

# Document Type Definitions

## Schemas

A schema is a set of rules that defines the structure of elements and attributes and the types of their content and values in an XML document.

Analogy: A schema specifies a collection of XML documents in the same way a BNF definition specifies the syntactically correct programs in a programming language.

A schema defines what elements occur in a document and the order in which they appear and how they are nested; it also tells what attributes belong to which elements and describes their types to some extent.

## Advantages of Schemas

- Define the characteristics and syntax of a set of documents.
- Independent groups can have a common format for interchanging XML documents.
- Software applications that process the XML documents know what to expect if the documents adhere to a formal schema.
- XML documents can be validated to verify that they conform to a given schema.
- Validation can be used as a debugging tool, directing the designer to items in a document that violate the schema.
- A schema can act as documentation for users defining or reading some set of XML documents.
- A schema can increase the reliability, consistency, and accuracy of exchanged documents.

## Document Type Definitions (DTDs)

DTDs were originally part of SGML (Standard Generalized Markup Language) the ancestor of both HTML and XML.

DTD is the most common schema language in use with XML documents.

A Document Type Definition consist of a set of rules of the following forms:

<!ELEMENT ... >

<!ATTLIST ... >

<!ENTITY ... >

<!NOTATION ... >

### Two Possibilities

Internal Subset: DTD rules are written as part of an XML document.

External Subset: DTD rules are placed in a separate file, usually with ".dtd" as its file extension, and referred to from inside the XML document.

## Element Specification

Elements in a document are defined by rules of the form:

<!ELEMENT *nameOfElement contentOfElement*>

### Possible Content

Sequence of elements (use a comma)

<!ELEMENT name (first, middle, last)>

Choice of elements (use |)

```
<!ELEMENT student (undergrad | grad | nondegree)>
```

Textual data (parsed character data)

```
<!ELEMENT first (#PCDATA)>
```

```
<!-- zero or more characters -->
```

Optional elements (use ?)

```
<!ELEMENT name (first, middle?, last)>
```

Repeated elements (use \* and +)

```
<!ELEMENT entries (entry*)> <!-- zero or more elements -->
```

```
<!ELEMENT entries (entry+)> <!-- one or more elements -->
```

Parentheses can be used for grouping

```
<!ELEMENT article (title, (abstract | body))>
```

```
<!ELEMENT article ((title, abstract) | body)>
```

Empty content

```
<!ELEMENT signal EMPTY>
```

Any content (might be used during development)

```
<!ELEMENT tag ANY>
```

Mixed content

```
<!ELEMENT narrative (#PCDATA | italics | bold)*>
```

#PCDATA must come first; no sequencing (,) and no occurrence modifiers (\*, +, ?) are allowed on subelements.

## Example

Put an internal DTD into the XML document *phone2.xml*.

The DTD declaration that contains the DTD rules comes immediately after the xml declaration.

The DTD rules are delimited by `<!DOCTYPE rootElement [ and ]>`.

### File: phoneD.xml

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE phoneNumbers [
<!ELEMENT phoneNumbers (title, entries)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT entries (entry*)>
<!ELEMENT entry (name, phone, city?)>
<!ELEMENT name (first, middle?, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT city (#PCDATA)>
]
```

```
<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
    <entry>
      <name>
        <first>Rusty</first>
        <last>Nail</last>
      </name>
```

```
<phone>335-0055</phone>
<city>Iowa City</city>
</entry>
<entry>
  <name>
    <first>Justin</first>
    <last>Case</last>
  </name>
  <phone>354-9876</phone>
  <city>Coralvile</city>
</entry>
<entry>
  <name>
    <first>Pearl</first>
    <middle>E.</middle>
    <last>Gates</last>
  </name>
  <phone>335-4582</phone>
  <city>North Liberty</city>
</entry>
<entry>
  <name>
    <first>Helen</first>
    <last>Back</last>
  </name>
  <phone>337-5967</phone>
  <city>Iowa City</city>
</entry>
</entries>
</phoneNumbers>
```

## Validation

Many tools are available to validate an XML document against a DTD specification.

Most common XML parsers can be configured to perform the validation as a document is parsed.

Our first example has an XML parser in a command-line tool *xmllint*, which is installed on the Department's Linux machines.

To validate *phoneD.xml* with the internal DTD, enter:

```
% xmllint --valid phoneD.xml
```

If the document is valid, it is parsed and printed.

If invalid, all errors are reported, but the document is still parsed if it is well-formed.

To check whether the document is well-formed only, enter:

```
% xmllint phoneD.xml
```

Alternatively, the DTD specification can be placed in a separate file, say *phone.dtd*.

This file contains only the sequence of DTD rules that fall between the brackets in the internal definition.

```
<!ELEMENT phoneNumbers (title, entries)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT entries (entry*)>
<!ELEMENT entry (name, phone, city?)>
<!ELEMENT name (first, middle?, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

The DTD declaration in the XML document needs to be changed to refer to this separate file. Use the keyword SYSTEM for a local file with a path specification or for a file in a web directory for personal use.

### File: phoneED.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE phoneNumbers SYSTEM "phone.dtd">

<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
    <entry>
      <name>
        <first>Rusty</first>
        <last>Nail</last>
      </name>
      <phone>335-0055</phone>
      <city>Iowa City</city>
    </entry>
  </entries>
</phoneNumbers>
```

### File: phoneWD.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE phoneNumbers SYSTEM
  "http://www.cs.uiowa.edu/~slonnegr/xml/phone.dtd">

<phoneNumbers>
  <title>Phone Numbers</title>
  <entries>
  </entries>
</phoneNumbers>
```

The execution of *xmllint* remains the same.

```
% xmllint --valid phoneED.xml
```

and

```
% xmllint --valid phoneWD.xml
```

A slightly nonstandard technique works when the XML document has no DTD declaration at all. An option for *xmllint* can be used to indicate the DTD file to use.

```
% xmllint --dtdvalid phone.dtd phone2.xml
```

The tool *xmllint* has many options, some of which we will be using later. You can get a detailed description using:

```
% man xmllint
```

## Element Example: elems.dtd

```
<!ELEMENT root (one+, (two | three)+,  
                four*, (five*, six)+, (one | two)?)>  
<!ELEMENT one EMPTY>  
<!ELEMENT two EMPTY>  
<!ELEMENT three EMPTY>  
<!ELEMENT four EMPTY>  
<!ELEMENT five EMPTY>  
<!ELEMENT six EMPTY>
```

## Valid Instance: e1.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE root SYSTEM "elems.dtd">
<root>
  <one/><one/><two/><two/><three/><four/>
  <five/><six/><five/><six/><one/>
</root>
```

## Valid Instance: e2.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE root SYSTEM "elems.dtd">
<root>
  <one/><three/><six/>
</root>
```

## Valid Instance: e3.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE root SYSTEM "elems.dtd">
<root>
  <one/><one/><two/><two/><three/><four/><six/><six/><one/>
</root>
```

## Invalid Instance: e4.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE root SYSTEM "elems.dtd">
<root>
  <one/><one/><two/><two/><three/>
  <four/><six/><six/><five/><one/>
</root>
```

## Attributes

The attributes for any element can be specified using a rule of the form (order makes no difference):

```
<!ATTLIST elemName attName1 attType restriction/default  
           attName2 attType restriction/default  
           attName3 attType restriction/default>
```

Some designers prefer individual rules.

```
<!ATTLIST elemName attName1 attType restriction/default>  
<!ATTLIST elemName attName2 attType restriction/default>  
<!ATTLIST elemName attName3 attType restriction/default>
```

The element name specifies which element the attribute belongs to. That means the attribute name and value will be included in the start tag for that element, unless the attribute is optional.

The numerous types for the attribute values can be seen on the next page.

Each attribute definition must have a restriction specification or a default value or both in one case. The possible formats can be found two pages on.

## Attribute Types

Attribute values can be constrained to any one of the following types.

CDATA	Value is made up of character data; entity references must be used for the special characters (<, >, \$, ", ').
(val <sub>1</sub>   val <sub>2</sub>   ...   val <sub>k</sub> )	Value must be one from an enumerated list whose values are each an NMTOKEN (see below).
ID	Value is an legal XML name that is unique in the document.
IDREF	Value is the ID attribute value of another element.
IDREFS	Value is a list of IDREFs, separated by spaces.
NMTOKEN	Value is a token similar to a valid XML element name, except it can begin with a digit, period, or hyphen.
NMTOKENS	Value if a list of NMTOKENs, separated by spaces.
ENTITY	Value is an entity declared elsewhere in the DTD.
ENTITIES	Value is a list of ENTITY values.
NOTATION	Value is a NOTATION defined in the DTD that specifies a type of binary data to be included in the document. This type is used rarely.

## Restrictions and Default Values

Some keywords put restrictions on the occurrences of an attribute and its value. Further, in some situations, a default attribute value can be specified in the DTD, but note that some parsers may not recognize default values.

### Possible Formats

1. Any regular attribute definition can have a default value.

```
<!ATTLIST cube size CDATA "10">
```

```
<!ATTLIST pixel color (red | green | blue) "blue">
```

2. An attribute for a particular element may be optional.

```
<!ATTLIST person age CDATA #IMPLIED>
```

3. An attribute for a particular element may be mandatory.

```
<!ATTLIST population year CDATA #REQUIRED>
```

4. An attribute for a particular element may be a fixed constant, in which case there must be a default value. The value of an ID type cannot be fixed.

```
<!ATTLIST language name NMTOKEN #FIXED "Java">
```

These formats require that the attribute type should be followed by a default value *or* by #IMPLIED *or* by #REQUIRED *or* by #FIXED *and* a default value.

## Attribute Example: product.dtd

Describe a catalog of tools to be sold by some company.

```
<!ELEMENT catalog (product+)>
<!ELEMENT product
    (specifications+, options?, price+, notes?)>
  <!ATTLIST product name CDATA #REQUIRED>
  <!ATTLIST product category
    (HandTool|Table|ShopPro) "HandTool">
  <!ATTLIST product partnum NMTOKEN #REQUIRED>
  <!ATTLIST product plant
    (Boston|Buffalo|Chicago) "Chicago">
  <!ATTLIST product inventory
    (InStock|BackOrdered|Discontinued) "InStock">
<!ELEMENT specifications (#PCDATA)>
  <!ATTLIST specifications weight CDATA #IMPLIED>
  <!ATTLIST specifications power NMTOKEN #IMPLIED>
<!ELEMENT options EMPTY>
  <!ATTLIST options finish (Metal|Polished|Matte) "Matte">
  <!ATTLIST options adapter
    (Included|Optional|NotApplicable) "Included">
  <!ATTLIST options case
    (HardShell|Soft|NotApplicable) "HardShell">
<!ELEMENT price (#PCDATA)>
  <!ATTLIST price msrp CDATA #IMPLIED>
  <!ATTLIST price wholesale CDATA #IMPLIED>
  <!ATTLIST price street CDATA #IMPLIED>
  <!ATTLIST price shipping CDATA #IMPLIED>
<!ELEMENT notes (#PCDATA)>
```

## An Instance of product.dtd: product.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE catalog SYSTEM "product.dtd">

<catalog>
  <product name="power drill" partnum="p23-456"
           plant="Boston" inventory="InStock">
    <specifications weight="18.3" power="ac">
    </specifications>
    <price msrp="12.33" wholesale="18.45"
           shipping="5.67">
      Keep these prices up-to-date.
    </price>
    <notes>
      Our best selling item.
    </notes>
  </product>
  <product name="radial saw" partnum="p83-730"
           category="Table" inventory="BackOrdered">
    <specifications weight="88.6"/>
    <price msrp="172.27" wholesale="144.85"
           street="157.94" shipping="19.80"/>
  </product>
</catalog>
```

## Test validity

```
% xmllint --valid --noout product.xml
%
```

The flag *noout* means no output.

## Other Validators

Several DTD validators can be found on the web.

### **Brown University: STG**

<http://www.stg.brown.edu/service/xmlvalid/>

This web page allows the user to upload an XML document file, but does not seem to be able to find a DTD file in the user's local directories.

### **Richard Tobin: XML Checker**

<http://www.ltg.ed.ac.uk/~richard/xml-check.html>

For this checker, the XML document file must reside in a web directory, say

<http://www.cs.uiowa.edu/~slonnegr/xml/roster.xml>".

## Validating with Java

In the next example, we use a DOM (Document Object Model) parser that is invoked from a Java program to parse an XML document and thereby check if it is well-formed and by setting a flag, check if it is valid relative to a DTD.

The classes and interfaces that we use are found in three packages available in Java 1.4 and beyond.

```
javax.xml.parsers  
org.w3c.dom  
org.xml.sax
```

The program `DomCheck.java` shows how to set up a basic DOM parser. We investigate DOM more deeply in the next section.

## Program: DomCheck.java

The code resides entirely in the main method.

From the command line, we execute the program using

```
% java DomCheck [-v] fileName
```

Without the "-v" option, the XML file is checked for being well-formed only.

With the "-v", it is further checked for validity against a DTD that it must be able to find.

```
// Show individual classes and interfaces for XML imports.
```

```
import javax.xml.parsers.DocumentBuilderFactory;  
import javax.xml.parsers.DocumentBuilder;  
import javax.xml.parsers.ParserConfigurationException;  
import org.w3c.dom.Document;  
import org.xml.sax.InputSource;  
import org.xml.sax.SAXException;  
import java.io.*;
```

```
public class DomCheck  
{  
    static public void main(String [] args)  
    {  
        String filename = null;  
        boolean validate = false;  
  
        if (args.length == 1)  
            filename = args[0];  
        else if (args.length == 2 && args[0].equals("-v"))  
        {  
            filename = arg[1];  
            validate = true;  
        }  
    }  
}
```

```

else
{
    System.out.println(
        "Usage: java DomCheck [-v] fileName");
    return;
}

// Create a DOM factory to create a parser that
// validates or not according to the flag setting.

DocumentBuilderFactory dbf =
    DocumentBuilderFactory.newInstance();
dbf.setValidating(validate);

// Use the factory to create a parser (builder) and
// use it to parse the XML document.

try
{
    DocumentBuilder parser = dbf.newDocumentBuilder();
    InputSource is = new InputSource(filename);
    Document doc = parser.parse(is);

    // If no exception, the test was successful.
    System.out.print("XML document is well formed");
    if (validate)
        System.out.print(" and valid");
    System.out.println(".");
}
catch (SAXException e)
{ System.out.println(e); }
catch (ParserConfigurationException e)
{ System.out.println(e); }
catch (IOException e)
{ System.out.println(e); }
}
}

```

## Error Handling

The DomCheck program works as it stands, but when an error occurs during the parsing, the error message printed shows a lot of irrelevant information.

We need to take charge of the error reporting by writing a class that implements the interface `org.xml.sax.ErrorHandler` and by registering an object from this class with the `DocumentBuilder` object that we created in `DomCheck`.

In the following class we implement the three methods in the `ErrorHandler` interface, using data in the `SAXParseException` to display information about the error that occurred.

### The errors come in three levels

Fatal Error: XML Document is not well-formed.

Error: Document does not agree with its DTD or it has no DTD.

Warning: The document can be parsed successfully but some definition in the DTD is misleading.

When handling an error or a fatal error, we throw the exception from the method so that the program terminates.

```
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXParseException;

class MyErrorHandler implements ErrorHandler
{
    public void warning(SAXParseException e)
    {
        System.out.println("Warning:");
        showDetails(e);
    }
}
```

```

public void error(SAXParseException e)
                    throws SAXParseException
{
    System.out.println("Error:");
    showDetails(e);
    throw(e);
}

public void fatalError(SAXParseException e)
                    throws SAXParseException
{
    System.out.println("Fatal error:");
    showDetails(e);
    throw(e);
}

void showDetails(SAXParseException e)
{
    System.out.println(e.getMessage());
    System.out.println(" Line: " + e.getLineNumber());
    System.out.println(" Column: " + e.getColumnNumber());
    System.out.println(" Document: " + e.getSystemId());
    System.out.println();
}
}

```

## Registering the Error Handler

Set the error handler using the `DocumentBuilder` object right after it is created in the program `DomCheck.java`.

```

DocumentBuilder parser = dbf.newDocumentBuilder();
parser.setErrorHandler(new MyErrorHandler());

```

## Example of a Fatal Error (well-formed violation)

Change the first `</name>` tag to `<name>`.

```
% java DomCheck -v phwf.xml
```

Fatal error:

The element type "name" must be terminated by the matching end-tag "</name>".

Line: 24

Column: 8

Document: file:///mnt/nfs/fileserv/fs3/slonnegr/phwf.xml

org.xml.sax.SAXParseException: The element type "name" must be terminated by the matching end-tag "</name>".

## Example of an Error (invalid document)

Remove the definition of the city element from the DTD.

```
% java DomCheck -v phnocity.xml
```

Error:

Element type "city" must be declared.

Line: 22

Column: 15

Document: file:///mnt/nfs/fileserv/fs3/slonnegr/phnocity.xml

org.xml.sax.SAXParseException: Element type "city" must be declared.

## Warnings

The XML recommendation describes several situations where the parser may provide a warning message:

- A declaration mentions an (optional) element type for which no declaration is provided.
- An attribute is declared for an element that is not itself declared
- More than one attribute-list declaration is provided for a given element type or more than one attribute definition is provided for a given attribute.
- If the same entity is declared more than once, the first declaration encountered is binding; an XML processor *may* issue a warning if entities are declared multiple times.

The Java DOM parser does not report any warnings.

## Using an Alias in Unix

Most versions of Unix provide ways to customize the user interface by defining command-line aliases. We use these features to make calling DomCheck for validation easy.

First we package the class files DomCheck.class and MyErrorHandler.class in a jar file.

Create a manifest file *domcheck.mf* as follows.

```
Manifest-Version: 1.0
Created-By: 1.0 (Ken Slonneger)
Main-Class: DomCheck
```

Create a script file *mkJar* for creating the jar.

```
jar vcfm domcheck.jar domcheck.mf DomCheck.class
                                         MyErrorHandler.class
```

Make the script file executable.

```
% chmod 700 mkJar
```

Execute the script file.

```
% mkJar
```

If you do not have a *bin* directory, create it from your home directory (~).

```
% mkdir bin
```

Copy *domcheck.jar* to ~/bin

Add this line to your *.cshrc* or *.tcshrc* file.

```
alias domcheck 'java -jar ~/bin/domcheck.jar -v \!:1'
```

Execute % **source .cshrc** // first time only

To test an XML file.

```
% domcheck phoneD.xml
```

## Public DTDs

A DTD intended for widespread usage is denoted with the keyword PUBLIC and a Formal Public Identifier (FPI) followed by a URI specifying its location if needed.

### Formal Public Identifiers

- Official names of resources such as DTDs.
- Format:  
standardsIndicator//organizationName//DTDname//language

### Standards Indicator

- + Means document has been approved by a standards body such as ISO.
- Means document has no backing from a standards group; W3C is not a standards organization; it only makes recommendations.

### Organization Name

This indicates the name of the organization or company that is responsible for the DTD.

### DTD Name

A descriptive name for the DTD, including a version number.

### Language

A language code, using the ISO 639 list, that specifies which language was used to write the DTD.

## Example: MusicXML

This example can be found in chapter 2 of *XML 1.1 Bible*. It describes the first three measures of a solo piece.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE score-partwise PUBLIC
    "-//Recordare//DTD MusicXML 0.7a Partwise//EN"
    "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise>
  <work>
    <work-title>Flute Swale</work-title>
  </work>
  <identification>
    <creator type="composer">Beth Anderson</creator>
    <rights>&#xA9; 2003 Beth Anderson</rights>
    <encoding>
      <encoding-date>2003-06-21</encoding-date>
      <encoder>Elliote Rusty Harold</encoder>
      <software>jEdit</software>
      <encoding-description>
        Listing 2-7 from the XML Bible, 3rd Edition
      </encoding-description>
    </encoding>
  </identification>
  <part-list>
    <score-part id="P1">
      <part-name>flute</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        :
```

## Testing Validity

```
% xmllint --valid --noout music.xml
%
```

When I first tried to validate this XML document, it came up as not well-formed. I had to edit four tags in the file, changing them from end tags to start tags to make the document well-formed. Then the validation succeeded immediately.

Both the Brown University validator and the Tobin validator reported that *music.xml* is valid.

## Entities in DTDs

An entity definition plays a role similar to a macro definition (*#define* in C) or a constant specification in a programming language. It is one of the four kinds of rules in a DTD.

```
<!ENTITY entName "substitution text">
```

### Advantages

- An entity acts as an alias for commonly used text.
- Repetitive typing can be reduced.
- Items that might change many places in an XML document can be collected into entity definitions in the DTD.

### Entity Values

Parsed entities

- well-formed XML

Unparsed entities

- other forms of text
- binary data

## Entity References

Entities are referred to by names placed between "&" and ";" as with the five predefined entities that we have seen already.

### Example: ent.xml

In this example the DTD is used only to define entities; no validation will be possible.

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE examples [
  <!ENTITY email "slonnegr@cs.uiowa.edu">
  <!ENTITY author "Ken Slonneger">
  <!ENTITY year "2005">
  <!ENTITY copyright "&#169;">
  ]>
<examples>
  <title>Document</title>
  <author>&author;</author>
  <copyright>&copyright; &author; &year;</copyright>
  <email>&email;</email>
</examples>
```

### Expanding the Entities

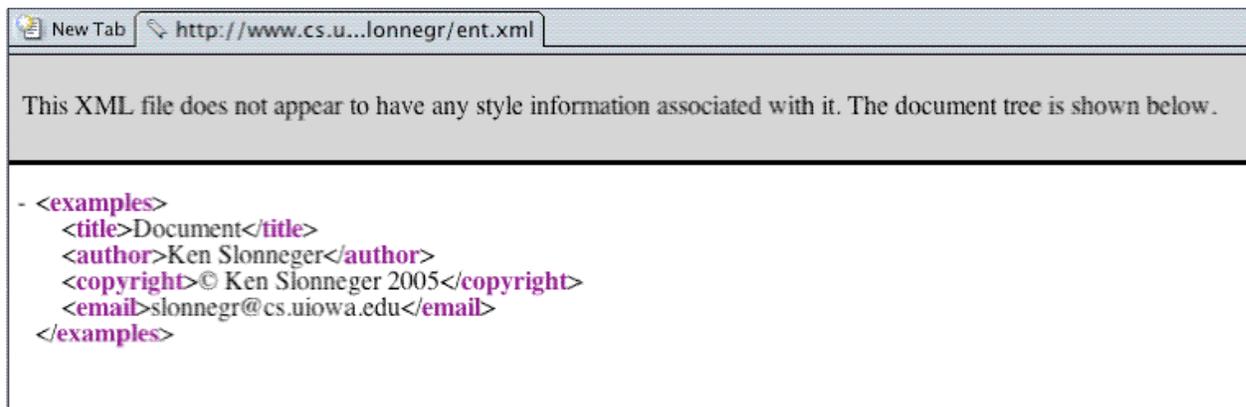
The tool *xmllint* can be used to resolve the entity references in the XML document, but notice that the character reference is not replaced. The flag *noent* mean "no entities".

```
% xmllint --noent ent.xml
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE examples [
  <!ENTITY email "slonnegr@cs.uiowa.edu">
  <!ENTITY author "Ken Slonneger">
  <!ENTITY year "2005">
  <!ENTITY copyright "&#169;">
  ]>
```

```
<examples>
  <title>Document</title>
  <author>Ken Slonneger</author>
  <copyright>&#xA9; Ken Slonneger 2005</copyright>
  <email>slonnegr@cs.uiowa.edu</email>
</examples>
```

Another way to see the entities resolved is to open the document in a browser that parses the XML and shows the result.

This picture is of Netscape Navigator.



## Parameter Entities

The entities covers so far are called *general entities*. They can be referenced both in the DTD and anywhere inside the root element of the XML document. References in the DTD must occur only in other entity definitions or in default attribute values, not in the DTD proper.

A second kind of entity, the *parameter entities*, can be referenced only in the DTD itself. They have the format:

```
<!ENTITY % entName "replacement text">
```

The entity references have the form %entName; for parameter entities.

## Example

Parameter entities seem to require the use of an external subset to get the entities to be expanded properly.

The tool *xmllint* can be used with the option "--noent" to force the resolution of entities.

### DTD File: pent.dtd

```
<!ENTITY % pcd "(#PCDATA)">
<!ENTITY % rgb "(red | green | blue)">
<!ELEMENT example (spot, (%rgb;)+)>
<!ELEMENT red %pcd;>
<!ELEMENT green %pcd;>
<!ELEMENT blue %pcd;>
<!ELEMENT spot %pcd;>
<!ATTLIST spot color %rgb; #IMPLIED>
```

### XML File: pent.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE example SYSTEM "pent.dtd">
<example>
  <spot color="red">A red spot.</spot>
  <blue>153</blue>
  <green>255</green>
</example>
```

## Testing

```
% xmllint --valid --noent pent.xml
```

Without the option "--noent" the validation fails because the entities are not expanded so the DTD is incorrectly written.

## Sharing DTDs

Parameter entities can be used to implement the sharing of DTDs.

Suppose that a company has a number of DTD fragments that need to be included in many of the DTD documents that are created in the company.

One possibility is to distribute the DTDs in files so that individual designers can copy and paste them into their own definitions.

Problem: This approach is not very robust. If any changes are made in the original DTDs, new files must be distributed and each designer will have to edit his or her files to reflect the changes.

Solution: Have only one copy of each of the shared DTD files and have the designers reference these files directly in the DTD documents that they are creating.

### Example: A DTD in Three Parts

In this small example we divide the *phone.dtd* specification into three parts: *name.dtd*, *entry.dtd*, and *phonePE.dtd*.

#### File 1: name.dtd

```
<!ELEMENT name (first, middle?, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
```

## File 2: entry.dtd

```
<!ELEMENT entry (name, phone, city)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ENTITY % name SYSTEM "name.dtd">
%name;
```

## File 3: phonePE.dtd

```
<!ENTITY % entry SYSTEM "entry.dtd">
<!ELEMENT phoneNumbers (title, entries)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT entries (entry*)>
%entry;
```

## XML Document: phonePE.xml

```
<?xml version="1.0"?>
<!DOCTYPE phoneNumbers SYSTEM "phonePE.dtd">
<phoneNumbers>
    :
</phoneNumbers>
```

## Test Validity

```
% xmllint --valid --noout phonePE.xml
%
```

This same basic technique can be used with general entities to build an XML document that has parts in several different files.