 간단한 예제입니다.

```
XMLOutputter outputter = new XMLOutputter();
outputter.output(doc, System.out);
```

여기 XML 문서를 읽어서 출력하는 간단한 클래스 예제가 있습니다.

```
import java.io.*;
import org.jdom.*;
import org.jdom.input.*;
import org.jdom.output.*;

public class PrettyPrinter {
   public static void main(String[] args) {
        // Assume filename argument
        String filename = args[0];

        try {
            // Build the document with SAX and Xerces, no validation
            SAXBuilder builder = new SAXBuilder();
            // Create the document
            Document doc = builder.build(new File(filename));

            // Output the document, use standard formatter
```

```
            XMLOutputter fmt = new XMLOutputter();
            fmt.output(doc, System.out);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

DocType을 읽기

이제 어떻게 Document의 세부적인 요소들을 읽어내는지 살펴봅시다. 대부분의 XML Document는 Document 타입을 하나씩 가지고 있습니다. 이를 JDOM에서는 DocType클래스에서 가지고 있습니다. 독자가 XML초보라고 가정한다면(너무 기분나쁘게는 생각지 마세요. 저는 그대로 번역할 뿐임다),
Document 타입은 다음과 같이 표현합니다.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

DOCTYPE다음에 나오는 단어는 element가 속해있는 이름을 뜻하고 PUBLIC다음의 문장은 Document 타입의 공개적인 identifier입니다. 그리고 마지막 문장은 Document 타입의 시스템 identifier입니다. DocType은 Document클래스에서 getDoctype()에 의해 얻을 수 있습니다. 또한 DocType클래스는 DOCTYPE선언의 부분적인 조각들을 반환하는 메소드도 있지요.

```
DocType docType = doc.getDocType();
System.out.println("Element: " + docType.getElementName());
System.out.println("Public ID: " + docType.getPublicID());
System.out.println("System ID: " + docType.getSystemID());
```

element데이타 읽기

모든 XML Document는 반드시 root element를 가집니다. 이 element는 Document내의 정보들의 시작 포인트입니다. 예를 들어 <web-app>라는 root를 가진 Document의 한 부분을 보면..

```
<web-app id="demo">
    <description>Gotta fit servlets in somewhere!</description>
    <distributable/>
</web-app>
```

하나의 Document객체에서 root element 인스턴스는 유효합니다.

Element webapp = doc.getRootElement();

자, 이제 Element의 속성(예를 들면 id), 내용, 자식 Element들을 제어할 수 있는 기본적인 모습이 되었네요.

자식이랑 놀자(Playing with children)

XML Document는 모두 트리 구조이고 Element는 임의의 수의 자식 Element들을 가집니다. 또 예를 들면, 위에서 <web-app> element는 <description>과 <distributable> 자식 태그를 가지죠?

그렇담 Element의 자식을 여러 메소드를 이용해서 가져올 수 있습니다. getChild()메소드는 만약 존재하는 이름의 자식이 없다면 null을 반환합니다.

```
List getChildren(); // return all children
List getChildren(String name); // 같은이름 가진 자식들 모두 반환
Element getChild(String name); // 이름을 가진 자식들 중의 첫번째 자식 반환
```

/* 자꾸 자식 자식 하니까 욕하는 거 같군요... 그래도 새끼보단 낫죠?

예제를 들어드립니다.

```
// Get a List of all direct children as Element objects
List allChildren = element.getChildren();
out.println("First kid: " + ((Element)allChildren.get(0)).getName());

// Get a list of all direct children with a given name
List namedChildren = element.getChildren("name");

// Get a list of the first kid with a given name
Element kid = element.getChild("name");
```

XML Document가 잘 알려진 문서 구조일 경우에는 getChild()메소드를 사용할 경우 매우 빠르게 근처 element들을 접근할 수 있습니다. 다음과 같은 xml이 있다고 합시다.

```
<?xml version="1.0"?>
<linux:config>
  <gui>
    <window-manager>
      <name>Enlightenment</name>
      <version>0.16.2</version>
    </window-manager>
    <!-- etc -->
  </gui>
```

```
</linux:config>
```

이럴 window manager 이름을 접근하는 직접적인 코드는 다음과 같죠.

```
String windowManager = rootElement.getChild("gui")
                                  .getChild("window-manager")
                                  .getChild("name")
                                  .getText();
```

Document가 유효하지 않을 때 일어나는 NullPointerExceptions을 주의하기 바랍니다. 간단한 문서 탐색기를 위해서 차후의 JDOM 버젼에서는 XPath를 지원할 겁니다. 자식은 getParent()를 이용해서 자신의 애비를 찾을 수 있습니다.(-_-)

element속성을 가져오기

속성 즉, Attribute은 element가 가진 정보 조각이라고 볼 수 있죠. 어떤 html프로그래머라도 개념을 이해하는 데에는 별로 어려운 점이 없을 겁니다. 즉 <table> element는 width와 border 속성을 가지고 있죠?

```
<table width="100%" border="0"> </table>
```

이 속성은 곧바로 Element에서 가져올 수 있습니다.

```
String width = table.getAttributeValue("width");
```

물론 이런 스트링이 아닌 Attribute 클래스의 인스턴스로 반환할 수도 있지요. 이러한 JDOM의 기능은 속성같은 것들이 namespace에 사용될 수 있게 하는 아주 유용한 기술입니다. (더 자세한 정보를 알고자 하는 분은 다른 곳에서 namespace에 대해 살펴보세요.)

```
Attribute widthAttrib = table.getAttribute("width");
String width = widthAttrib.getValue();
```

좀 더 편리함을 도모하고자 Attribute클래스는 다양한 primitive type으로 정보를 반환합니다.

```
int width = table.getAttribute("border").getIntValue();
```

어떤 원시타입으로든지 값을 반환할 수가 있지요. 만약 attribute가 원시 타입으로 변환될 수 있는 성질의 것이 아니라면 DataConversionException이 발생합니다. attribute가 존재하지 않는다면 getAttribute()는 null값을 반환하고요..

element 내용 추출

이제 element 내용을 좀 더 쉽게 가져오는 방법을 생각해보고, element.getText()를 사용해서 element의 텍스트 내용을 추출하는 쉬운 방법을 보이겠습니다. 표준적인 방법이라면 다음과 같은 elements를 생각하겠지만..

```
<name>Enlightenment</name>
```

가끔 element는 중간에 주석을 담기도 하고 텍스트 내용이나 자식 element들을 가지기 마련이죠.

```
<table>
    <!-- Some comment -->
```

```
            Some text
            <tr>Some child</tr>
            <?pi Some processing instruction?>
        </table>
```

그래서 다음과 같이 텍스트와 자식을 다음과 같이 가져옵니다.

```
        String text = table.getText(); // "Some text"
        Element tr = table.getChild("tr"); // <tr> child
```

아주 간단한 표준적인 방법이죠. 가끔 출력을 해야할 때, 특정 Element의 모든 내용을 얻어야 할 때가 있습니다. 그런 경우 Element의 getMixedContent()메소드를 이용하면 특정 Element 내에 있는 모든 주석이나 텍스트, Element등의 모든 내용을 List객체로 반환합니다.
아래 코드는 하나의 element의 내용의 전부를 프린트하는 예제입니다.

```
        List mixedContent = table.getMixedContent();
        Iterator i = mixedContent.iterator();
        while (i.hasNext()) {
            Object o = i.next();
            if (o instanceof Comment) {
                // Comment has a toString()
                out.println("Comment: " + o);
            }else if (o instanceof String) {
                out.println("String: " + o);
            }else if (o instanceof ProcessingInstruction) {
                out.println("PI: " + ((ProcessingInstriction)o).getTarget());
            }else if (o instanceof Element) {
                out.println("Element: " + ((Element)o).getName());
            }
        }
```

처리지시(Processing instructions)와 함께

처리지시자(processing instruction:PI)는 가끔 어떤 XML Document에서는 툴을 제어하기 위해 사용됩니다.
예를 들어 Cocoon 웹 컨텐츠 생성 라이브러리에선 xml파일은 담은과 같은 처리지시를 가지고 있죠.

```
        <?cocoon-process type="xslt"?>
```

각 처리지시 인스턴스는 target과 data를 가지고 있죠. target은 첫번째 단어고 data는 뒤에 나오는 모든 단어들입니다. 그리고 이것들은 각각 getTarget()과 getData()를 이용해 반환하죠..

```
        String target = pi.getTarget(); // cocoon-process
```

```
String data = pi.getData(); // type="xslt"
```

여기서 data는 attribute의 리스트와 유사하기 때문에 Processinginstruction클래스는 내부적으로 getValue(String name)으로 attribute과 같은 방식으로 data를 반환하죠. 따라서 다음과 같이 사용해도 무방합니다.

```
String type = pi.getValue("type"); // xslt
```

처리지시는 주석과 마찬가지로 Document 어디에나 존재하므로 그것들을 주석들과 같은 방식으로 가져올 수 있습니다.

```
List mixed = element.getMixedContent(); //리스트에는 처리지시도 담길겁니다.
```

처리지시는 root element 밖에 존재할 수도 있습니다. 그래서 다음과 같이 Document클래스에서 getMixedContent()메소드를 실행하면 역시 처리지시를 가져올 수가 있죠.

```
List mixed = doc.getMixedContent();
```

실제로 root element 외부에 있는 처리지시는 종종 있어서 Document클래스는 Document레벨에서의 처리지시를 반환하는 여러 메소드를 지원합니다.

```
List allOfThem = doc.getProcessingInstructions();
List someOfThem = doc.getProcessingInstructions("cocoon-process");
ProcessingInstruction oneOfThem =
        doc.getProcessingInstruction("cocoon-process");
```

이렇게 하면 Cocoon 파서는 첫번째로 cocoon-process타입을 코드로 부터 다음과 같이 읽습니다.

```
String type =
        doc.getProcessingInstruction("cocoon-process").getValue("type");
```

Namespaces

Namespace는 XML을 진일보시킨 중요한 개념입니다. Namespace는 서로 다른 namespace를 가진 동일한 element이름을 서로 구별하게 해주죠. name 충돌을 피하기 위한 자바 패키지도 유사한 일을 하죠.

JCOM에서는 helper class인 org.jdom.Namespace에서 namespace를 지원합니다.
Namespace.getNamesapce(String prefix, String uri)메소드를 이용해서 namespace를 받죠. 어떤 자식이 하나의 namespace내에 있을 때 그 자쉭을 getChild()와 getChildren()의 오버로드된 버젼을 이용해 두번째 인자에다가 Namespace 인자를 줌으로써 가져올 수있습니다.

```
Namespace ns =
        Namespace.getNamespace("xhtml", "http://www.w3.org/1999/xhtml");
List kids = element.getChildren("p", ns);
Element kid = element.getChild("title", ns);
```

namespace가 주어지지 않았다면 element는 default의 namespace에 있다고 가정합니다. 귀찮다면 namespace는 무시해도 좋겠죠.. ^ ^

리스트 만들기, 중요체크하기

JDOM은 J2SE에서 제공하는 Collection인 List와 Map을 이용해서 디자인 되었죠. 그래서 상당히 유연하고 파워풀한 기능을 발휘하는 겁니다.
List와 Map은 좀 불규칙적이죠. 불규칙적이라는 의미는 그것들의 내용이 바뀔 수 있고 재정렬이나 더하고 지우는 일련의 변화를 줄 수 있다는 뜻입니다. 이 부분은 2장에서 자세하게 다루겠습니다.

예외

아마도 JDOM라이브러리 내에서 여러 예외 클래스를 봤을 겁니다. 편리성을 더하기 위해 모든 예외클래스들은 JDOMException에서 상속받은 것이죠. 모든 JDOMException을 세세하게 종류별로 예외처리 할 수 있기 때문에 편리하죠.

web.xml파일을 읽기 위해 JDOM을 사용해보기

이제 서블릿 API2.2의 웹 어플리케이션 디플로이 설정 파일인 web.xml파일을 파싱하기 위해 JDOM을 사용해보죠. 사용자가 웹어플리케이션에서 서블릿이 등록된 것, 얼마나 많은 init파라미터를 각 서블릿에서 가지고 있는지, 보안 규칙이 어떻게 정의되어있고 웹 어플리케이션이 분산되어 있는지 아닌지를 알고자 한다고 가정해봅시다.

여기 예제 web.xml파일이 있습니다.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <servlet>
        <servlet-name>snoop</servlet-name>
        <servlet-class>SnoopServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>file</servlet-name>
        <servlet-class>ViewFile</servlet-class>
        <init-param>
            <param-name>initial</param-name>
```

```
            <param-value>1000</param-value>
            <description>
                The initial value for the counter <!-- optional -->
            </description>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>mv</servlet-name>
        <url-pattern>*.wm</url-pattern>
    </servlet-mapping>

    <distributed/>

    <security-role>
      <role-name>manager</role-name>
      <role-name>director</role-name>
      <role-name>president</role-name>
    </security-role>
</web-app>
```

파일을 처리할 때 다음과 같이 출력되길 원한다고 할 때

```
This WAR has 2 registered servlets:
        snoop for SnoopServlet (it has 0 init params)
        file for ViewFile (it has 1 init params)
This WAR contains 3 roles:
        manager
        director
        president
This WAR is distributed
```

JDOM을 이용해서 출력하는 건 정말 쉽습니다.

```
import java.io.*;
import java.util.*;
import org.jdom.*;
import org.jdom.input.*;
import org.jdom.output.*;
```

```
public class WarReader {

  public static void main(String[] args) {

    PrintStream out = System.out;

    if (args.length != 1 && args.length != 2) {
      out.println("Usage: WarReader [web.xml]");
      return;
    }

    try {
      // 유효성검사 없이 Document building을 요청한다.
      SAXBuilder builder = new SAXBuilder(false);
      Document doc = builder.build(new File(args[0]));

      // Get the root element
      Element root = doc.getRootElement();

      // Print servlet information
      List servlets = root.getChildren("servlet");
      out.println("This WAR has "+ servlets.size() +" registered servlets:");
      Iterator i = servlets.iterator();
      while (i.hasNext()) {
        Element servlet = (Element) i.next();
        out.print("\t" + servlet.getChild("servlet-name")
                                  .getText() +
                    " for " + servlet.getChild("servlet-class")
                                  .getText());
        List initParams = servlet.getChildren("init-param");
        out.println(" (it has " + initParams.size() + " init params)");
      }

      // Print security role information
      List securityRoles = root.getChildren("security-role");
      if (securityRoles.size() == 0) {
        out.println("This WAR contains no roles");
```

```
      }
      else {
        Element securityRole = (Element) securityRoles.get(0);
        List roleNames = securityRole.getChildren("role-name");
        out.println("This WAR contains " + roleNames.size() + " roles:");
        i = roleNames.iterator();
        while (i.hasNext()) {
          Element e = (Element) i.next();
          out.println("\t" + e.getText());
        }
      }

      // Print distributed information (notice this is out of order)
      List distrib = root.getChildren("distributed");
      if (distrib.size() == 0) {
        out.println("This WAR is not distributed");
      } else {
        out.println("This WAR is distributed");
      }

    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

End of Document