XML Basics

Following are some things that you should be aware of before you embark on creating your first PDS4 XML label. This is especially true if you are coming from the PDS3 world, where things like case and keyword order were largely irrelevant; or the HTML world, where closing tags can be treated fairly cavalierly.

Reading and Editing an XML File

If you've never worked with XML before, simply looking at XML can be a bit intimidating - there's a lot of syntax there to read around. Until you get used to the syntax, and even afterwards, it can be helpful to look at XML files in a program that provides some value-added display capabilities. The commonly available options are:

- **XML-aware editors**. Editors like *Eclipse* and *oXygen*, which support XML and schema markup and validation, can offer not only syntax highlighting, but also alternate views of the content - like document trees or expandable graphic layouts. This is your best option if you want to do anything other than browse casually through the file.
- **Editors with syntax highlighting**. If you have a favorite program editor that provides color syntax highlighting, there's a good chance the more recent updates will include options for XML highlighting and possibly more. *UltraEdit*, for example, has an option for showing the document as a tree without the XML markup.
- **Web Browsers**. All the major web browsers (Google Chrome, Internet Explorer, Firefox, and Safari, for example) can display XML files with syntax highlighting and the option of collapsing/expanding code blocks. For Chrome, IE, and Firefox, all you have to do is open the file with the browser. For Safari, you might need to turn on the Development menu first - open your *Preferences* menu, select the *Advanced* tab, and check the *Show Development Menu in the Menu Bar* box.
  If your browser or version doesn't seem to be doing the job (that is, you get a mostly blank page, or you get a page that just looks like a solid block of unformatted text), try the "View Source" option of your browser.
- **Browser Plug-ins**. If your browser allows you to add extended functions via plug-ins, extensions, or apps, try browsing those collections for XML viewers. As of this writing, the selection is limited but growing. The best one I've personally tried so far for browsing through PDS4 labels is the "*XV - XML Viewer*" extension available for the Google Chrome browser. If you've come across a particularly good example, please let us know.

XML Syntax

Case Sensitivity

XML is case-sensitive: `<Begin>`, `<begin>`, and `<BEGIN>` are all different tags to an XML parser. Notwithstanding, if you are defining new tags (as you do when you create a local data dictionary) you should ***never*** use case alone to distinguish two tags.

XML Tags Must Be Closed

Unlike HTML, *all* XML tags *must* be closed. So this is not valid:

```
<start_time>2014-07-04T10:00:00
```

`<stop_time>2014-07-04T21:00:00`

XML Tags Must Be Closed in Order

All XML tags must be opened and closed in strict Last Opened - First Closed order. That is, all tags opened inside one tag must be closed before you close the outside tag. So this is valid:

`<em>This is <strong>OK</strong></em>`

But this is *not*:

`<em>This is <strong>NOT VALID</em></strong>`

Character Restrictions

You ***may not*** use the greater than (>), less than (<), or ampersand (&) characters in your text fields - an XML parser will *always* assume these begin a tag or an entity reference (a stand-in for a character that is not available for one reason or another). Instead, you must use the entity references for these characters:

Use `"<"` for the '<' character.
Use `">"` for the '>' character.
Use `"&"` for the '&' characer.

You must make these substitutions *all the time* in *every text field* where you want to use these characters. So, for example, in a table field description of a PDS4 label you might see something like this:

```
<description>
  This field is set to "-999" when the observed counts are &gt;
10000.
</description>
```

Because this will cause a syntax error:

```
<description>
  This field is set to "-999" when the observed counts are >
10000.
</description>
```

When you are writing code to deal with XML text fields, you may need to remember to decode the entity references before proceeding - depending on the text handling of the XML parser you are using.

End-of-Line Characters

XML does not require a specific form of line break, so you can use whatever is convenient (carriage return, linefeed, or a combination) when creating an XML file. XML parsers will do the right thing largely because they're parsing on tags, not records - so whatever line break you are using is just whitespace to XML.

When writing code to process XML files, if you are using a conformant XML parser all line breaks will be normalized to linefeed characters (unless you specifically prevent this). If you are not using a conformant parser, you will need to read the documentation to determine what whitespace

processing it does on end-of-line characters, if any. In any event, you *will* need to worry about appropriate output carriage control for those tags that should preserve whitespace in their values (mainly description, note, and comment fields), and modify the line breaks accordingly when that matters.

XML Schema (XSD)

XML Schema is Strictly Ordered

The XML Schema definition language (XSD) is strictly ordered. That is, attributes and classes ***must*** appear in the order in which they are defined in the XSD file. While it is possible to circumvent this, it is difficult and it can have a serious negative impact on validation. So unless otherwise indicated, you should assume that you *must* put classes and attributes in the order illustrated.

Note that while schema-aware editors can tell you whether any particular class or attribute is a valid choice, they tend to sort the options alphabetically - so it can be difficult to guess which order attributes should be in for a large class. If you get an error message that an attribute is not valid at a particular place when you know the attribute does belong in the class, then it is almost certainly an ordering error. Check the XSD or the PDS4 Information Model for correct ordering.

XML Primer for PDS4

This page lists the XML standards applicable to PDS4 labels and processing. It provides links to the standards themselves, and some brief overview points about each.

The XML development effort has a design philosophy that is modular in approach. Standards are developed to be useful across contexts, and when a part of a larger standard appears to have broader application than the original context, it is split off into a separate standard for development, so that later work can reference an existing standard and avoid re-defining the wheel. That is why, for example, the *Namespaces in XML* standard is not just part of the *XML 1.0* standard - because the concept of namespace is also applicable in schema files, catalog files, and other types applications.

As a side-effect, though, it seems like you have to know half a dozen different standards to get anything done in XML - thus this summary page.

XML

XML has two official (that is, W3C Recommendation) versions:

- XML 1.0: http://www.w3.org/TR/REC-xml/
- XML 1.1: http://www.w3.org/TR/xml11/

Overview

The XML Standard defines the overall syntax for XML files, which are called *documents* by the XML Standard - that is, PDS4 labels are XML documents. The XML Standard covers the following topics relevant to PDS4 labels:

- Syntax for elements and attributes - the used of '<' and '>' around tag names; the requirement for closing tags; comment and processing instruction format; etc.
- Character set - Allowed characters for names and content (with reference to the Unicode standard)
- The required XML declaration that must appear as the first line of any XML document.

It also includes syntax for writing Document Type Definitions (DTDs), which are used to define the element and attribute names and format constraints. In PDS4, however, we will be using **XML Schema** to do that rather than DTDs.

The XML standard *does not* define the element names themselves. Any application or set of applications planning to make use of XML must either define its own set of element names and associated data types, or use one of the publicly defined systems (like DocBook, which is an XML mark-up language used for creating books and articles).

Version 1.0 vs. Version 1.1

For most purposes within the PDS, the distinction between version 1.0 and 1.1 of the XML standard can be ignored. The major differences are:

- 1.1 expands the allowed character set for names to accommodate expansion of the Unicode standard since version 1.0.
- 1.1 has looser character constraints on names, in anticipation of future expansion of the Unicode standard. Where version 1.0 prohibited everything that wasn't explicitly allowed, version 1.1 allows anything that is not explicitly prohibited.
- 1.1 expands line-end conventions to include Unicode conventions (and a couple others)
- 1.1 defines "full normalization" constraints. These only come into play in the US when working with documents converted from word processing environments where typographic ligatures or letters with diacritical marks might be transcoded as either a single Unicode character or a sequence of the individual characters that must be used to compose the final character. If you don't know what that means and you want to, try this Wikipedia article on "Unicode equivalence" as a starting point.

While that last point is not likely to come up in a PDS4 label context in the US, it may be relevant to international organizations looking to adopt PDS4 standards and tools for local use. In this case, it may well be worth the effort to ensure that all software used is working to the XML 1.1 standard.

XML Schema

The XML Schema Definition Language (XSD) has two official (W3C Recommendation) versions, each of which comes in two parts:

- XML Schema Definition Language 1.0:
- Part 1: Structures
- Part 2: Datatypes

The 1.0 version also has an associated Primer, which should be largely applicable to both versions.

- XML Schema Definition Language 1.1:
- Part 1: Structures
- Part 2: Datatypes

Overview

XML Schema Definition Language (XSD) describes an XML language that can be used to define elements, attributes, and content constraints for a set of XML documents. With XSD you can

effectively create a new XML-based "language" by defining element and attribute names and making constraints and requirements on content and usage.

XSD is used by PDS to define the structure of PDS4 labels and enforce content requirements. If you are creating or editing PDS4 labels, you will need to know how to reference XSD files from the labels; how to use the XSD files to validate your labels; and at some point you will probably want to know how to get structural information out of XSD files so you can see what you can optionally include in your labels. You will probably *not* have to write your own schema files, unless you really want to.

Version 1.0 vs. Version 1.1

Most of the differences between XSD 1.0 and XSD 1.1 would not be visible to an end-user who is simple referencing schemas for writing and validating labels. Because of a number of improvements to validation processing, if you have the option of using XSD 1.1 validation (it might be called "XML Schema 1.1" in your software), it's probably a good idea to use it.

Namespaces in XML

The full W3C Recommendation is available here:

- Namespaces in XML 1.1

Overview

The namespaces standard defines how namespaces are referenced within XML documents. This is the standard that reserves the "*xmlns*" attribute for defining short-hand prefixes for namespaces within XML documents, It also specifies that namespace identifiers must follow the Internationalized Resource Identifiers (IRI) format. PDS uses Uniform Resource Locators (URLs - a subset of IRIs) for namespace identifiers within the PDS system.

Version 1.0 vs. Version 1.1

There is an earlier version of the namespace standard, but for PDS purposes the differences are not significant, apart from errata, the substantive changes from 1.0 to 1.1 are:

- Version 1.1 provides a way to un-declare prefixes.
- Version 1.1 defines namespace names as being IRIs, rather than URIs.

PDS applications are unlikely to ever be so complex as to require un-defining namespace prefixes, and PDS has made a policy decision to use URIs rather than IRIs (the character set constraints are tighter for URIs than IRIs). So Version 1.0 of the namespace standard should be completely sufficient for PDS work.

Other X-standards

In keeping with the modular standards concept, there are other XML standards that are being used and that you might hear referenced, but which you may not have to worry about unless and until you are doing some detailed PDS4 development or validation work that requires them. These include:

XML Catalog

This W3C recommendation defines a special *catalog* file format for use in translating logical references, like URIs, to physical locations. It also defines the method for resolving references according to the information in the catalog file. If you do any serious work with PDS4 labels, you will likely set up an XML catalog file to resolve schema references. You can read the Understanding XML Catalog Files page for some history, some explanation, and some specific advice on the likely most useful parts of the standard for PDS work.

XPath

This W3C recommendation provides syntax for selecting specific tags ("nodes" in the XML-speak of the standard) either by their syntactic relationship to other parts of the document or by their values. This syntax is used in *Schematron* files to find and test various PDS attributes for conditions that are difficult or impossible to dictate via XSD definitions, and also for defining and validating enumerated value lists for attributes that are restricted to specific values. If you are planning to write your own Schematron rules, you will need to become familiar with this standard.

XInclude

This W3C recommendation formally defines a way to include the contents of an external file into an XML file being processed (that is, the traditional programmers "include" concept). This will probably come up in PDS4 eventually, but not for the early builds.

In all these cases, your friendly, neighborhood PDS node consultant should be able to provide you with examples or templates and additional advice.

Understanding XML Catalog Files

XML catalog files use some terminology that can be fairly opaque to those new to XML. Following is an explanation of the key terms used in the XML Catalog standard and their relevance to the PDS4 context.

Identifiers: Public vs. System

The concepts of *public identifier* and *system identifier* predate XML. Both concepts were key in the pre-OASIS world of SGML (the ancestor of XML). These identifiers allowed a document author to reference a file external to his own document. Typically, this would be a Document Type Definition (DTD). DTDs predate schemas, but do the same sort of job - defining the valid content of an SGML file. Standard DTDs were developed to provide interoperability between systems. Perhaps the most widely-known DTD is the DTD that defines the DocBook documentation system.

At this stage of the game, the distinction between *public* and *system* identifiers was clear and simple: The *public identifier* was a globally unique, permanent and invariant identifier assigned to a resource, like the DocBook DTD. The format of the *public identifier* was defined as part of the ISO 8879 (SGML) standard as the Formal Public Identifiers (FPIs) format, and there was (presumably still is) at least one registration authority to assign namespaces to insure that unique FPIs can be formulated by diverse organizations. So the *public identifier* was clearly a logical identification of a resource.

In this regime, the *system identifier* was always a physical location - a reference to a file on disk, for example. The SGML standard required that at least one of the two identifiers was present, but did not require both.

Enter Catalog Files

At this point, the *public identifier* was a logical reference that could not easily be resolved, but at least it was transportable, unlike the *system identifier*. To address this problem, the SGML Open project, which eventually became OASIS, developed the first catalog-type standard (OASIS Technical Resolution 9401:1997) to map *public identifiers* to *system identifiers* in an external ("catalog") file, which could be referenced by applications.

Now, in this pre-XML world, this was a pretty straightforward task. The *public identifier* was always a logical reference, and the *system identifier* was always a physical reference to a locally accessible file. So a DocBook author, for example, could include both types of identifier in his source files as he was preparing them, and when he sent them out into the world the receivers could set their applications to ignore the system identifiers in the document and instead translate the *public identifiers* using their own catalog files. In other words, the application could choose whether the *public* or *system* identifiers should be "preferred" - a term that will come back later with much reduced significance for XML.

Time Passes...

SGML begat XML, the SGML Open group became OASIS Open, and URIs have largely supplanted FPIs. In XML documents, the *public identifier* is optional, while the *system identifier* is usually required (to identify things like name spaces and import files). But in XML, these references are also required to be URIs, which are themselves logical pointers. So in the XML regime, the system identifier **does not** point to a physical location.

OK, it *might* point to a physical location - some URIs do. But in general URIs are **not required to be resolvable in themselves**, so you can't count on someone else's URI being directly resolvable to a physical file. Which is why XML documents may include **schemaLocation** attributes - to indicate the physical location of the files needed to define name spaces or to be imported into the current document.

XML Catalog Standard

OASIS rolled up its sleeves and beefed up the early mapping standard to become the XML Catalog 1.0 standard, to address both SGML and XML mapping needs. The catalog file maps the values of *public identifiers*, *system identifiers* and URIs generally to (other) URIs that actually do resolve to a physical file. It will do this for anything your application considers to be an external id (either a *public identifier* or a *system identifier*), as well as for any other URIs it encounters. A few things to keep in mind when reading/writing catalog entries:

- The XML Catalog standard explicitly states that the first matching line is the one applied - anything else will be ignored. When you are writing your translation elements, put the most specific matches first, and the more general matches later. For example, if you are trying to match a URI that ends in a file name, put that element **before** any element that matches just the path.
- Applications can choose to be more or less picky about URI formatting in your catalog files. According to the XML Catalog standard, catalog processors must normalize URIs before running a comparison, but some processors may be more liberal in what they'll recognize and translate for you than others if, for example, you use local OS path syntax rather than the Unix-like syntax technically required by the "file:" protocol. The oXygen editor, for example, is fairly lenient about URI formatting in the catalog file. Other applications may not be so forgiving.
- One of the consequences of the evolution from DTD and external (public/system) identifiers to XML and URIs is that the distinction between public and system identifiers is largely moot. The external identifiers in our PDS XML documents - the references to the XML Schema and

XML Schema-Instance name spaces, for example - are not required to have system identifiers (the definitions are "built-in", as it were). Since everything else falls under the "URI" rubric, our XML Catalog files tend to contain only URI-type mappings.

- As a result, applications may be more or less lenient about discriminating between public/system identifiers and general URIs when matching strings and applying mappings. So for some applications, using a *system identifier* mapping rather than a URI mapping will still translate all occurrences of the matching URI, even if it technically isn't being used as a *system identifier*.
- It is possible to write complex catalog files, with elements for including additional files or branching from one catalog file to another. Most PDS data preparers and users do not need any of those complications. The standard set-up and a few simple URI mapping parameters will do the job for most of us.
- Catalog files are not transportable. They are the epitome of environment-specific configuration. When following someone else's example, be particularly careful about the file specification URIs you will be translating to - they will depend critically on your local file system.

XML Catalog File Elements

Here is what you need to know to write or edit an XML Catalog file.

Every catalog file will begin with the usual <?xml> tag and possibly a <!DOCTYPE> declaration (some applications require it, some forbid it), followed by the <catalog> tag which begins the catalog information proper and identifies the namespace associated with the XML Catalog standard. These can be copied verbatim from any valid catalog file; if you use an XML Catalog generation tool, these will be provided for you. The <catalog> tag may have a prefer attribute with a value of either "public" or "system". As explained above, for PDS purposes this preference setting is meaningless – we will only be mapping URIs, not external identifiers.

Between the <catalog> and </catalog> tags, these are the tags that will likely be most useful and most common in catalog files supporting PDS labels:

<uri name="*name_string*" uri="*physical_reference*"/>

The <uri> element does a straight one-to-one mapping from the URI given as the value of "name" to the URI given as the value of "uri". So *name_string* is what appears in the XML file, and *physical_reference* is the actual location of the file that contains the answer (the namespace definition, the XML fragment to be included, etc.). This **must** be resolvable. For most of our users this will resolve to a file on the local file system, so it will begin with the string "file:///". It could also resolve to a web location, in which case it will likely begin with something like "http:" or "ftp:". The URIs should both be URI-encoded, for safety.

<rewriteURI uriStartString="*old_prefix*" rewritePrefix="*new_prefix*"/>

The <rewriteURI> element can be used to map many URIs at once, based on a common initial substring in those URIs. For example, say you have reproduced the PDS schema directories in a local repository. You could then map all your PDS namespace references at once by replacing the "http://pds.nasa.gov/pds4" part of every namespace URI with a reference to the root directory of your schema repository. As with the <uri> element, the URI created must be resolvable. *Old_prefix* is the prefix as it appears in the XML file; *new_prefix* is the replacement that turns that string into a resolvable reference.

<uriSuffix uriSuffix="*uri_suffix*" uri="*physical_reference*"/>

The <uriSuffix> element matches based on the *end* of the URI string - so if the URI in the XML document ends in *uri_suffix*, then the entire URI is mapped to the *physical_reference* (which must, of

course, be resolvable). Note that this is not at all like <rewriteURI>, which effectively does a string substitution on the URI from the XML document. <uriSuffix> matches based on the suffix only, but then expects to map this to a complete, new URI. (One of the few differences between the XML Catalog 1.0 and 1.1 standards is the addition of this element in the 1.1 standard.)

<delegateURI uriStartString="*prefix_string*" catalog="*physical_reference*"/>

The <delegateURI> element lets you hand off URI translation for a set of URIs to a different catalog file. This can be useful if you are working in a fairly complex environment where some of your URI translations are stable and some aren't (or some are in production mode and others in development). This could also be used to set up a hierarchy of public and private XML catalogs. When a URI in the XML document starts with the *prefix_string*, the URI will be immediately handed off to the catalog file indicated by the *physical_reference* for processing. (Note, though, that the catalog processing will stop at the *first match encountered*, so take care with where you locate your delegate element.)

There are analogous elements to the above for mapping *public identifiers* and *system identifiers*, as well as a <group> element for providing default preferences and base URIs for these elements, and a <nextCatalog> element for explicitly passing control to another catalog file (rather than letting your application work through a predefined list). In addition, all the elements listed above will take an xml:base attribute to specify a base URI, so that relative URIs can be turned into absolute URIs. For most PDS uses, where all required URIs are also required to be absolute and the public/system preference is not applicable, these are not necessary. If you think you might need or want them, read the standard carefully and try it.

Some Simple Examples

Following are some simple XML Catalog files for a couple of common scenarios. Note that these all contain both the *DOCTYPE* reference and the catalog namespace reference ("urn:oasis:names:tc:entity:xmlns:xml:catalog"). Some PDS4 tools may choke on the *DOCTYPE* directive; it can be removed as long as the namespace reference remains in the <catalog> statement.

rewriteURI

This catalog file uses a single rewriteURI statement to map all PDS4 namespace schema references to a copy of the schema tree on a local (NFS-mounted) directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.1//EN" "http://www.oasis-
open.org/committees/entity/release/1.1/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <rewriteURI uriStartString="http://pds.nasa.gov/pds4"
              rewritePrefix="file:///n/sbnops/lcltools/schema"/>
</catalog>
```

So, for example, a reference to the schema URI "httpd://pds.nasa.gov/pds4/pds/v1/PDS4_PDS_1301.xsd" will be translated to the local file reference "/n/sbnops/lcltools/schema/pds/v1/PDS4_PDS_1301.xsd".

uri

This catalog file adds uri statements **before** the rewriteURI statement to catch references to mission (EPOXI) schema files still in local development. The uri statements have to come first because the catalog processor will stop with the first statement that matches - so in this case if the rewriteURI statement came first, the processor would never make it past there.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.1//EN" "http://www.oasis-open.org/committees/entity/release/1.1/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <uri name="http://pds.nasa.gov/pds4/mission/epoxi/v1/EPOXI_1-0.xsd"
       uri="file:///home/raugh/Oxygen/epoxiDD/epoxi_draft.xsd"/>
  <uri name="http://pds.nasa.gov/pds4/mission/epoxi/v1/EPOXI_1-0.sch"
       uri="file:///home/raugh/Oxygen/epoxiDD/epoxi_draft.sch"/>
  <rewriteURI uriStartString="http://pds.nasa.gov/pds4"
              rewritePrefix="file:///n/sbnops/lcltools/schema"/>
</catalog>
```

The uri statement replaces the entire matched name with the associated URI value, so the string "http://pds.nasa.gov/pds4/mission/epoxi/v1/EPOXI_1-0.sch", for example, will be replaced by the reference to the local file "/home/raugh/Oxygen/epoxiDD/epoxi_draft.sch".

delegateURI

Alternately, if you are working with several mission dictionaries in active development scattered across your disc space or network, you might want to use a catalog file specifically to handle the mission dictionaries and use your local schema tree for the rest. In that case, you would use a delegateURI statement to trap references to all mission namespaces and pass them off to a different catalog file, while the rest fall through to be handled by the rewriteURI:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.1//EN" "http://www.oasis-open.org/committees/entity/release/1.1/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <delegateURI uriStartString="http://pds.nasa.gov/pds4/mission"
               catalog="file:///home/raugh/Oxygen/XMLCatalogs/mission_schemas.xml"/>
  <rewriteURI uriStartString="http://pds.nasa.gov/pds4"
              rewritePrefix="file:///n/sbnops/lcltools/schema"/>
</catalog>
```

In this case, all references beginning with "http://pds.nasa.gov/pds4/mission" will be passed to the "mission_schemas.xml" catalog file for resolution. Say that catalog file looks like this:

*Contents of **mission_schemas.xml**:*

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.1//EN" "http://www.oasis-
open.org/committees/entity/release/1.1/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
<rewriteURI uriStartString="http://pds.nasa.gov/pds4/mission/epoxi/v1"
rewritePrefix="file:///home/raugh/Oxygen/epoxiDD"/>
<rewriteURI uriStartString="http://pds.nasa.gov/pds4"
rewritePrefix="file:///n/sbnops/lcltools/schema"/>
</catalog>
```

Given these two catalog files (in their proper places, of course):

- a reference to "http://pds.nasa.gov/pds4/mission/epoxi/v1/EPOXI_1-0.xsd" will be passed to *mission_schemas.xml*, which will return a new value of "file:///home/raugh/Oxygen/epoxiDD/EXPOXI_1-0.xsd";
- a reference to "http://pds.nasa.gov/pds4/mission/di/v1/DEEP_IMPACT_1-1.xsd" will also be passed to *mission_schemas.xml*, but will return a new value of "file:///n/sbnops/lcltools/schema/mission/di/v1/DEEP_IMPACT_1-1.xsd"; and
- a reference to "http://pds.nasa.gov/pds4/pds/v1/PDS4_PDS_1300.xsd" will be handled by the first catalog file, and will return a value of "file:///n/sbnops/lcltools/schema/pds/v1/PDS4_PDS_1300.xsd".

Anatomy of the XML Prolog

*This page was updated in June 2016 with information on the **schematypens** attribute.*

The *prolog* of an XML document comprises everything from the start of the file to the document root tag. It may contain the XML declaration, processing instructions, comments, and a document type definition. In XML 1.0, all these things are optional; in XML 1.1 the XML declaration is required.

All PDS4 labels will contain both an XML declaration, required by PDS, as well as at least one processing instruction, as it is processing instructions that create the connections to the Schematron parts of namespace definitions. In fact, PDS4 labels will, in general, have one processing instruction for each PDS-controlled namespace referenced in the label.

Example PDS4 Label Prolog

Here's a sample prolog from an early prototype label that references the PDS4 core namespace, four discipline dictionaries, and a mission dictionary:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="http://pds.nasa.gov/pds4/pds/v1/PDS4_PDS_1201.sch" schematypens="http://
purl.oclc.org/dsdl/schematron"?>
<?xml-model href="http://pds.nasa.gov/pds4/disp/v1/PDS4_DISP_1100.sch" schematypens="http://
purl.oclc.org/dsdl/schematron"?>
<?xml-model href="http://pds.nasa.gov/pds4/sp/v1/PDS4_SP_1100.sch" schematypens="http://
purl.oclc.org/dsdl/schematron"?>
```

```
  <?xml-model href="http://pds.nasa.gov/pds4/geom/v0/PDS4_GEOM_0520.sch"
schematypens="http://purl.oclc.org/dsdl/schematron"?>
  <?xml-model href="http://pds.nasa.gov/pds4/sbn/v0/sbnDD_0100.sch" schematypens="http://
purl.oclc.org/dsdl/schematron"?>
  <?xml-model href="http://pds.nasa.gov/pds4/mission/epoxi/v0/epoxiDD_0100.sch"
schematypens="http://purl.oclc.org/dsdl/schematron"?>
```

We will examine this piece by piece.

XML Declaration

The first line is the XML declaration. It defines the XML standard version the label adheres to, and also defines the character set to be used. It has the format of a processing instruction, but very specific content requirements. It *must* be the very first thing in the file - not even white space may precede it. The XML declaration in our example prolog is:

```
  <?xml version="1.0" encoding="UTF-8"?>
```

Here's what's going on:

version="1.0"

Version number is required in your XML declaration. This one declares that the label is following the W3C XML recommendation version 1.0. XML parsers will assume version 1.0 if they get a document without an XML declaration, but PDS will require that you include this statement not only for the XML version, but for the character set which follows it. For PDS4 purposes, the version could also equally well be "1.1". Also, you can use single quotes around the version number rather than double quotes, if you prefer. Which quote style you choose is not significant, and it can vary through the label.

encoding="UTF-8"

You must also specify which character encoding standard you will be using in the label. The default value stuck in here by various label generators will depend on your software. For PDS4 purposes, you should be using "UTF-8". (The XML standard also requires that all conformant software implement UTF-8 support.) Simply changing the value in the XML declaration, however, will likely not cause your label editing software to start using a different encoding. You will need to search through your preferences to change that.

The valid values than might appear here are defined by the IANA Official Names for Character Sets standard. Common values you might see include:

- "ISO-8859-1" - This is the single-byte "Latin" codepage that maps the first 256 Unicode characters, which in turn include the 127 ASCII characters, to a single-byte value. *It is **not** equivalent to either* US-ASCII *or* UTF-8 *for characters beyond the 127 ASCII characters.*
- "ISO-8859-*x*" - The related ISO-8859-* code pages contain characters from non-English alphabets in the higher (above 127) locations. There may be non-English characters in common among these code pages, but they will likely appear in different places in the different code pages.
- "US-ASCII" - Once again, the first 127 bytes correspond to the ASCII character set, but anything beyond that may vary from other encodings, both in content and position.

  In all these cases, if you *know* there are no bytes with values greater than 127 then you can change the `encoding` value to "UTF-8". But if there are any higher-order characters in the file

you will need to convert the file to UTF-8 prior to archiving. (Some editors can do this as a "Save-as" function; some treat it as a font-related conversion.)

In addition, you might also see a ***standalone*** *declaration* in an XML declaration. It would look like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

A standalone value of "yes" would indicate that everything you need to know about what tags are used in the document is included inside the document. This will never happen in PDS4 labels (the markup is defined by the XSD and SCH schema files external to the label and referenced elsewhere), so if you see a standalone attribute in an XML declaration in a PDS4 label it needs to have a value of "no", which is also the default. You *may* see a value of "yes" in an XML file submitted as an archive product, since any XML files included in the archive will require some sort of formal document structure definition, and in some cases it might be convenient to include it as a *Document Type Definition* (*DTD*) inside the XML file rather than as a separate DTD or schema file.

Finally, white space not inside quotes is not significant in your XML declaration. White space includes blanks, tabs, and line breaks. This would also be valid:

```
<?xml
  version="1.0"
  encoding="UTF-8"
  standalone="no"?>
```

xml-model Processing Instruction

Processing instructions are delimited by the character pairs **<?** and **?>** (same as for the XML declaration). The xml-model processing instruction is the focus of a relatively new (first proposed in 2010; last revised 2012) W3C standard "Associating Schemas with XML Documents". It exists to provide an explicit link between an XML document and the schema that define(s) its valid content. The Schema Referencing in PDS4 Labels page provides complete documentation and instructions on formulating <?xml-model?> processing instructions for PDS4 labels.

Here's what's going on:

href="http://pds.nasa.gov/pds4/sbn/v0/sbnDD_0100.sch"

The *href* attribute points to the location of a schema that defines the document content. In the PDS4 case, this will be a Schematron file (there are other types of schema files that could be referenced by an *xml-model* instruction, but PDS is not making use of them). Depending on your processing environment, the value of *href* could be a resolvable URL, a local file reference, or a URI that can be resolved to a physical file by XML Catalog processing.

schematypens="http://purl.oclc.org/dsdl/schematron"

The *schematypens* (read "schema type namespace") attribute indicates which schema language is used in the file pointed to by the *href* attribute. There are several different flavors of the Schematron standard, so for PDS4 archive files it is important to specify the ISO Schematron version. The URI in the examples above corresponds to the ISO Schematron URI.

Only one schema reference can be made in each <?xml-model?> instruction, but you may have multiple instructions. A PDS4 label should contain one <?xml-model?> instruction for each dictionary used in the label.

As with the XML declaration, white space is not significant between the parts of the *xml-model* instruction.

Other Prolog Elements

The only other prolog components you should ever find in a PDS4 label would be white space (blank lines) and XML comments (delimited by **<!--** and **-->**). Neither of these is required, of course.

In XML document files inside the archive, you may find a *Document Type Definition* (*DTD*) declaration. A DTD declaration will open with "<!DOCTYPE", and may consist of a reference to an external definition (perhaps a standard DTD like the *DocBook* DTD), or a series of type definitions statements for elements, attributes, and all the other sorts of things that PDS uses XML Schema files to define.

References

Here are some links to the various standards mentioned above:

- XML Catalogs V1.0, October 2002
- XML Catalogs V1.1, October 2005
- OASIS Technical Resolution 9401:1997 (pre-XML catalogs)
- Extensible Markup Language (XML) 1.0 (Fifth Edition)
- XML Schema Definition Language (XSD) 1.1 Part 1: Structures