DITA for Solo Writers

Lone-DITA
2007 Louise Kasemeier

Contents

Introduction

There are many introductory documents concerning DITA in general. Good places to start are listed at the end of this topic.

Moving from traditional unstructured documentation, such as *Framemaker* or *Word* to structured documentation using DITA involves a shift in how you think about the information in your documents as well as an ability and desire to learn new technologies.

In some cases, structured documentation will offer little or no benefit to a small team or solo writer. But there is an increasing need to provide modular documentation, documentation that is customized to the needs a particular user or group of users and documentation in a variety of online and paper formats. These requirements can be particularly difficult for small teams to meet. The initial investment in time and effort is repaid by a much more streamlined method for producing useful, targeted documentation for users.

These documents are the results of my own efforts to implement a DITA solution for a solo writer with a budget of virtually nothing. It can be done! I should make it clear that I am not recommending that everyone should forgo the expert advice of a professional consultant (I would have used one if I could!). But if, like me, that is not an option for you, then I hope that this guide will be useful.

Benefits of Switching to DITA

Single Source -> Multiple Formats	DITA allows you to produce documentation in multiple formats from a single set of XML source files. The current open source toolkit provides support for HTML, HTML Help, JAVA Help, Eclipse Help and PDF output. DITA eliminates the need to maintain multiple version of the same documents; a process which can be both error-prone and time consuming.
Reuse Content	Content can be reused within and across topics and output formats, making updating documents easier and reducing development time.
Multiple Versions	DITA conditional processing provides a mechanism to produce multiple versions of a document from the same source. Documents can be created for spefic audience types, platforms, product versions and more, without needing to maintain multiple sets of content.
Separation of Content and Formatting	Formatting is applied according to your style guidelines by stylesheets, freeing the writer to concentrate on writing content.
Consistency	The DITA DTD assists you in maintaining information that is consistently and logically formatted and structured across a document set.
Vocabulary	DITA specialization architecture allows you to use a vocabulary for your XML that is meaningful to other writers.

Getting Started

So how do you go from *DITA Open Toolkit* and blank page to multiple documents in multiple formats? Here is the basic workflow:

1. Create your content in XML files. The XML tags you use will conform to the specification in the DITA DTD provided in the *DITA Open Toolkit*.
2. Create a map file defining which topics will be included in your output document, the order in which they will appear and their hierarchy.
3. Create stylesheets to define the layout and formatting of each output type (PDF, HTML etc) or modify those provided with the *DITA Open Toolkit*.
4. Create a filter file if required. A filter file specifies that blocks of content or groups of map entries marked with a particular attribute/value pair will be excluded or flagged. I.e exclude all elements with audience='administrators'.
5. Using ANT and a build file that specifies which DITA map file and filter file to use, which output directory to create the docs in and which 'transtype' (output format - PDF, HTML, CHM etc) to use, transform and process/compile the documents.

This workflow assumes that you have already completed planning tasks which include:

- Plan your content model. This is the process of determining how you will decide which topics or blocks of content will be included in your document sets and how content will be re-used. Topics and smaller chunks of content can be re-used throughout a DITA documentation set through DITA maps used to assemble topics and through the content inclusion mechanism. This mechanism allows you to include a re-usable block of content in the same or across different topics. You can also take the filtered re-use approcah. This method uses selection attributes to select which content should be included or excluded. For example, the audience attribute could be used to exclude all content that is specifically for expert users.
- Plan how you will manage your content. You will need to implement some method of keeping track of your topics and your metadata schemes. I use a combination of file system and a simple *MS Access* database to keep track of my content but you may want to implement your own method, such as a spreadsheet. Specialized content management software is extremely expensive for small team/solo writers.
- Set up your stylesheets according to your own requirements. This may be very simple, such as editing the CSS files for HTML and CHM outputs or it could be more complicated and entail actually modifying the XSL files.
- Create any specializations that you need for topic types, map types, domains and attributes.

Related Technologies

The technologies that you will need to learn in order to produce DITA documentation depend to some extent on the types of output that you want to produce. If you are producing HTML or CHM files, chances are that you are already familiar with those technologies. If you are producing PDF files, it is by no means likely that you are already familiar with XSL-FO since this is not necessary to produce PDFs from *Framemaker* or similar applications.

For each technology listed below, I have included a link to the appropriate *W3schools* tutorial. These tutorials provide a good introduction and starting point for beginners.

Technologies that anyone embarking on a DITA project should familiarize themselves with thoroughly are:

- XML: XML tags are used to structure and tag your content. XML is the fundamental technology in DITA and you should have a good understanding of it.

- XSLT: XSLT is used to transform XML into a different format such as HTML or XSL-FO. XSLT uses an additional technology called XPATH to locate elements and attributes in the XML document.
- DTD/XSD: DTD or Document Type Definition is used to define the legal structure of an XML document. Since XML itself imposes no restrictions on the elements and attributes you can use in a topic, a DTD is required to ensure that your XML adheres to the DITA standard. Without this check in place, the stylesheets defined for use with DITA documents will not work, because your XML will not be DITA compliant. Most XML editing tools will validate your XML against the DTD for you provided you have referenced the DTD correctly in your XML.

For PDF output using Idiom's FO plug-in:

- XSL-FO: FO stands for Formatting Objects and is a standard used to define layouts for PDF documents. FO is transformed into PDF by a FO processor such as XEP by RenderX (this is the processor used by the Idiom FO plug-in).

For HTML-based outputs such as HTML or Microsoft HTML Help:

- HTML: Standard HTML mark up for web pages, HTML help files etc.
- CSS: Cascading Stylesheets are used to apply consistent styles to HTML documents. Styles can be associated to a specific HTML tag and are applied to all instances of that tag or named styles can be created and applied individually to HTML tags using the class attribute.

DITA Toolkit Introduction

The *DITA Open Toolkit* is at the core of any system for producing structured documentation using DITA. The toolkit contains the following items:

- DITA DTD (Document Type Definition)

  The DTD defines the DITA specification and is used to create and validate DITA content and map files. Its is made up of a set of core .dtd and .mod files and several 'domains' which specialize and extend DITA. This architecture allows content creators to further specialize and extend to meet their own requirements if necessary.

  The DTD is located in the <*dita_dir*> /dtd directory
- XSL Stylesheets for major output formats, including HTML, PDF, Microsoft HTML Help, JavaHelp, Eclipse

  XSL stylesheets are used to transform your XML content file into a suitable format for output or further processing by the Microsoft Help Compiler or an FO Processor for example. Stylesheets also control the layout and formatting of the documentation.

  The XSL stylesheets are located in the <*dita_dir*> /xsl directory
- DITA Documentation

  dita-readme - basic information on configuring and using the *DITA Open Toolkit.*, release notes and some troubleshooting info.

  ditaref-book - DITA element and properties reference - This details every element and property in the DITA specification. It is an invaluable resource for understanding the structure of DITA content.

  installguide/installing-dita - Installation instructions for Windows and Linux. Also provides info on installing ANT, XSLT Processors, Java and other components used by the toolkit that must be installed separately.
- ANT build file templates for all output formats

  ANT is used to conduct the process of building a document in a specified format from a set of DITA content files. The ANT scripts ensure that the proper stylesheets are applied and that the correct compilers/processors etc are invoked to generate the final output document. A single simple build file is used to launch the process and you only need to edit a few simple parameters, such as the input map file, output directory and 'transtype' (output format i.e PDF, HTML Help etc)
- dost.jar

  A set of java classes used by the *Open Toolkit* when building output.
- Samples and Demonstration Files

  A set of content files and maps that you can build and experiment with to familiarize yourself with DITA.

Download the toolkit from *dita-ot.sourceforge.net*

Additional Components Needed to Use the DITA Toolkit

In addition to the DITA Toolkit itself, you will need to download, install and configure several additional components. All of these are free.

- JAVA

  You will need to install J2SE Java Development Kit 1.4.2_08. This is **NOT** the latest version but this is the version specified in the installation instructions, so be sure to get the correct one.

Note: You need the JDK (Developer's Kit) not the Runtime (JRE)

Download the correct version of JAVA from *http://java.sun.com/products/archive/j2se/1.4.2_08/index.html*

After Installation you will need to set the JAVA_HOME environment variable.

- ANT

  ANT is an open source tool that is used to declare a sequence of build actions. ANT is used in the DITA process to apply XSL stylesheets in the correct order and to compile or process the results into the required output format. Although you can invoke DITA transforms from the command line, the ANT method is much easier and more streamlined once you have learned the basics of the ANT syntax. In fact you will rarely need to modify the main ANT scripts. You will be able to launch the document build process by modifying the relevant short build file provided in the *ant* directory in the Toolkit.

  Download ANT from *http://ant.apache.org/bindownload.cgi*

  You will need to set the ANT_HOME environment variable and also add the bin directory to the PATH environment variable.

- Idiom FO Plugin

  Since version 1.2, the DITA architecture has supported 'plug ins' which allows third party developers to extend the toolkit functionality. The toolkit comes with a pdf transform using the FOP processor. However, Idiom generously released their own PDF transform as a free plug-in. I cannot recommend enough that you use this plug-in rather than the default PDF transform. The PDF output is of a much more professional standard and customization of the output is easier. The major disadvantage is that you will almost certainly end up having to purchase the XEP processor from RenderX since the free version places a small footer advertising the product on each page of your document. Its more than worth the $300.

  The Idiom FO plug-in installation and set up process includes the installation of SAXON 6.5 as a part of the XEP install, so there is no need perform a separate installation of an XSLT processor as described in the DITA Toolkit Installation document.

  After unzipping the plug-in, you will also need to install and download the XEP processor from RenderX and the ICU4J library.

  http://sourceforge.net/project/showfiles.php?group_id=132728

  Any references to PDF output in this document refer to the Idiom FO output.

Creating DITA Files

DITA content can be created using any good XML editor. The most useful feature of any editor is the ability to validate against the DITA DTD. Another extremely useful feature is a menu or list of valid tags that you can choose from at any point during the creation or editing of a topic.

Even if you choose an editor that gives you a formatted view of your content as you edit it, you will still need to be familiar with XML, structured document concepts and the general structure of DITA documents. Otherwise you will find yourself becoming frustrated by the seemingly illogical procedures for adding tags and content.

You can choose to combine multiple topics in a single XML file by using the <dita> tag as the root tag and creating topics as its children. I prefer to create a single topic per XML file. This can be helpful when you don't have an expensive content management system, since you can include useful information in the filename such as the topic type and id i.e concept_creating_content.dita as a first step in managing your content. I find this invaluable when I am building my ditamaps. It also helps to maintain a 'topic-based' approach to your content rather than 'book-based' which will allow you to take fuller advantage of DITA content reuse and conditional processing functionality.

You will need to plan the directory structure of your content files, map files and build scripts. Once again, the file system can be a useful tool for content management for smaller DITA implementations. Create different directories for content for different products or other major category and another for common content that will be shared across many docs. You will also need to plan locations for output for different documents and formats.

The diagram below shows the directory structure that I use for my content. In keeping with the content reuse approach, I keep my map files separate from my content files. A map could include content from multiple content areas/directories to produce a variety of documents.

Aim to keep your content files in a sub-directory of your maps directory. If you have a path which contains "../" in topicrefs in a map file, you can end up with additional directories in your output directory.

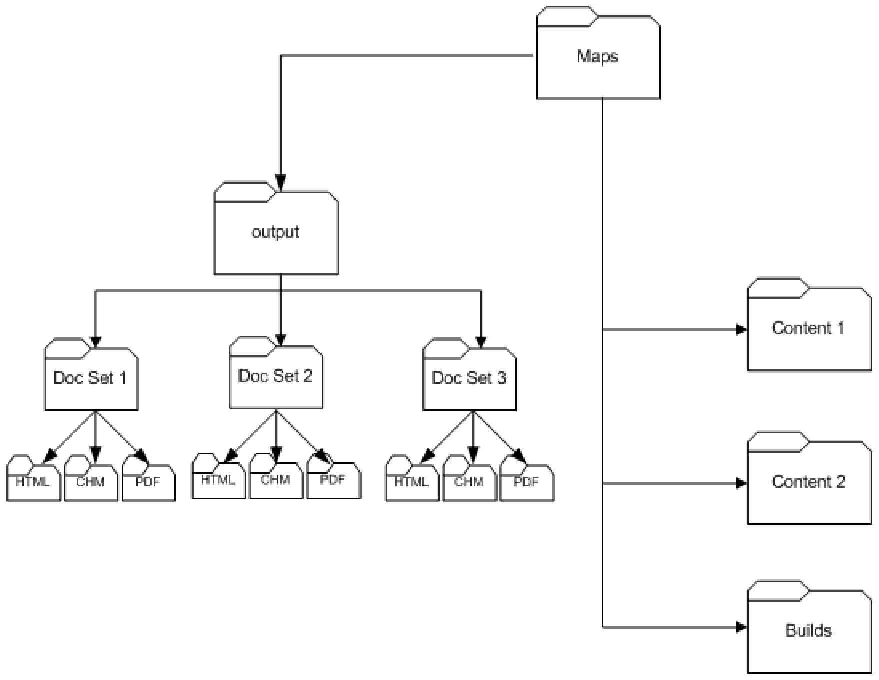


Figure 1: Example DITA content directory structure

The *DITA Open Toolkit* uses a standard XML catalog resolver. This means that you do not need to change the DTD reference in your content files when you change their path or create files in a new location. An XML catalog maps a location to a public identifier. The public identifier is referenced in the DOCTYPE declaration in your content files and allows the DTD location to be resolved.

XML Editing Tools

I use three free tools to produce my DITA content. Each of the three has features that are useful under different circumstances and for different purposes.

XMLMind

XMLMind is a very nice free XML editor which has the added advantage of a free DITA plug-in that provides user-friendly templates for standard DITA topic types and for DITA map files. The templates allow you to creat content in a semi-wysiwyg environment. The XML source is hidden and the content is formatted to make it easier to read and work with. I find that the application has a lot of useful features and have had no reliability issues with it so far. With the added benefit of DITA templates, this editor is about the best free option I have found.

Its only major downfall is the lack of an actual XML source viewer/editor. Sometimes you just have to get your hands dirty and deal with code. This is especially true if you want to start digging into the DTD and stylesheets or even attempt some specialization.

Microsoft Visual Studio Express Web Developer

In fact any of the free *Visual Studio* tools from Microsoft has the XML editor feature. This provides a good interface for working with the actual source code itself. A nice feature is the *Intellisense* which provides context menus of valid tags as you type. It also has good validating and code formatting options including outlining, allowing you to collapse and expand sections of the XML. It is useful when you need to edit the source code, such as when you are writing DTD for specialization or custom XSLT templates.

The code editor in Visual Studio Express also has a very useful **Search/Replace in Files** function. This allows you to search all files in a pre-defined directory or set of directories. This is invaluable when you have lots of XML files for containing your DITA content.

All the *Microsoft Visual Studio Express* editions are available for free download from www.microsoft.com.

Cooktop

Cooktop is another XML source editor/viewer. I use it mainly for its XSL transform function which allows you to apply any XSL stylesheet to any XML file and view the results using a variety of XSL processors, including SAXON and various versions of MSXML. This is useful for customizing and experimenting with DITA XSL files. The only real problem I have found with it is a tendency to insert illegal whitespace when using the **Format XML** function.

Cooktop also has an XSL debugging function which allows you to test XSL code against an XML file and view the results.

Creating DITA Content with XMLMind

XMLMind is a free XML editor that has a DITA template add-on. This makes DITA authoring considerably easier than in a standard shareware XML editor. After installing the DITA add-on, the **New** option on the **File** menu will give you the option to create a concept, task, reference or general topic or a dita map file. *XMLMind* will provide you with an appropriate template to create your topic, with areas for your title, body, task steps, reference sections etc as appropriate.

XMLMind is simple to use and has a ton of features. Its worth spending some time learning the various node selection and editing features and shortcuts. The DITA Addon also gives you a toolbar that makes it easier to highlight content and insert common elements.

Finally, *XMLMind* is customizable If you create specializations for DITA, you can configure *XMLMind* to work with them in the same way as it works with concept, task and reference topics.

Tutorial Introduction

This tutorial will walk you through each stage in the process of producing DITA documentation. At the end of the tutorial you should be able to output a variety of document sets describing how to create a thanksgiving dinner!

Documentation Goals

We will establish several requirements for our Thanksgiving Dinner documentation before we start:

- The documentation should be output as a set of HTML pages and also as a PDF document.
- We should be able to produce documents tailored to different audiences. The different types of audience that we are concerned with are:
  - General
  - Low Cholesterol Eaters
- We should also be able to produce cooking instructions that are aimed at different levels of culinary ability:
  - Novice
  - Experienced

We will need to create a scheme for marking up the content for different audiences depending on their dietary requirements and also according to the difficulty of the recipe.

Tutorial Requirements

The tutorial will guide you through the process of downloading and installing the *DITA Open Toolkit* and Idiom FO plug-in for PDF output and their supporting applications. You will need a PC capable of running these and an internet connection for downloading them. This tutorial was developed and testing using a WindowsXP installation of the *Open Toolkit* version 1.3.1.

Editor specific instruction in the content development section refer to *XMLMind*. However, if you are using another editor, the basics of content creation will still apply. You will just need to modify the specifics of implementation to suit the editing tool you are using. Make sure that you have downloaded and installed the DITA plugin for *XMLMind* before you begin. You can do this using the **Install Add-ons** function from the **Options** menu in *XMLMind*.

All files and scripts that are created in the course of this tutorial can be downloaded as a single zip file from HERE.

How To Install the Toolkit

1. Download and install the *Java Development Kit 1.4.2*: *http://java.sun.com/j2se/1.4.2/download.html*

   This is not the latest version. Make sure that you are downloading the correct one.. Also be sure to download the SDK using the **Download J2SE SDK** link, not the JRE (Java Runtime).
2. Download and install the *Microsoft HTML Help Workshop* : *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/hwMicrosoftHTMLHelpDownloads.asp*

Install the *HTML Help Workshop* in the default location. Otherwise you will need to edit the hhc.dir property in the *build_init.xml* file in the DITA root directory.

You can also install *Javahelp* or other applications and tools needed for output formats that you want to use but we will not be using them in this tutorial.

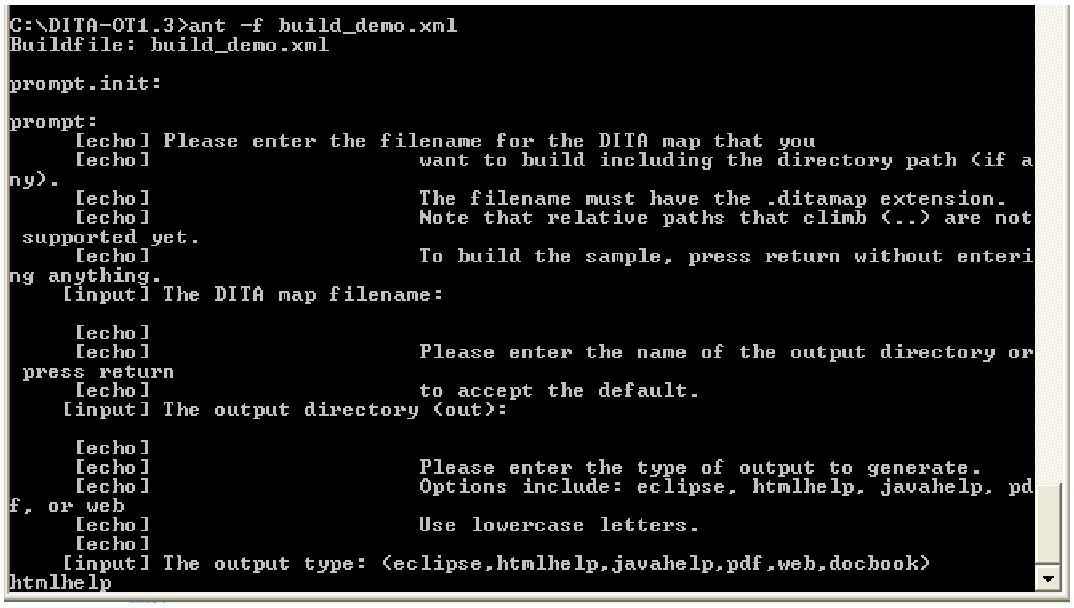
3. Download the latest version of the *DITA Open Toolkit*: *http://sourceforge.net/project/showfiles.php?group_id=132728*

   These instructions apply to version 1.3.1. This version has a greatly improved and streamlined installation and configuration procedure over previous versions. Be sure to download the full package. The file name is *DITA-OT1.3.1_fullpackage_bin.zip*.
4. Whilst you are there, download the latest version of the Idiom FO plug-in from the **Plugins : Transforms** section. You will need it later.
5. Unzip the Toolkit into the default directory. This will be C:\DITA-OT1.3 or similar.
6. Run the batch file *startcmd.bat* from the DITA root directory to set up the necessary environment variables.

   Tip: Using the procedure above and DITA Toolkit 1.3 or later, all environment variables should be set correctly. You should not have to set any of them manually.
7. Familiarize yourself with the contents of the *Open Toolkit*, including the documentation.

How To Test the Toolkit Installation

1. Run *startcmd.bat* from the toolkit directory.
   A command line window will open with the prompt set to the DITA install directory.
2. Type ant -f build_demo.xml to launch the ANT build of the demo files.
3. Press **Enter** at the prompt to build the samples.
4. Press **Enter** at the next prompt to accept the default output directory.
5. Type htmlhelp at the next prompt to build HTML Help output.
6. Press **y** then **Enter** to start the build.



```
C:\DITA-OT1.3>ant -f build_demo.xml
Buildfile: build_demo.xml

prompt.init:

prompt:
     [echo] Please enter the filename for the DITA map that you
     [echo]                    want to build including the directory path (if a
ny).
     [echo]                    The filename must have the .ditamap extension.
     [echo]                    Note that relative paths that climb (..) are not
 supported yet.
     [echo]                    To build the sample, press return without enteri
ng anything.
    [input] The DITA map filename:

     [echo]
     [echo]                    Please enter the name of the output directory or
 press return
     [echo]                    to accept the default.
    [input] The output directory (out):

     [echo]
     [echo]                    Please enter the type of output to generate.
     [echo]                    Options include: eclipse, htmlhelp, javahelp, pd
f, or web
     [echo]                    Use lowercase letters.
     [echo]
    [input] The output type: (eclipse,htmlhelp,javahelp,pdf,web,docbook)
htmlhelp
```

Figure 2: Building the samples

7. When the build is complete, navigate to the out directory and open the *hierarchy.chm* help file to view the sample output.

8. Repeat this process with other output types if you like, such as pdf, web etc. The pdf type is the default transform using FOP, not the Idiom plug-in, which we will install in the next section.

How To Install the Idiom FO Plug-in

1. If you have not already done so, you will need to download the latest version of the plugin: *http://sourceforge.net/project/showfiles.php?group_id=132728*

   It is located in the **Plugins : Transforms** section.
2. Extract the contents of the zip file into the demo directory in the *DITA Open Toolkit* directory.
3. Download *XEP Personal Edition* from RenderX. *http://www.renderx.com*
4. Install XEP and its licence into demo/fo/lib/xep in your DITA Toolkit directory. You will need to register on the RenderX website to receive your licence file by email. You will need to locate this file during the installation.

   The personal edition is free but places an advertising footer on each page. Eventually you will want to invest in the full version.

   XEP comes with its own installer. Just be sure to point it to the *demo/fo/lib/xep* directory instead of the default.
5. Download *ICU4J*: *http://icu.sourceforge.net*. Click **Download ICU** and locate the *ICU4J* download.
6. Copy the *ICU4j* jar into demo/fo/lib
7. Run *startcmd.bat* and type ant -f integrator.xml.

   Running the *integrator* task uses the DITA plugin architecture to add a new pdf2 target to your *DITA Open Toolkit* installation. Other third party plug-ins work in the same way by adding new targets to your build files. You will learn more about ANT and build files when we start producing some output.

   The pdf2 target will output PDF using XEP and the Idiom stylesheets. The pdf target will output pdf using the default FOP process.

How To Test Idiom FO Output

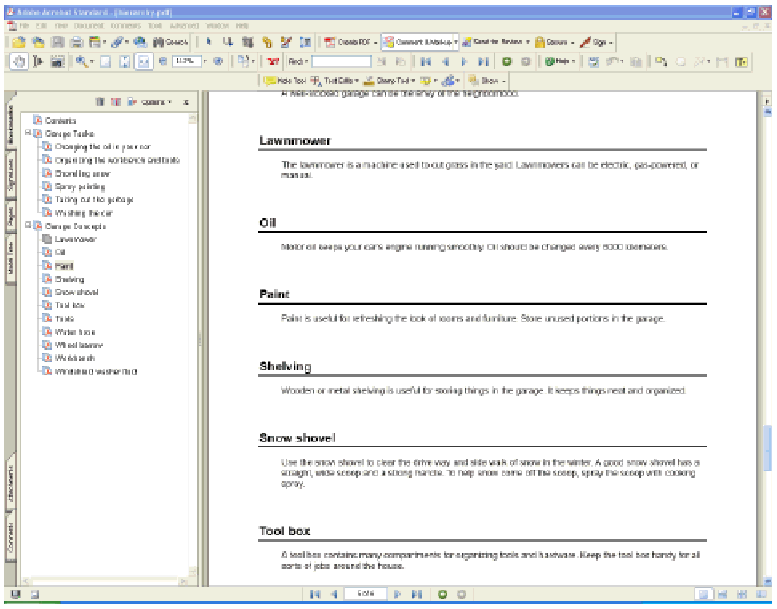
1. Create a new text file.
2. Copy and paste the following ANT script into the new file:

```
  <?xml version="1.0" encoding="UTF-8" ?>
<!-- (c) Copyright IBM Corp. 2004, 2006 All Rights Reserved. -->

<!--
  | basedir can be specified to other places base on your need.
  |
  | Note: input, output, and temp directories will base on the basedir if
  | they are relative paths.
  * -->

<project name="sample_pdf2" default="sample2pdf2" basedir=".">

  <!-- dita.dir should point to the toolkit's root directory -->
  <property name="dita.dir" value="${basedir}${file.separator}.."/>

  <!-- if file is a relative file name, the file name will be resolved
       relative to the importing file -->
  <import file="${dita.dir}${file.separator}integrator.xml"/>

  <target name="sample2pdf2" depends="integrate">
    <ant antfile="${dita.dir}${file.separator}conductor.xml" target="init">
      <!-- please refer to the toolkit's document for supported parameters, and
      specify them base on your needs -->
    <property name="args.input"
      value="${dita.dir}${file.separator}samples${file.separator}hierarchy.ditamap"/>
    <property name="output.dir"
      value="${dita.dir}${file.separator}out${file.separator}samples${file.separator}pdf"/>
    <property name="transtype" value="pdf2"/>
   </ant>
  </target>
</project>
```

We will look at ANT scripts in more detail later. For now notice the `args.input`, `output.dir` and `transtypes` properties. These are the main properties that you must set to produce outputs of different types using different map files.

3. Save the file into the *ant* directory in the *DITA Open Toolkit* directory as *sample_pdf2.xml.* This file can also be found in the tutorial zip file.
4. Run *startcmd.bat*.
5. Switch to the *ant* directory by entering the command `cd ant`.
6. Type `ant -f sample_pdf2.xml`.
   If you encounter any problems during the build, theres a good chance that you didn't licence XEP correctly. Make sure that a valid licence file is in the *demo/fo/xep* directory. The process will report an invalid licence just before the build fails if this is the case.
   The sample PDF will be created in the *out/samples/hierarchy.pdf* directory of the *Open Toolkit*.

   Note: The RenderX footer that appears at the bottom of each PDF page will be removed if you purchase XEP instead of using the free version.



A well-stocked garage can be the envy of the neighborhood.

Lawnmower

The lawnmower is a machine used to cut grass in the yard. Lawnmowers can be electric, gas-powered, or manual.

Oil

Motor oil keeps your car's engine running smoothly. Oil should be changed every 5000 kilometers.

Paint

Paint is useful for refreshing the look of rooms and furniture. Store unused portions in the garage.

Shelving

Wooden or metal shelving is useful for storing things in the garage. It keeps things neat and organized.

Snow shovel

Use the snow shovel to clear the drive way and side walk of snow in the winter. A good snow shovel has a straight, wide scoop and a strong handle. To help snow come off the scoop, spray the scoop with cooking spray.

Tool box

A tool box contains many compartments for organizing tools and hardware. Keep the tool box handy for all sorts of jobs around the house.

Figure 3: The sample output using the Idiom FO plug-in

Planning a Content Model

When planning a content model for your documentation, you need to consider:

- What topic types do you need to use?

  The standard topic types provided for in the *DITA Open Toolkit* are task, concept, reference and topic. If you find that you need additional topic types, then you will need to specialize.

See the documentation in the *docs/articles* directory in the DITA Toolkit for a good introduction to specialization. You should be familiar with DTD before you read it. We are going to create a specialized information type for recipes later in this tutorial, as well as a domain and some extended attributes.

- What topics do you need to include?

  In the *Tutorial Introduction* on page 13 we established that we would need content for general and low cholesterol dinners and that we need documentation sets for novice and experienced cooks. We are going to need different versions of some thanksgiving dishes to satisfy these needs:

  - Turkey
  - Vegetarian Nut Roast
  - Mashed Potatoes
  - Low Fat Mashed Potatoes
  - Green Bean Casserole
  - Cornbread Stuffing

  In addition, lets include some additional cooking instruction topics for novice cooks:

  - How to boil potatoes
  - How to check meat temperature

  We will also include some conceptual topics:

  - Selecting a turkey
  - Laying an attractive dining table

  And finally a reference topic:

  - Shopping List of ingredients and quantities needed

- How will you identify content that can be filtered or flagged and what values will you use?

  The main *selection* attributes available in DITA that can be used to filter and flag content are: platform, product, audience, otherprops, importance, rev and status. Importance and status take one of a list of values defined in the DITA specification. The rest can take any value you choose to give them. Otherprops can be used for any data that doesn't fit into one of the other attributes. You can also create specialized attributes of your own.

  Selection attributes can be assigned to an entire topic, either in the topic itself or in the map that references it. Values in the map will override values in the topic. These attributes can also be assigned to block or phrase elements such as p, ol, ul, xref, ph, keyword etc. Although you will not get an error if you apply one of these attributes to an inline element such as i or b, you can only filter content at the block level.

  Our *Tutorial Introduction* on page 13 give us a clear indication of the major metadata attributes we will need to use and their possible values. We could utilize the default metadata attributes:

  - otherprops: this attribute can be **general** or **low_cholesterol**
  - audience: this attribute can be **novice** or **experienced**.

  However, otherprops is not very descriptive, so the tutorial will walk you through the process of creating an extended metadata attribute called fatcontent that can be used for this purpose.

Creating Content

Now our content has been planned, we can create it. I will be assuming that you are using *XMLmind* for this.

A single topic should be a self-contained chunk. This means that it should be limited to information related to one subject and of the type defined by the topic type. Ideally, you should be able to re-order topics and still have the document make sense (although some orders will usually be more logical than others).

If you don't want to create the tutorial content yourself, you can download a zip file containing the topic files and map files for this project using the link at the end of this topic. See *How To Install the Tutorial Files* for instructions on installing the tutorial files.

Creating Tasks

I create tasks first but this is just my individual preference and you will create content in the best way for you and any other writers on your team. One of the advantages of DITA (for me anyway) is that it easily accomodates non-linear content creation which suits my way of working very well.

The task topic type is designed to contain a list of numbered instructions referred to as steps. Each step has one required element, which is the text of the instruction or command itself and a number of optional elements. These include an additional information element, an example for that step or the results the user can expect after performing the step. All these elements can contain standard body content including text, images, figures, notes etc.

Note: The best way to learn about the valid structure and content of one of the standard DITA topic types is to review the Reference Guide. This can be found in the *docs* directory as *ditaref-book.pdf* or *ditaref-book.chm*. The Reference Guide contains a full list of elements for each of the four main types. The description of each element tells you what its parents can be, what children it can have and which attributes are required and optional.

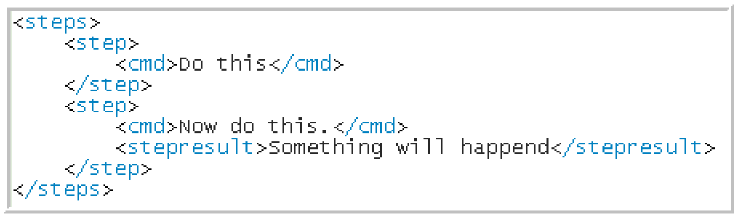
The sequence of elements in a task is very tightly structured. This is helpful to both you as a writer and your users since the content will be complete and consistent.

Always use the DITA tags for the purpose for which they were intended. It is easy to get sloppy as a solo writer, and use an info tag for a section that is actually a step example or step result. You can run into problems later if you do this. For example, if you decide to modify a stylesheet to place the word '**Example:**' at the start of each step example, this will only work if you have tagged all examples consistently. Suitable content for each tag is described in the Reference Guide.

Tasks and Lists

Lists are created in two main ways using DITA:

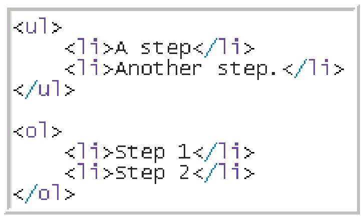
- A set of steps in a task topic is created using its own regular structure containing steps with commands, examples, results and other information in each step. The steps structure in DITA allows for steps where the user must choose from several options and also for choice tables designed to present actions that a user could take and the results of each. Steps are a *specialized* type of list.

  ```
  <steps>
      <step>
          <cmd>Do this</cmd>
      </step>
      <step>
          <cmd>Now do this.</cmd>
          <stepresult>Something will happend</stepresult>
      </step>
  </steps>
  ```

  Figure 4: XML code for a series of steps in a task in DITA

- A list of items in any topic type marked by bullets or by numbers. These use the familiar HTML tags ol for an ordered or numbered list and ul for an unordered or bulleted list (of course, any glyph can be used for a bullet that is supported by the output type). The li element is used for each individual item.



```
<ul>
    <li>A step</li>
    <li>Another step.</li>
</ul>

<ol>
    <li>Step 1</li>
    <li>Step 2</li>
</ol>
```

Figure 5: XML code for unordered and ordered lists in DITA

Inline Formatting

As a general rule, we apply inline formatting to text to indicate that the formatted words belong to a particular category of information. For example, a software company style guide may specify that all window names be in italics, all button names be in bold and all code examples be in Courier New font.

DITA provides you with some common HTML-style inline formatting tags, i for italics, b for bold, u for underline etc as part of the **highlighting** domain. DITA-compliant style sheets such as those in the *Open Toolkit* or in the Idiom FO plugin will deal with these elements by applying the appropriate styles. You cannot, make all formatting changes in this way. For example, you can't change the font size, face, color etc using an inline formatting tag.

Domain Specialization

A better way to handle inline formatting is to define the types of words and phrases that will need to be marked out from the rest of the text and, if necessary, create a specialized domain to handle these. Several specialized domain are provided with the DITA specification including programming and user interface domains. Which means that if you are using DITA to document software products, you may have all you need without further specialization.

For example, the uicontrol element can be used to mark button names, field names, menu options etc. You can then have your style sheet transform these into bold when you build your documents. The obvious advantage is that if at some later date you decide that all user interface control names should be in purple text, you simply need to change your style sheet. You can still use the highlight elements for phrases that have not been defined in another domain.

DITA domain specialization is a complex topic that we will look at in more detail later. For now we will use the highlighting domain.

Short Descriptions and Abstracts

The *DITA Reference Guide* recommends that you use a shortdesc element in each topic. The element is designed to provide summary information for a topic. For HTML output, this information is used as the link summary in the mouse-over text or search engine results listing.

PDF2 output displays the short description at the start of each topic. However, its is relatively easy to override the XSLT template to prevent this behaviour, whilst retaining the link/search engine summary functionality for online formats.

A shortdesc element in the topic itsef can be overidden by the map file.

The abstract element is used to provide a lengthier summary of the topic. It can contain multiple shortdesc elements and other paragraph level content such as tables and images.

How To Create a Task Topic

1. In *XMLMind* , select **File** -> **New** .
2. Select **Task** from the DITA section.
   A DITA task topic template opens, containing the task , title , taskbody , steps and the first step elements .

Note: The templates provided with *XMLMind* associate the topic with the DTD on the docs.oasis-open.org web site. You can edit the templates to reference a local DTD or to use the XML catalog. The templates are located in the *XMLMind* program directory in *addon\dita_dtd_config\template.*

Notice that your current position in the xml structure of the topic is in the text content of the title element , which is a child of the task element. This is shown in the **node path bar** above the editing area.

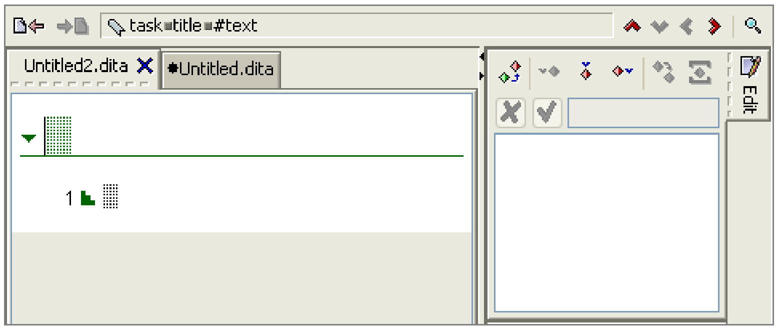
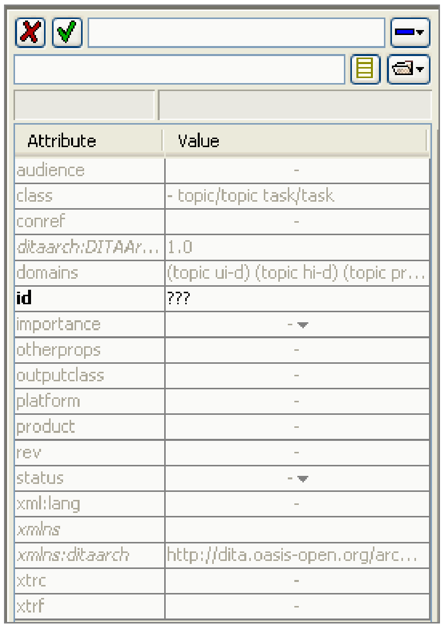


Figure 6: XMLMind task topic template

3. Click on task in the **node path bar** to select the task element.
   A list of possible attributes is displayed in the attributes pane to the right of the editing area.



Attribute	Value
audience	-
class	- topic/topic task/task
conref	-
ditaarch:DITAAr...	1.0
domains	(topic ui-d) (topic hi-d) (topic pr...
id	???
importance	-
otherprops	-
outputclass	-
platform	-
product	-
rev	-
status	-
xml:lang	-
xmlns	
xmlns:ditaarch	http://dita.oasis-open.org/arc...
xtrc	-
xtrf	-

Figure 7: XMLMind Attributes list for the task element.

The id attribute is displayed in bold and its value is set to *???*. This indicates that the id attribute is required. An id attribute is required on the root level element for every topic. *XMLMind* will use ??? as the default value for any required attribute. This will usually throw an error when you build the documents but not cause the build to fail.

4. Type *make_mashed_potatoes* in the id field.
   Give your topics meaningful IDs. This makes managing your content easier later. Try to implement a regular scheme for assigning IDs to your topics, for example, the ID is always the same as the title with an underline(_) character used instead of spaces. Each topic ID should be unique.

Tip: In *XMLMind* , be sure to click in another attribute field before clicking in the editing area after editing an attribute value or your change will be lost.

5. Click back into the title text area. Check the **node path bar** to make sure that you are in the right place.
6. Type 'How to Make Mashed Potatoes' as the topic title.
7. Click next to the 'step' icon . The node path bar indicates that you are in the cmd element of the first step element in the steps container element.

   You can click the step icon to select the entire step including all of its child elements.

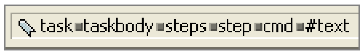
   
   task ▪ taskbody ▪ steps ▪ step ▪ cmd ▪ #text

   Figure 8: Node path bar showing step elements

8. Type 'Boil the potatoes in a large pan of salted water'.
   Immediately, we can see that we are missing something... an ingredients list! Eventually we will create a specialized topic type for recipes but for now we will use a regular un-ordered list. In the top right of the *XMLMind* window you will see the *Edit* pane. This pane contains **Replace** , **Insert Before** , **Insert After** , **Convert** and **Wrap** buttons. Clicking any of these will display a list of valid elements for that action according to the DITA DTD.
9. Using the **node path bar** , select the steps element.
10. Click the **Insert Before** button in the *Edit* pane.

    The **Insert Before** function allows you to insert a *sibling* element before the current one. XML relationships are often described as family relationships. A sibling is an element that is at the same level as the current node. A parent is above and contains the current element. A child is below and contained by the current element .

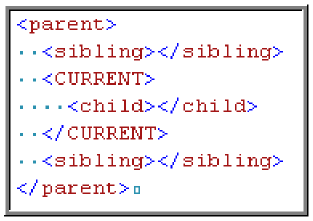
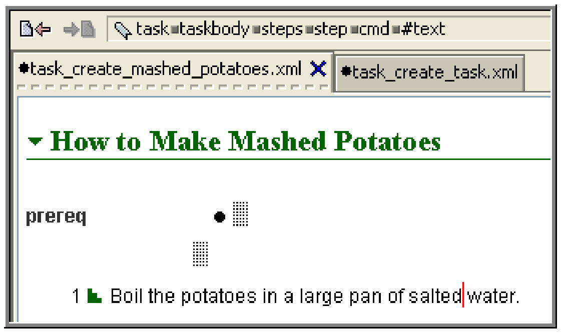
    
    ```
    <parent>
      <sibling></sibling>
      <CURRENT>
        <child></child>
      </CURRENT>
      <sibling></sibling>
    </parent>
    ```

    Figure 9: XML element relationships

    A list of the valid elements that can be inserted before the currently selected one is displayed. There should be two: context and prereq .
11. Select prereq from the list of valid elements since this most closely fits the information that we want to provide.
    We will use an unordered list element (or bulleted list) to provide our list of ingredients. The unordered list element (ul) will be inside the prereq element so we need to use the **Insert** button to insert a child.
12. Click the **Insert** button and select ul from the list of valid elements.
    XMLMind will automatically insert an li element as the child of your ul element. Just as in HTML, ul represents the entire list, each li element represents an item in the list. You will see this as a bulleted item in the editing area .

    
    task ▪ taskbody ▪ steps ▪ step ▪ cmd ▪ #text

    task_create_mashed_potatoes.xml | task_create_task.xml

    How to Make Mashed Potatoes

    prereq ●

    1 Boil the potatoes in a large pan of salted water.

    Figure 10: Inserting an un-ordered list

13. Type '2lb potatoes' as the first list item.
14. Click the **Insert After** button to insert a sibling element after the current element. The only valid sibling for an li element (list item) is another li element.
15. Continue adding list items for the following ingredients:

    - 2 tbsp butter
    - 2 tbsp milk
    - 1 tsp salt

    (I make no claim that this is a good recipe!)
16. If you have an extra text element after your list, click it to select it then right click on the #text element in the **node path bar** and select **Delete** .
17. Select your first step and add another step after it.

    In *XMLMind*, you can add the same element type as a sibling by CTRL + clicking on the element in the node path bar. Shift + clicking will insert the same element before.

    When you add a new step, *XMLMind* will automatically insert a cmd element for your instruction text.
18. Add 'Mash the potatoes.' as the next instruction.
    At this point, lets add a handy hint telling would be Thanksgiving cooks that a potato masher is the preferred device for mashing potatoes!.
19. Insert an info element after the cmd element.
20. Add a note element as a child of the info element.

    All sections of a task step following the cmd element, such as info , stepresult , stepxmp etc can contain the full range of block and inline elements, such as notes, figures, images, tables, bold, italic etc. See the DITA Reference Guide in the toolkit docs directory for information about these elements. The cmd element cannot contain block level elements.

    The note element is used for tips, warnings, cautions and other similar items in addition to the standard note. To see the different types of note element, click the type attribute field in XMLMind for a drop-down list or consult the DITA Reference Guide.
21. Enter some suitable text in the note element.
22. Add further steps indicating that the cook should add the butter, milk and salt to the potatoes and beat them well. In the final step, add a stepresult element, indicating to the cook how their finished potatoes should look, what consistency they should be etc etc.
    You can consult the sample topic files by downloading them from *here* .
23. Create a directory *C:\DITAContent\tutorial* (modify the drive letter as appropriate) and save the file as *task_make_mashed_potatoes.xml*.

How to Apply Inline Formatting

In the creating mashed potatoes topic, lets italicize all food names.

1. In the ingredients list of the mashed potatoes topic, select the word 'potatoes'.
2. Click the **Wrap** button in the **Edit** pane.
   The **Wrap** function allows you to insert an element as the parent of the currently selected text.
3. Select the i element from the list of valid elements.

Tip: The DITA plugin for *XMLmind* also provides a toolbar with buttons to add inline formatting.

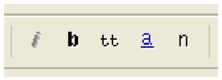
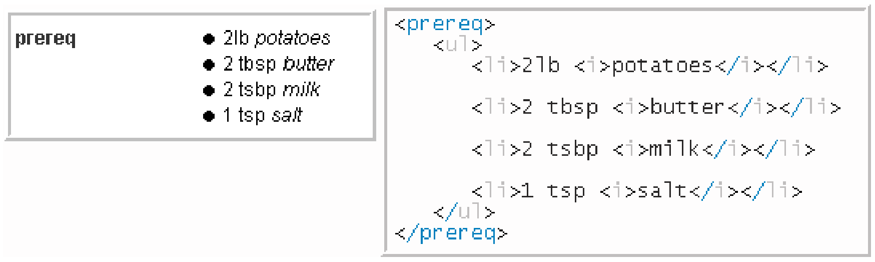


Figure 11: XMLmind DITA Highlighting Toolbar

To continue adding unhighlighted text immediately after the highlight element, press **Insert** to insert a text element.

XMLMind will italicize the text in the editing area. But always bear in mind that the formatting applied by *XMLMind* as you create content does not reflect the final formatting of your documents when you build them.



prereq

- 2lb *potatoes*
- 2 tbsp *butter*
- 2 tsbp *milk*
- 1 tsp *salt*

```
<prereq>
   <ul>
      <li>2lb <i>potatoes</i></li>

      <li>2 tbsp <i>butter</i></li>

      <li>2 tsbp <i>milk</i></li>

      <li>1 tsp <i>salt</i></li>
   </ul>
</prereq>
```

Figure 12: Inline formatting in XMLMind and the corresponding XML source code

4. If you wanted to change the italics to bold, you can do so by selecting the i tag using the **node path bar**, then click the **Convert** button and select the b element.

Tip: To remove highlighting from a phrase, select it in the **node path bar** and **Convert** it to (text). In the *XMLMind* element lists, (text) refers to text content of a tag and can be used to add text before or after a phrase element. You can also use the **Remove Highlighting**(n) button from the **DITA Highlighting** tool bar but you must select the highlight element itself from the **node path selection** bar first, not simply select the text..

Convert will replace the selected element, but preserve its content. **Replace** will replace the currently selected element with an empty element of the new type resulting in the loss of any content in the old element..

Creating Concepts

Concepts are topics that provide the background information that users need to understand and follow the task topics.

In the tasks that we created for this tutorial, we used an element context, which can be used before the steps element. You can use this in place of a full concept topic if you only need to provide a line or two of background information for a task. If you have more than that, use a concept topic.

The concept topic type is designed to contain relatively loosely structured content. The title element tags the topic title, as in all the other standard DITA topic types. The conbody element is used to contain the topic content. This can be followed by a related-links element.

Note: As with tasks, use the Reference Guide to familiarize yourself with the elements that can be used in a DITA concept topic. This can be found in the *docs* directory as *ditaref-book.pdf* or *ditaref-book.chm*.

The conbody element can contain p (paragraph), fig, image, table, xref and a number of other block and inline elements. Conceptual information can be presented in many forms and this is reflected in the relatively open structure of a concept topic.

Images

There are two main ways to place an image in your topic:

- Use the image element to add an image by itself. You can use this in the middle of a block element, as I do in this tutorial to display an image of a button that I refer to in the text.
- Use the fig element to insert an image with a caption and/or description.

Tip: Whilst fig is most often used with images, it can also be used with text content that you want to assign a title and/or description to and treat as a single entity.

For other types of media, the DITA object element corresponds to the HTML <object> tag and can be used to insert flash, video, applets etc.

You must set the href and placement attributes on the image element correctly. The href attribute is the path to the image file. The placement attribute can be set to *inline* or *break*. A fig element containing an image with a caption will almost always be set to placement = break. This will cause the flow of the text to stop, insert the image then continue on the next line. Setting placement to *inline* in HTML output, for example, will put the image and caption on the same line.

Note: When creating task topics, you cannot insert a fig element into a cmd element. You must add an info element or a stepresult element after the cmd to contain the figure. You can insert an inline image element into a cmd element though.

Links and Cross References

Links can be placed anywhere in a topic using the xref element or you can place a list of related links after the main content of the topic using the related-links element. It is easier to maintain links between topics using the related links section and a relationship table (see below) than to scatter xrefs throughout your content.

Related links can be grouped using two different elements:

- linklist: this element will preserve the order of the links as you write them.
- linkpool: this element allows the list to be ordered during the document build process. The DITA Open Toolkit produces links grouped by topic type i.e **Related Concepts**, **Related Reference** etc.

When you link to another DITA topic, the build process will automatically resolve the title of the destination topic and use that as the text of the link. In a non-DITA link the address will be used as the link or you can add a linktext element as a child of the link element to specify the text to use. Use the format and scope attributes of the link element to specify the content type of the destination file and whether it is an internal or external link. This allows the processor to determine whether the link should be treated as an internal cross reference or as a link to an external web site or other resource.

The build processes will also output "breadcrumb" style links. These links provide navigation between the siblings and parents/children of a topic. They are created by default for HTML/HTML Help output types and can also be enabled for PDF if desired although by default they are off since this type of linking is less useful in a linear print document. You can set the collection-type of groups of topics in your map file to determine how your "breadcrumbs" will be formatted. For example, you can describe a set of topics in a map as a sequential collection-type, which will result in numbered child links. I use this feature in this tutorial to output the numbered links at the end of the topics. We will look at this in more detail when we create a map file.

Related links aredisabled by default in pdf2 output. You can enable them by setting the disableRelatedLinks property to false in the build file.

Note: linklist, linkpool and other similar elements used to contain a list group of similar objects can take the collection-type attribute but I havent been able to determine how this affects the output using the *Open Toolkit* (if it does).

Relationship tables are used to define how topics are related to each other in the DITA map. This information is used to control the related links output. It allows the relationships to be maintained ina single place. It also means that relationships can be different in different maps. Collection types are another important aspect of topic relationships. They determine whether a group of related topics have a sequential relationship (such as in a tutorial where topics should be read in order) or a family relationship, in which topics relate to the current topic and to each other.

How To Create a Concept Topic

1. In *XMLMind*, select **File** -> **New**.
2. Select **Concept** from the DITA section.
   A DITA task topic template opens, containing the concept, title and conbody elements.

   Note: The templates provided with *XMLMind* associate the topic with the DTD on the docs.oasis-open.org web site. You can edit the templates to reference a local DTD or XML catalog reference to a DTD.

   Notice that your current position in the xml structure of the topic is in the text content of the title tag, which is a child of the concept tag. This is shown in the **node path bar** above the editing area.

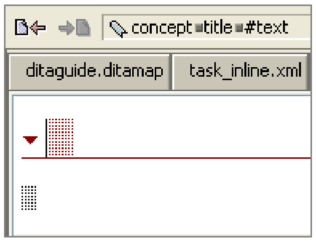
   

   Figure 13: XMLMind concept topic template

3. Set the id attribute on the concept element to *festive_tables*.
4. Enter the title *Festive Table Decoration.*
5. Click in the text area below the title element. This will place you in the first p element in the conbody element.
   As in HTML, p is used to denote a paragraph. You can use the same inline formatting elements in a paragraph in a concept topic as you can in block elements in a task topic.
6. Enter the following text:

   It doesn't have to cost much to add some whimsy and style to the Thanksgiving table. Buy a large sheet of inexpensive fabric in an earth-tone color to cover the table. Don't waste money on fine linens. Add height and drama to the buffet by placing shoeboxes, coffee cans and other items under the fabric, then place the dishes on top of the raised areas.

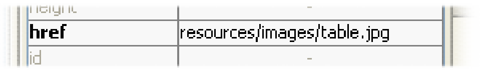
7. Now we will add a picture of a festively decorated table to the topic by inserting a fig element as a child of the p element. Make sure your cursor is positioned inside the p tag, at the end of the text, before inserting the fig element.
   XMLMInd allows you to insert elements using the keyboard:

CTRL+I	Insert
CTRL+H	Insert Before
CTRL+J	Insert After
CTRL+R	Replace
CTRL+T	Convert
CTRL+SHIFT+T	Convert [Wrap]

   After pressing the desired keyboard shortcut, type the first few letters of the element to select it in the list and hit Enter to insert it.
8. Press **CTRL+J** and type *fig* to select the fig element. Press **Enter** to insert it.

A title element for a caption and an image placeholder will be inserted for you.

9. Type a festively decorated table into the title element.
10. Select the image placeholder and select the href attribute in the **Attributes List**.
11. Click the **Browse** button and choose an image file. (An appropriate image is included in the tutorial zip file).



height	-
href	resources/images/table.jpg
id	-

Figure 14: Setting the href attribute

12. Select *break* from the placement attribute list.

    Although the fig title is displayed above the image in the *XMLMind* template (the title element must come before the image element in the fig element), the caption will be correctly displayed below the image with a figure number in HTML and PDF output.
13. Finally, we will direct the reader to some useful web sites on the topic of table decorating. Use the **node path bar** to select the conbody element.
14. Insert a related-links element after the conbody element.

    XMLMind will insert the first link element for you

Figure 15: A newly inserted related-links element

15. We are going to group all the external web links into a linkpool element. The advantage of this is that you can set the linkpool format attribute to html and scope to external and it will apply to all child link elements. Select the link that *XMLMInd* inserted for you and wrap it with a linkpool element.
16. Set the format attribute of the linkpool element to *html*.
    This makes sure that the build process will interpret this as a non-dita link rather than as an internal cross reference.
17. Set the scope attribute to *external*.
18. Select the first link element by clicking on the chain link icon .
19. Select the href attribute in the **Attributes List**.
20. Enter http://www.hgtv.com/hgtv/ah_entertaining_decor/ as the URL.
21. Add a couple more links to the list.

    Note: You can use the linktext element to provide text for the link, otherwise the build process will use the URL as the text of an external link. The linktext element is a child of the link element.

22. Save the file as *content_festive_table.xml in the DITAContent\tutorial directory*.

Creating Reference Topics

Reference topics provide lists, charts or tables of related information. These are the topics that get down into the nitty gritty of a subject area and provide complete and detailed information. Reference topics can be used in programming API references, user interface references etc.

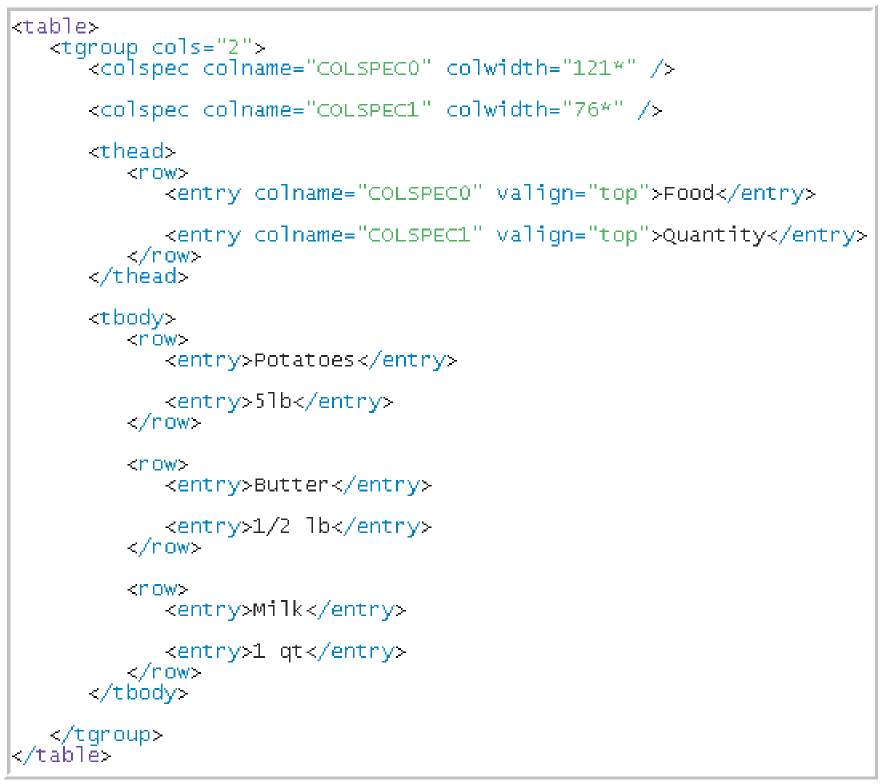
The reference topic type is designed to contain multiple sections, each with its own heading. The title element tags the topic title, as in all the other standard DITA topic types. The refbody element is used to contain the topic content. The refbody element can contain multiple section elements, each with its own title element to contain the section heading. Reference topics commonly contain tables and simple tables to display lists of items and their definitions or descriptions. The refbody element can be followed by a related-links element.

Note: As with tasks, use the Reference Guide to familiarize yourself with the elements that can be used in a DITA concept topic. This can be found in the *docs* directory as *ditaref-book.pdf* or *ditaref-book.chm*.

Creating Content: Tables

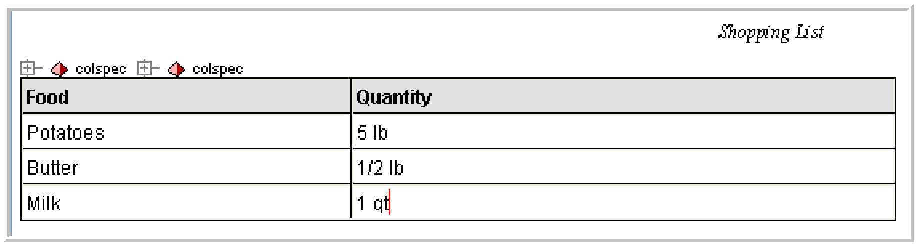
DITA has two main table elements:

- table: this gives access to the full table capabilities of DITA. A table contains one ore more tgroup elements which in turn contain the column layout, table head and table body. The column layout and table head are optional. The table body contains the actual rows and cells of the table. A table element can also contain a caption using the title or desc child element.
- simpletable: a simpletable element is used for a table which has a regular structure and no caption. These kinds of table are good for multi-column tabular data such as lists of properties and their definitions or a phone list. A simpletable can contain a single table head and multiple rows.



```
<table>
   <tgroup cols="2">
      <colspec colname="COLSPEC0" colwidth="121*" />

      <colspec colname="COLSPEC1" colwidth="76*" />

      <thead>
         <row>
            <entry colname="COLSPEC0" valign="top">Food</entry>

            <entry colname="COLSPEC1" valign="top">Quantity</entry>
         </row>
      </thead>

      <tbody>
         <row>
            <entry>Potatoes</entry>

            <entry>5lb</entry>
         </row>

         <row>
            <entry>Butter</entry>

            <entry>1/2 lb</entry>
         </row>

         <row>
            <entry>Milk</entry>

            <entry>1 qt</entry>
         </row>
      </tbody>

   </tgroup>
</table>
```

Figure 16: XML code for a table in DITA



Food	Quantity
Potatoes	5 lb
Butter	1/2 lb
Milk	1 qt

Figure 17: DITA Table in XMLMind

The colspec element is the first child of a tgroup element and is used to define the widths, names, ruling etc of the columns in the group. You can redefine the column layout for each different tgroup in the table.

In a simpletable, the relative widths of its columns are defined using the relcolwidth attribute on the simpletable element itself. The widths are expressed as rations. e.g. 1* 4* indicates that the second column is four times as wide as the first.

The keycol attribute in a simpletable is used to define a 'key column'. In practice, this highlights the specified column in the table as a vertical header so it should normally be the first column. The column is specified as a numerical index, 0 for the first column, 1 for the second etc.

How To Create a Reference Topic

This reference topic consists of a table of ingredients and quantities required for our Thanksgiving dinner.

1. Create a reference topic and give it a title of *Shopping List*.

   How To Create a Task Topic on page 19 and *How To Create a Concept Topic* on page 25 for information on creating topics.
2. *XMLMind* inserts a refsyn element by default. This element is used to define the syntax or signature of a reference topic. For example, it might be used to give a command line utility's calling syntax. It is not especially useful for a shopping list so go ahead and delete the element.
3. The next element is a section element and its title element. A reference topic can be structured with multiple sections, each with its own title. Type *Dairy* into the **title** box for the first section.
4. Insert a table(head) element as the child of the first p element.

   XMLmind will insert a skeleton table when you insert a table element. If you choose **table(head)**, the skeleton table will have a header row.

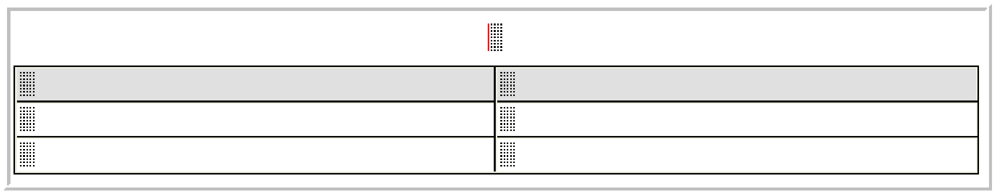
   

   Figure 18: A skeleton table in XMLMind
5. Because this is a full table, we have the option to add a caption. We aren't going to though, so use the **node path bar** to select the title element of the table and press **Delete** to remove it.
6. Click somewhere in the second column.
7. In *XMLMind*, from the **DITA** menu, select **Column** then **Insert After**.

   This has the effective result of incrementing the cols attribute on the tgroup element. If you increase the attribute manually, *XMLmind* doesnt draw an additional column until you manually populate it with entry elements.
8. Enter Item, Brand and Quantity as the column headings.
   The header row is contained by the thead element. This contains one or more row element and an entry element for each cell.
9. Add an item the list: *Milk, Mayfield, 1 quart*

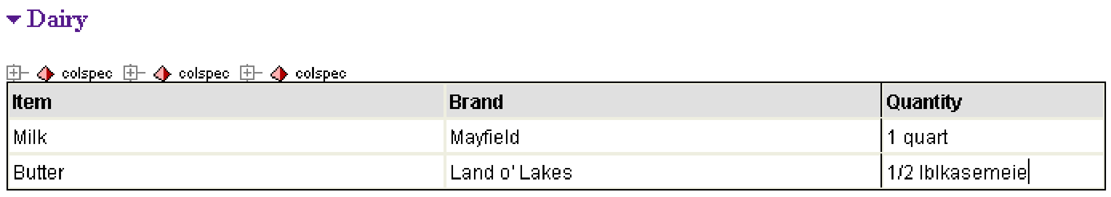
   The main rows of the table are contained by the tbody element.
10. Continue adding items. Select **Rows** -> **Insert After** from the **DITA** menu to insert additional rows.

    As with columns, you can also add the row elements manually but you will also need to add each cell individually.
11. Because this is a full table, we also have a little more control over the formatting and layout of the table. You can use the colspec element to set the width of individual columns, define the table ruling and assign the column a name. There can be a colspec element for each column in the table. Select the tgroup element in the **node path bar** and insert a colspec element before it.
12. Set the colwidth attribute to 200. Set the colsep value to 0 to remove the ruling from the right edge of the column. Set rowsep to 0 to remove horizontal ruling for the column.

13. Insert two more colspec elements as siblings of the first. Set their attributes as follows:

    - width 200, colsep 1, rowsep 0
    - width 100, colsep 1, rowsep 0

    You should end up with a table that looks like the one below in *XMLMind*.



Dairy

colspec colspec colspec

Item	Brand	Quantity
Milk	Mayfield	1 quart
Butter	Land o' Lakes	1/2 lblkasemeie

Figure 19: The first section of the shopping list topic

14. Add a second section element after the first and give it a title of *Produce*.
    In this section we will use a simpletable. In fact, we could have used a simpletable for the first section too but I wanted to demonstrate some of the differences.
15. Add a simpletable(head) element to the section.
16. Use the **DITA** menu in *XMLMind* to add a column in the same way as you did for a regular table.
    simpletable uses the strow, sthead and stentry elements for rows, headers and cells.
    simpletable does not accept the colspec element. You set the relative column widths using the relcolwidth attribute.
17. Set the relcolwidth attribute to 2* 2* 1*.
    This will set the third column to half the width of the first two.
18. Add a few items to the *Produce* list to complete the topic.
19. Save the file as *ref_shopping_list.xml* in the *DITAContent\tutorial* directory .

Complete the Content

1. Add the rest of the topic that we planned in the content planning stage.
   Planning a Content Model on page 16
   You can also download the topic and map files here.
   You should have the following topics:

   - concepts
     - Festive Table Decoration
     - Selecting a Turkey
   - tasks
     - How To Make Mashed Potatoes
     - How To Make Low Fat Mashed potatoes
     - How To Make Turkey
     - How To Make Vegetarian Nut Roast
     - How To Make Green Bean Casserole
     - How To Make Cornbread Stuffing
     - How To Boil Potatoes
     - How To Check Meat Temperature
   - reference
     - Shopping List

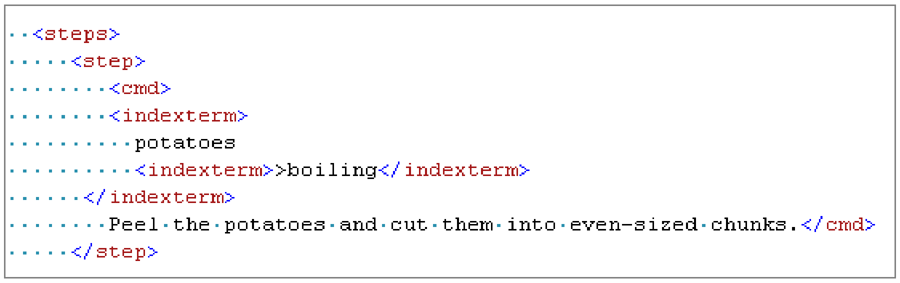
Keywords and Indexes

Generating Indexes

The basic element used to specify index entries is indexterm. This element can be used either in a topic itself or in a reference to a topic in your map file.

In topics, index terms can be grouped together in the prolog or they can be inserted into your content.

Note: You cannot *wrap* existing text in an indexterm element. Each indexterm element must have its own content that will not appear in the topic output but will be added to the index.



```
··<steps>
·····<step>
········<cmd>
········<indexterm>
··········potatoes
··········<indexterm>>boiling</indexterm>
······</indexterm>
········Peel·the·potatoes·and·cut·them·into·even-sized·chunks.</cmd>
·····</step>
```

Figure 20: XML code for index terms

The figure above shows the code for inserting index terms into your text.

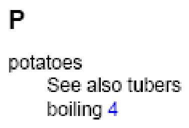
If you want to place index terms in your prolog, you will need to insert a prolog element after the topic title. Insert a metadata element as a child of the prolog, followed by a keywords element as a child of metadata and finally a list of indexterm elements.

Nested Index Entries

You can nest indexterm elements to create nested or multi-level index entries. In the code in the figure above, *boiling* would become a sub-entry of *potatoes*. You can also create see and see also index entries by inserting an index-see or index-see-also element as a child of the indexterm element.

```
<indexterm>potatoes
  <index-see-also>potatoes</index-see-also>
  <indexterm>tubers</indexterm>
</indexterm>
```

Would produce the index entry *tubers, see also potatoes* with no page number.



P

potatoes
    See also tubers
    boiling 4

Figure 21: Index output in PDF

Keywords

You can mark any word or phrase in your text content as a keyword by wrapping it in the keyword element. Keywords are place into the web page metadata when the output is HTML or HTML Help but other than that they do not have any special processing applied to them by the *Open Toolkit*. In fact, the most useful aspect of the keyword element is that you can specialize it. So you could create a ingredient element as a specialized keyword for our Thanksgiving documents and use it to format references to recipe ingredients. The programming domain supplied with the *Open Toolkit* provides specialized keyword such as option and apiname.

Add Index Entries

Now we will go back and add some index entries to our Thanksgiving documentation topics. We will add out first index entry into the topic content and the second into the prolog.

1. Open the How To Make Mashed Potatoes topic in *XMLMind*.
2. Click in the text of the first step.
3. Insert an indexterm element.
4. Type 'making mashed potatoes' as the content of the indexterm element.
   Next we will add an index entry to the prolog.
5. Select the title element using the **node selection bar**.
6. Insert a prolog element after the title.
   The prolog is displayed as a yellow box in *XMLMInd*. *XMLMind* will insert a data element automatically.
7. Select the data element using the **node selection bar**.
8. Replace it with a metadata element by:
   - Clicking the **Replac**e button and selecting **metadata**.
   - Pressing CTRL+R and selecting **metadata**.

   Note: *XMLMind* will inset a data element as the first child of the metadata element.
9. Select the data element and replace it with a keywords element.

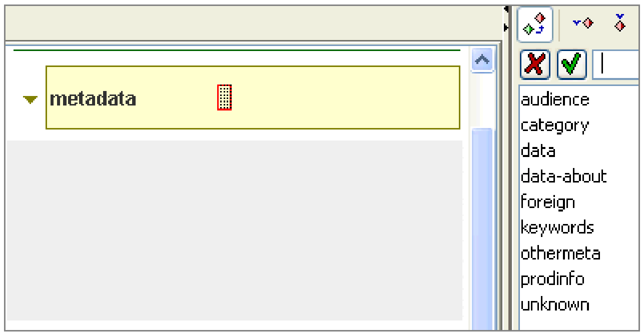


Figure 22: Replacing the data element with a keywords element

   XMLMind will inset an option element as the first child of the keywords element.
10. Select the element and replace it with an indexterm element.
11. Enter 'mashed potatatoes:making' as the text content of the indexterm element.
12. Add another indexterm element as a child of the first.
13. Enter 'making' as the text content.
    In your final output this will lead to the index entry 'mashed potatoes' with a sub entry 'making'.

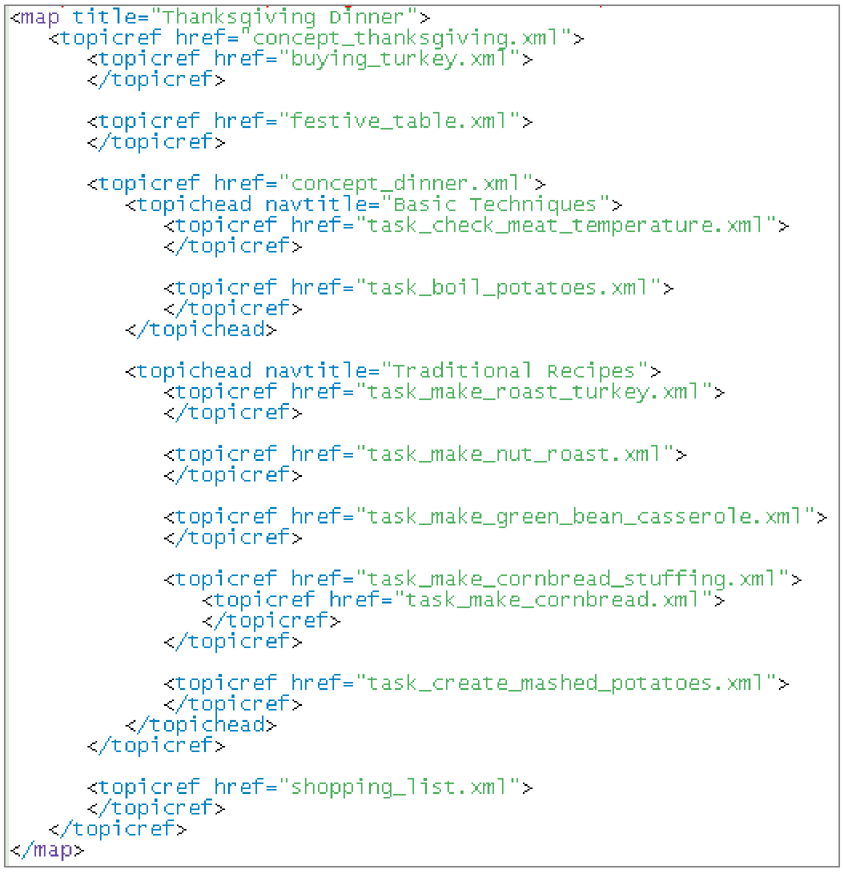
DITA Map Files

A DITA map file is used to pull together all your topics into a document. The map file determines the sequence and hierarchy of the topics and in some cases, the relationship between them. You can create as many map files as you need. For example, you can create one map file for PDF output it and a different one for HTML output.

You can also filter topics out of the output at the map file level. For example, you could apply the attribute '*audience='administrator*' to a topic reference in the mapfile, then produce a document without administration topics by filtering those out using a ditaval file. This would exclude that topic and all its children. Note that if the the topic content itself has the audience attribute set to something else, the setting in the map file will override it. All topic reference and topic grouping elements accept the full list of selection attributes; audience, platform, product, importance, status, rev and otherprops.

DITA maps use a DTD of their own to validate their content. This is found in map.dtd, map.mod, mapgroup.ent and mapgroup.mod in the *DITA Open Toolkit dtd* directory. All valid elements for map files are descibed in the 'Map Elements' chapter of the *DITA Reference Guide*.

The content of a simple map file consists of a list or tree of references to DITA topics:



```
<map title="Thanksgiving Dinner">
   <topicref href="concept_thanksgiving.xml">
      <topicref href="buying_turkey.xml">
      </topicref>

      <topicref href="festive_table.xml">
      </topicref>

      <topicref href="concept_dinner.xml">
         <topichead navtitle="Basic Techniques">
            <topicref href="task_check_meat_temperature.xml">
            </topicref>

            <topicref href="task_boil_potatoes.xml">
            </topicref>
         </topichead>

         <topichead navtitle="Traditional Recipes">
            <topicref href="task_make_roast_turkey.xml">
            </topicref>

            <topicref href="task_make_nut_roast.xml">
            </topicref>

            <topicref href="task_make_green_bean_casserole.xml">
            </topicref>

            <topicref href="task_make_cornbread_stuffing.xml">
               <topicref href="task_make_cornbread.xml">
               </topicref>
            </topicref>

            <topicref href="task_create_mashed_potatoes.xml">
            </topicref>
         </topichead>
      </topicref>

      <topicref href="shopping_list.xml">
      </topicref>
   </topicref>
</map>
```

Figure 23: XML Code for a DITA Map File

The root element of a map file is map. The most important attribute of this element is title. Although it is an optional attribute, you should supply it as it is often used as the document title in output. It appears on the front matter of PDF output, for example.

Each topicref element provides a reference to a single topic. The location of the topic is provided by the href attribute. In the example code above, each XML file contains a single topic, so you need only provide the file name. If you have multiple topics in your XML files, you will need to provide the reference in the form of *filename.xml#topic_id*. The topicref elements can be nested to reflect the topic hierarchy.

The topichead element provides you with a way to group multiple topics under a single heading without creating a parent topic. Different outputs treat this element differently; PDF effectively ignores it whilst HTML Help uses it in the Contents panel. Compare the two content page outputs below and note that the *Basic Techniques* and *Traditional Recipes* headings are missing from the PDF output.

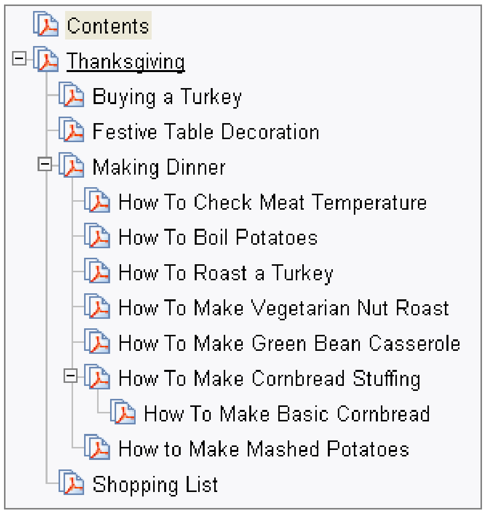

Contents
Thanksgiving
Buying a Turkey
Festive Table Decoration
Making Dinner
How To Check Meat Temperature
How To Boil Potatoes
How To Roast a Turkey
How To Make Vegetarian Nut Roast
How To Make Green Bean Casserole
How To Make Cornbread Stuffing
How To Make Basic Cornbread
How to Make Mashed Potatoes
Shopping List

Figure 24: Contents Page Generated for PDF Output

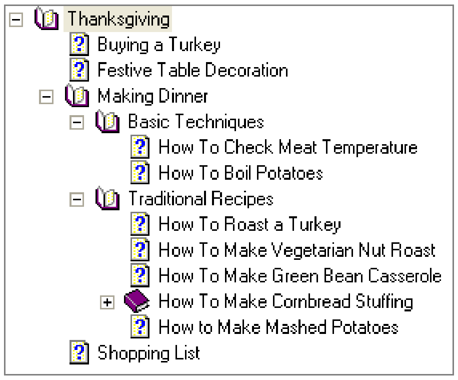

Thanksgiving
Buying a Turkey
Festive Table Decoration
Making Dinner
Basic Techniques
How To Check Meat Temperature
How To Boil Potatoes
Traditional Recipes
How To Roast a Turkey
How To Make Vegetarian Nut Roast
How To Make Green Bean Casserole
How To Make Cornbread Stuffing
How to Make Mashed Potatoes
Shopping List

Figure 25: Contents Page Generated for HTML Help Output

The topicgroup element is also used to group topic references together but this is 'behind the scenes' grouping only. It allows you to apply the same attribute settings to a group of topics, such as collection-type.

Collection Types

The collection-type attribute allows you to define the nature of the relationship between topic references. It can take one of the following values:

- family
- unordered
- sequence
- choice

In terms of the effect that you will see in the output, this is mostly apparent in automatically generated related links sections. This means that this attribute is most applicable to online formats such as HTML Help. In PDF output, related links are disabled by default since they are of limited use in printed output. If you do want to display them, you can add the property disableRelatedLinks to the build file and set its value to 'No' . However, the default stylesheets for PDF output do not take into account the collection-type attribute.

In HTML output, the collection-type attribute is useful for controlling the output of related links at the bottom of a topic. If you set the collection-type of a group of links to sequence, sibling links will be numbered in the related links section.

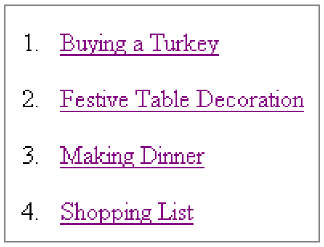
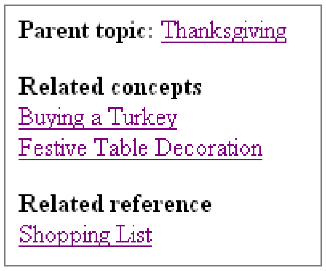

1. Buying a Turkey
2. Festive Table Decoration
3. Making Dinner
4. Shopping List

Figure 26: Related Links Generated for a Sequence Collection Type

If you set the collection-type attribute to family, related links will be grouped by topic type. A family represents a group of links that relate to each other as well as to the current topic.


Parent topic: Thanksgiving

Related concepts
Buying a Turkey
Festive Table Decoration

Related reference
Shopping List

Figure 27: Related Links Generated for a Family Collection Type

choice and unordered collection types generate parent and child links only although you could customize the *Open Toolkit* stylesheets to treat them differently.

Titles in Content Pages and Navigation

The use of and differences between various type of titles in DITA can be confusing. By default, the title that you give to a topic using the topic level title element will be used as the text for all links, cross references, content pages and navigation that reference that topic in addition to the topic heading itself.

Navigation Titles

A navigation title is the text used to reference or link to a topic in a contents page. You can set this text using the navtitle attribute of the topicref element in a map file. Even if you do that though, the default behaviour is to ignore navtitle on a topicref element. You must also set the locktitle attribute on the topicref element to *yes*. I cannot think of a situation where you would want the reference in the contents to be different to the heading of the topic. But if you need to do it, thats how it can be accomplished.

Tip: If you want to modify the text used for a link somewhere other than in a contents page, use the linktext element as a child of the link element. This is most useful when linking to an external web page or other resource. In this instance, make sure that you set the format attribute to html, pdf etc so that the build process does not try to resolve it as an internal DITA link.

The navtitle attribute on the topichead element is used to set the heading for the group of topics and should always be supplied.

navtitle can also be used in topic files themselves. It can be a child of the titlealts element immediately follows the title element and is used to supply navtitle and searchtitle values for a topic. It is optional.

Search Titles

The searchtitle attribute (used by topicmeta and titlealt elements) is used only by HTML outputs. It is used to create the html title tag which is often used by search engines to provide a summary of the page content.

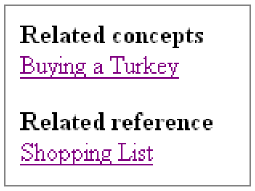
Relationship Tables

Relationship tables are used to define the relationships between topics. These can be a little difficult to understand and it can be a challenge to envisage them in use, but they can help manage the connections between different topics in complex documentation sets and make finding the infomation they need much easier for your users.

Table 1: A relationship table

type='task'	type='concept'	type='reference'
task_make_roast_turkey.xml task_make_nut_roast.xml task_check_meat_temperature.xml	concept_buying_turkey.xml	ref_shopping_list.xml
task_boil_potatoes task_makle_mashed_potatoes		ref_shopping_list.xml
task_make_cornbread_stuffing.xml task_make_cornbread.xml		ref_shopping_list.xml
task_make_green_bean_casserole.xml		ref_shopping_list.xml

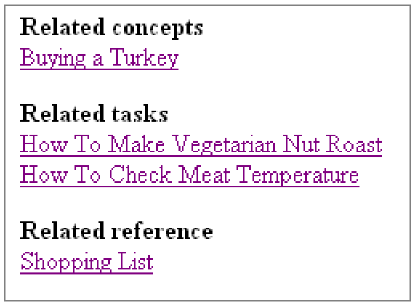
In the relationship table, topics that are on the same row are related to each other. By default, topics that are in the same cell are not. Therefore, a topic in the first cell in the table would generate the following related links section in HTML:



Related concepts
Buying a Turkey

Related reference
Shopping List

You can make links in the same cell relate to each other by setting the cell's collection-type attribute to *family*. This would result in these related links for the *make roast turkey* topic:



Related concepts
Buying a Turkey

Related tasks
How To Make Vegetarian Nut Roast
How To Check Meat Temperature

Related reference
Shopping List

Relationship tables are provided in a map file in addition to the the sequence/hierarchy.

Types in the relationship table must map to topic types in your DITA DTD. If you need additional types, you will need a specialized topic type.

The tables are built in a similar way to standard tables using reltable, relrow, relcell, relheader and relcolspec elements. They lack any output or formatting attributes, such as colwidth or colsep but some will take the collection-type attribute as described above. See the *DITA Reference Guide* for full details on each one.

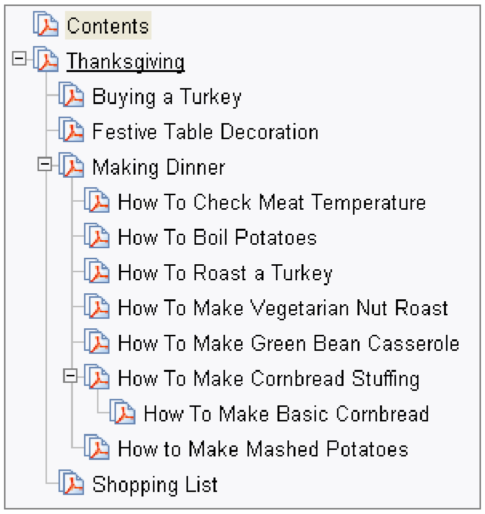
How to Create a Map File

1. In *XMLMind*, select **New** from the **File** Menu.
2. Select **Map** from the **DITA Map** section.
3. Click **OK**.
4. Select the map element and set the title attribute to 'Making Thanksgiving Dinner'. *XMLMind* inserts the first topicref element for you.
5. Save the file into your *DITAContent* directory as *thanksgiving.ditamap*.

   Note: Ideally I would like to create a maps directory for all ditamap files. However, if you create a map file that requires you to travel up the file directory and back down (i.e ../../mytopic.xml), the build creates a lot of extra directories when it creates output. For best results, topic files should always be in child directories of the map directory.

   This will make sure that the relative paths in your hrefs are set correctly.
6. Select the topicref element.
7. Set the href attribute to point to your introductory concept topic on Thanksgiving.

   XMLMind allows you to drag and drop XML files from WIndows Explorer into a topicref. Drop the file directly on to the element or drop it on the arrow symbol at the end of the element to insert it as the last child.
8. Insert another topicref as a child of the first.
9. Set the href to point to your concept topic on buying turkeys.
10. Continue adding topicref elements to the map until you have a structure similar to that shown below:



Contents
- Thanksgiving
  - Buying a Turkey
  - Festive Table Decoration
  - Making Dinner
    - How To Check Meat Temperature
    - How To Boil Potatoes
    - How To Roast a Turkey
    - How To Make Vegetarian Nut Roast
    - How To Make Green Bean Casserole
    - How To Make Cornbread Stuffing
      - How To Make Basic Cornbread
    - How to Make Mashed Potatoes
  - Shopping List

Figure 28: The structure of the Thanksgiving map file.

Next we will create some topic headings using the topichead element.

11. Select the *How To Check Meat Temperature* topicref element.
12. From the *XMLMind* **Select** menu, choose the **Extend Selection to Following Sibling** option. The *How To Check Meat Temperature* topic and the *How To Boil Potatoes* topic should now be selected.
13. Wrap the selected elements in a topichead element.
14. Set the navtitle attribute of the topichead element to 'Basic Recipes'.
15. Using the same procedure, create a heading group for the remaining recipes called 'Traditional Recipes'.
16. Save the map.

How To Create a Relationship Table

1. Select the first topicref element in your map file.
2. Insert a reltable element before it.
   XMLMind will insert the first row and cell for you.
3. Click the **Select Next** button or press CTRL + Down Arrow to select the next child element.
   The first relrow element will be selected.
4. Insert a relheader element before it.
   XMLMind will insert a relcolspec element as a child of relheader.
5. Click the **Select Next** button or press CTRL + Down Arrow to select the nect child element.
   The relcolspec element will be selected.
6. Set the type attribute to 'task'.
7. CTRL + left click on the relcolspec element in the **node path selection** bar.
   This will insert another relcolspec element after the first. CTRL + left clicking on an element in the **node path selection bar** will insert a duplicate element as the next sibling. This is a very useful shortcut.
8. Create relcolspec elements for concept and reference types.

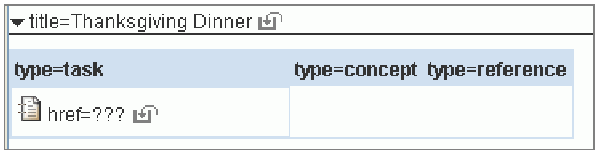


Figure 29: Creating a relationship table

9. Select the relcell element. It is a child of the first relrow element.

   The cell contains a single topicref element.
10. Select the topicref element and point the href attribute to the *How To Make Roast Turkey* task.
11. CTRL+left click on the topicref element in the **node path selection** bar to add another.

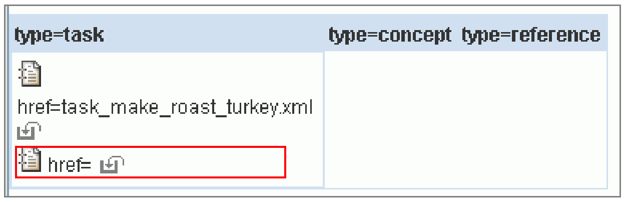


Figure 30: Adding topic references to a relationship table.

12. Add entries for *How To Make Nut Roast* and *How To Check Meat Temperature* to the first task cell.

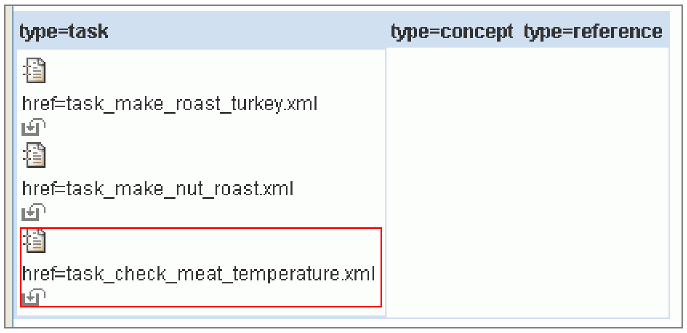

type=task
type=concept
type=reference
href=task_make_roast_turkey.xml
href=task_make_nut_roast.xml
href=task_check_meat_temperature.xml

Figure 31: Completing the first cell

13. Continue to add rows, cells and topic references so that the contents of the table match the figure below.

Table 2: Thanksgiving Relationship Table

type='task'	type='concept'	type='reference'
task_make_roast_turkey.xml task_make_nut_roast.xml task_check_meat_temperature.xml	concept_buying_turkey.xml	ref_shopping_list.xml
task_boil_potatoes task_makle_mashed_potatoes		ref_shopping_list.xml
task_make_cornbread_stuffing.xml task_make_cornbread.xml		ref_shopping_list.xml
task_make_green_bean_casserole.xml		ref_shopping_list.xml

14. Save the map file.

Building Output

Now that you have a full set of topics, supporting image files and a map file, its time to build a document.

The steps involved in building a document varies depending on the output type, but generally includes two main stages:

- Applying XSL stylesheets

  This stage transforms your XML files into a language format that can either be viewed directly by another application or can be compiled or processed into a viewable or printable format.

  In HTML output for the web, this stage transforms the XML content into HTML, including the formatting and layout. As well as the topics, the stylesheets will generate a simple table of contents page.

  In HTML Help output, this stage creates the HTML topic files and also the .hhc (contents) and .hhk (index) files used by the Microsoft HTML Compiler to create a compiled help file (.chm).

  In PDF output, this stage transforms your XML content into XSL-FO. FO stands for Formatting Objects and is an xml-based language used to describe complex documents intended for print.
- Compiling or processing

  This stage converts the files created in the first stage are converted into viewable/printable documents.

  In web HTML output, this stage creates a simple frameset to display the contents and topics.

  In HTML Help output, this stage will run the Microsoft HTML Help Compiler to create the .chm file.

  In PDF output, this stage runs the XEP processor to generate a PDF file from the XSL-FO.

ANT

The build process can be very complex, involving multiple stylesheets and applications. Fortunately the *Open Toolkit* (with Idiom FO plug-in) provides ANT scripts to handle most of this for you.

ANT is a JAVA application that allows you to control complex build processes using custom scripts. In order to launch the DITA builds, you only need to supply a small 'launcher' script with a few simple parameters.

XML Catalogs

An xml catalog creates a mapping between a generic identifier and a location on the local machine. The file *catalog-dita.xml* in the root directory of the *Open Toolkit* provides mappings for the DITA DTD files. This means that you don't have to modify your DOCTYPE reference in your XML files whenever you move the location of your content or create new DITA content in a different location.

DOCTYPE Declarations

This is an example of a DOCTYPE declaration:

```
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "../dtd/task.dtd">
```

The first item, 'task' indicates the root element of the document.

'PUBLIC' indicates that this declaration is using a public identifier.

The third item specifies the public id. This is used to locate the correct mapping in the catalog file.

The final item is a system identifier. This provides the location of the file as a path or uri and is used if the public ID cannot be located in the catalog.

Catalog Entries

In the Open Toolkit catalog, all the entries are contained in a group element:

```
<group xml:base="dtd/">
```

xml:base provides a base location for catalog entry mappings that are children of the group element. This is relative to the catalog file location.

This is the corresponding catalog entry for the DTD referenced in the DOCTYPE declaration above:

```
<public publicId="-//OASIS//DTD DITA Task//EN" uri="task.dtd"></public>
```

The uri is relative to the location specified in xml:base for the entry's parent group element.

catalog-dita-template.xml

This file is copied to the catalog each time you build output using the *Open Toolkit*.

Specialization Files

You should add entries to the catalog template (*dita-catalog-template.xml*) for each new file used by your specialization. The *dita-catalog.xml* file is updated using the contents of *dita-catalog-template.xml* each time that you build. For example, you entries for the recipe information type files would look like this:

```
<public publicId="-//LD//DTD DITA Recipe//EN" uri="recipe.dtd">
 </public>

 <public publicId="-//LD//ELEMENTS DITA Recipe//EN" uri="recipe.mod">
 </public>
```

catalog-ant.xml

You should also add catalog entries for each specilaization file to the catalog-ant.xml file, which also resides in the *DITA Open Toolkit* directory. These entries have a slightly different syntax. For example:

```
 <dtd publicId="-//LD//DTD DITA Recipe//EN"
       location="${dita.dtd.dir}${file.separator}recipe.dtd" />
  <dtd publicId="-//LD//ELEMENTS DITA Recipe//EN"
       location="${dita.dtd.dir}${file.separator}recipe.mod" />
```

ANT Build Files

In your ANT build file, you need to provide the following parameters:

- **DITA Directory:** The directory where the Open Toolkit is installed.
- **Input File:** The map file to use.
- **Output Directory:** Location to save the final documents

- **Transform Type:** The output type, i.e pdf2, xhtml, htmlhelp

Note: For PDF output using the Idiom FO plugin, the transform type is pdf2. Transform type pdf will attempt to use the default FOP process.

In the *Open Toolkit* installation section of this tutorial, we used an ANT build script that allowed us to specify these parameters. You can use this to build any outputs that don't use a plugin but it doesnt give you access to any of the advanced build options, such as filters, that we will be covering later. It is better to create your own 'launcher' build files and faster if you need to build the same output repeatedly.

In the /ant directory of the *DITA Open Toolkit*, you will find sample and template build files that you can use as the basis for your own.

Note: If you save your build files in a different location, make sure that you update the *dita.dir* parameter.

ANT Script Syntax Basics

You don't need a great deal of intricate knowledge of ANT script to work with these simple build files.

The main thing to notice are the property elements which are used to specify the required and optional build parameters. This element takes name and value attributes. Name is the name of the parameter and value is the value to set it to.

The '$' character indicates a parameter. The current value of the parameter will be inserted into the script when it is run. i.e: *$dita.dir* will insert the value of the *dita.dir* parameter that you specify at the top of the file.

If you want to learn more about ANT script, you can examine some of the more complex files in the *Open Tookit* directory, such as conductor.xml.

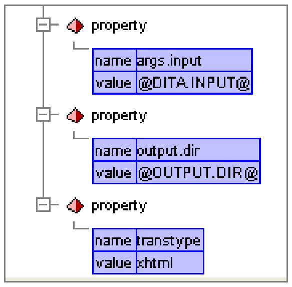
How To Build HTML Output

A copy of the final build file can be found in the *build* directory in the *tutorial* zip file.

1. Open the *template_xhtml.xml* file from the /ant directory of your *DITA Open Toolkit*.
2. Create a *builds* directory under *DITAContent* and save the file into it as *thanksgiving_html.xml*.
3. Open the file you just copied in *XMLMind*.
   First, we will modify the *dita.dir* value to point to the root directory of the *Open Toolkit* and set *basedir* to point to the base directory of our content files.
4. Select the property element named *dita.dir*.
5. Set its value attribute to the path to your *Open Toolkit* i.e. *C:${file.separator}DITA-OT1.3*.

   Note: Use the ${file.separator} variable instead of forward and backward slashes in paths in your build files.

6. Locate the property element named *basedir*.
7. Set its value attribute to '..' the path to your content directory i.e. the parent directory of the build file.
   Next, we will modify the last three property elements.



property

name	args.input
value	@DITA.INPUT@

property

name	output.dir
value	@OUTPUT.DIR@

property

name	transtype
value	xhtml

Figure 32: Creating a build file for xhtml

8. Select the property element named *args.input*.
9. Set its value attribute to *thanksgiving.ditamap.*
10. Select the property element named *output.dir*.
11. Set its value attribute to *tutorial${file.separator}out${file.separator}html*

    Note that the transtype property is set to XHTML to produce html pages. XHTML is a version of HTML with stricter rules. E.g. every tag must be closed. This makes it easier to work with when using XSL stylesheets.
12. Save the file to the
13. Run the *startcmd.bat* batch file from the Open Toolkit directory.

    This will open a command prompt at the *Open Toolkit* root directory.
14. Type *ant -f c:\DITAContent\builds\thanksgiving_html.xml* at the command prompt. If necessary, modify the path to reflect the location of your content.
15. Press **Enter** to build the HTML files.
    You should receive a few screens of feedback from the build process and hopefully BUILD SUCCESSFUL at the end.
16. Got to the *\tutorial\html* directory and take a look at your output.

    Congratulations on your first DITA build!
17. If you did not get a successful build, see *Troubleshooting* on page 70 for some troubleshooting advice.
18. Take a look at the HTML output. Examine how the table of contents looks and how related links are built at the foot if each topic according to your relationship table.

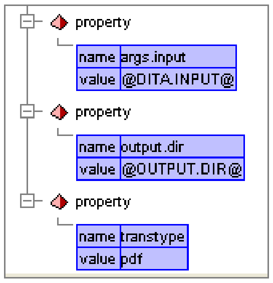
    Tip: You can create a batch file that provides a shortcut to start the build using a certain build file. To do this:

    a) Right-click on the startcmd.bat file in the *DITA Open Toolkit* directory.
    b) Select **Edit**.
    c) Add your ANT command at the end of the script. i.e *ant -f tutorial\thanksgiving_html.xml*.
    d) Save the file under a different name, i.e. thanksgiving_html.bat. Be sure to save it in the *DITA Open Toolkit* root directory.
    Now you can double-click this file to build the HTML output for the Thanksgiving docs.

How To Build PDF Output

Now we will build the PDF version of the Thanksgiving document. There is no template for the *pdf2* transtype (Idiom FO PDF output) so we will use the default pdf template as a base. A copy of the final build file can be found in the *build* directory in the *tutorial* zip file.

1. Open the *template_pdf.xml* file from the /ant directory of your *DITA Open Toolkit*.
2. Save it as *thanksgiving_pdf.xml* in your *builds* directory.
3. Open the file you just copied in *XMLMind*.
   First, we will modify the *dita.dir* value to point to the root directory of the *Open Toolkit* and set *basedir* to point to the base directory of our content files.
4. Select the property element named *dita.dir*.
5. Set its value attribute to the path to your *Open Toolkit* i.e. *C:${file.separator}DITA-OT1.3*.
6. Locate the property element named *basedir*.
7. Set its value attribute to the path to your content directory i.e. '..'
   Next, we will modify the last three property elements.



name	args.input
value	@DITA.INPUT@

name	output.dir
value	@OUTPUT.DIR@

name	transtype
value	pdf

Figure 33: Creating a build file for pdf

8. Set args.input *thanksgiving.ditamap.*
9. Set its output.dir to *utorial${file.separator}out${file.separator}tpdf.*
10. Set the transtype property to *pdf2*
11. Run the *startcmd.bat* batch file from the *Open Toolkit* directory.

    This will open a command prompt at the Toolkit root directory.

    Tip: You can enable logging to a file by adding *-logger org.dita.dost.log.DITAOTBuildLogger* to the end of your ant command. Remember that you can place the ant command into a batch file to save on some typing. See *How To Build HTML Output* on page 40 for instructions. The log will be created in the output directory or you can set the *args.logdir* parameter in your build file to save it somewhere else.

12. Type *ant -f c:\DITAContent\builds\thanksgiving_pdf.xml* at the command prompt.
13. Press **Enter** to build the HTML files.
    You should receive a few screens of feedback from the build process and hopefully BUILD SUCCESSFUL at the end.
14. Got to the *\tutorial\pdf* directory and take a look at your output.
15. If you did not get a successful build, see *Troubleshooting* on page 70 for some troubleshooting advice.
16. Again, take a look at the PDF document. Examine how the table of contents looks and how related links are built at the foot if each topic according to your relationship table. Look at the layout and consider how you might want to change it.

Styles and Formatting

One of the benefits of using DITA as a solo technical writer, is the separation of formatting from content development. Once my stylesheets were set up the way I wanted, I was free to develop my content without worrying about layouts and formatting. This has reduced the time needed to produce new documents.

To customize the formatting and styles of your outputs, you will need to modify the stylesheets used when it is built. You should always try to avoid directly modifying stylesheet files that are a part of the *Open Toolkit* or a plug-in. Changing the Toolkit files themselves will make upgrading to a new version more difficult, since you will lose all your changes when you replace the files. Instead, you should create new stylesheets and reference them from the Toolkit stylesheets or in ANT build script parameters.

Customizing XHTML and HTML Help Output

Both XHTML and HTML Help outputs are formatted by the same set of XHTML stylesheets. When building HTML Help, additional build steps create the content, index, project files etc and compile them into a chm file.

Customizing HTML output can be done using CSS styles or by modifying XSL templates:

- The simplest way to customize HTML output is by modifying the CSS styles used. You can do this by creating an additional CSS stylesheet and adding the appropriate properties to the ANT build script to associate it with your HTML files. You can also associate individual DITA content elements with a CSS class in your custom stylesheet using the outputclass attribute.
- You can add parameters to the ANT build file for your HTML output that specify an HTML running header and footer. You can also include some custom HTML in the head area of each page.
- For more advanced customizations, you can create custom XSL transforms. XSL transforms determine how DITA elements will be processed into HTML tags. i.e. an XSL template specifies that the first topic title in an HTML page should be wrapped in an h1 tag. You can override these transforms so that elements are processed differently or you can add new ones for elements that you have specialized.

The majority of customizations can be accomplished using CSS.

CSS Basics

There are two basic approaches to applying styles in CSS:

- Redefine the style of an existing tag.

  I.e you can redefine the styles for the body tag. These styles would then apply to the entire body section of the HTML page and can be used to apply page backgrounds and set default font styles. You can redefine other tags in the same way; p for paragraph styles, td for table cell styles, li for list items etc.
- Create named styles and apply them to tags in your HTML page individually.

  You can create named styles in the CSS stylesheet by prefacing the name with a period (.) character. You can then apply the style to an HTML tag using the class element.

  i.e. In the CSS: .mystyle {color:red}; in the HTML: <div class="mystyle">.

  In DITA, you can apply a CSS style in the XML content by specifying the custom style name in the otherclass attribute of the DITA element. i.e <step otherclass='mystyle">. This will be processed to produce the correct class attribute in XHTML output types. Styles applied to HTML tags will override any style applied to a parent tag. i.e applying a style to a p tag will override the same style on the body tag.

  CSS stylesheet files are associated with an HTML page using the <link> tag in the <head> section of the HTML. In the *DITA Open Toolkit*, the default spreadsheet "commonltr.css" is copied to the output directory and associated with each HTML page when you build XHTML output. You can also use parameters in your ANT build file to specify include a CSS file.

  If the same style is present in both CSS files, the one in your custom file will override the default, since the custom one is included after the default in the HTML source code.

CSS Stylesheet Editors

CSS stylesheet files use the .css extension. They are plain text files, so can be edited directly using *Notepad* or any text editor. There are numerous visual editors for CSS. Most web page editing applications such as *Macromedia Dreamweaver* have one. I use *Microsoft Visual Studio Express Web Developer* for CSS editing because it's free, it has a visual editor and I already use other features of the application.

How To Customize CSS Styles for XHTML/HTML Help

1. Create a new .css file in the editing application of your choice.
2. Create a body {} element to redefine the HTML body tag styles.

   You can use the body element styles to determine the appearance of the page background and the default styles of all text on the page.

   Tip: In *VS Express Web Developer*, right click on the CSS document and select **Add Style Rule** to create a new CSS element. Right click on the element name and select **Build Style** to open the visual css editor

3. Set the background-color style to a pale yellow such as #ffffcc.
4. Set the font to 11pt Arial.
   The source code for your stylesheet should be as follows:

   ```
   body
   {
       background-color: #FFFFcc;
       font-family: Arial;
       font-size: 11px;
   }
   ```

5. Save the file as *thanksgiving.css* in the tutorial directory.
6. Open the *thanksgiving_html.xml* ANT build file.
7. Insert a new property element after the *transtype* property.
8. Set the name attribute to *args.css*.
9. Set the value attribute to *${dita.dir}${file.separator}tutorial${file.separator}thanksgiving.css*.
10. Add another property with name *args.csspath* and value *css*.
11. Add a third property with name *args.copycss* and value *yes*.

Parameter Name	**Description**
args.copycss	determines whether CSS file will be copied to the output directory.
args.css	The full path to the custom CSS file,
args.csspath	The directory in the output directory that the CSS file will be copied to.

12. Save the build file.
13. Build the HTML output.

How To Customize the Header and Footer in an HTML Page

1. Create a new blank text file and add the following code to it:

   ```
   <p class="header">The Thanksgiving Tutorial!</p>
   ```

2. Save the file into the *tutorial/includes* directory as *header.xml*.

   The HTML code that you use as a header must be XHTML compliant and well formed. The main difference is that XHTML is stricter about closing tags. Every tag must be closed, even if it is empty. I.e a br tag must be written as <br></br> or <br />.

3. Create a second blank text file and add the following code:

   ```
   <p class="footer">&#169; 2007 Lone-DITA</p>
   ```

4. Save the second file into the *tutorial/includes* directory as *footer.xml*.

   You can add styles for use in your header and footer into your custom stylesheet since the HTML code will be copied into the output files and have access to the same CSS styles as the rest of the content.

5. Open the *thanksgiving.css* file in your CSS editor.
6. Add a style named .header.
7. Give the style font-size 14pt and color purple.
8. Add a style named .footer.
9. Give the style font-size 9pt and color darkgray.
   You should end up with CSS code like this:

   ```
   .header
   {
   font-size: 14pt;
   color: purple
   ```

```
}

.footer
{
font-size: 9pt;
color: darkgray
}
```

10. Save the file and close it.
11. Open the *thanksgiving_html.xml* ANT build file.
12. Add another property with name *args.ftr* and set the value to be the absolute path to the *footer.xml* file. This should be something like *file:/C:/DITAContent/tutorial/includes/footer.xml*.
13. Add another property with name *args.hdr* and set the value to be the absolute path to the *header.xml* file.
14. Save the build file and build the XHTML output.
    You should see the header and footer on each page.

Modifying the XSL for XHTML/HTML Help Output

You must be reasonably familar with XSLT before attempting this section of the tutorial. I will be trying to show you shortcuts, but you will still need some basic knowledge of XSLT syntax and how it works.

You should avoid modifying any files distributed with the *Open Toolkit* or you will lose your changes when you upgrade it. Create an override XSL file containing your modified XSLT then update the dita2xhtml.xsl file to include it.

XSL Basics

XSLT processing is based primarily around *templates*. A template is applied to an element or set of elements in the XML source that matches the given search parameters. The template then determines how those XML elements will be transformed into the desired output.

Example: A template could be created to match all <keyword> elements in the XML source. It could then output the content of the keyword element wrapped in a <span class="keyword"> tag in the HTML file.

DITA Open Toolkit XHTML Transforms

The high-level XSLT stylesheet that is applied to XHTML outputs is <dita>\xsl\dita2xhtml.xsl. However, this file consists mostly of a collection of includes. The includes reference the stylesheets contained in the <dita>\xsl\xslhtml directory.

dita2htmlImpl.xsl is the stylesheet that contains the majority of the transforms for XHTML but there are also several others that handle specific aspects of processing and also processing of specialized domains such as the *programming* and *highlight*.

XSL File Name	Description
dita2htmlImpl.xsl	Contains most common transforms for XHTML output.
rel-links.xsl	Contains templates for xref and related-links.
taskdisplay.xsl	Contains templates for processing special task elements, such as steps.
refdisplay.xsl	Contains templates for processing special reference elements such as property.
pr-d.xsl, hi-d.xsl, ut-d.xsl, ui-d.xsl, sw-d.xsl	Contains templates for processing elements in the relevant domain.
map2TOC.xsl, mapwalker.xsl	Conatins templates for generating a TOC from the DITA map file.

dita2htmlImpl.xsl

You can learn a great deal about XSLT and how it is used by the *DITA Open Toolkit* by looking through this file.

The easiest way to find a template is to search for its class attribute. This is the method that the XSL templates themselves use to locate specific elements in your DITA content files.

The class attribute is generated by the DITA DTD. The class attribute value is assigned to each element and is used by the XSLT processor to correctly identify the range of content that each template should process.

Caution: Do not edit the class attribute manually when editing your content. You will break the XSLT transforms.

The attribute value is in the form domain/element.

Element	Class Value
p	topic/p
steps	topic/ph task/steps
refbody	topic/body reference/refbody
i	topic/ph hi-d/i

You will notice that elements that are part of a specialized topic type (task, reference etc) or a specialized domain (programming,highlight) have two values. The first value is the general element that the specific one is specilaized from. This will be explained in more detail in the *Specialization* section.

Tip: You can find the class attribute of any element by examining the attribute list in *XMLMind*.

So to find the template for paragraphs, you can search *dita2htmlImpl.xsl* for 'topic/p'. You will see a template that is applied to all elements with a class attribute that contains *topic/p*. This is the template that you would override in order to modify the way in which paragraphs are output.

How To Customize XSL for XHTML Output

In this section we are going to create a simple XSL customization that will display keywords in red or blue depending upon their otherprops attribute. In the real world, specialization may be a more effective way to acheive this type of result.

1. Create a blank text file named *custom_html.xsl* and save it into the *xsl/xslhtml* directory in the *DITA Open Toolkit* installation directory.
2. Enter the following text into your *custom_html.xsl* file to create the stylesheet 'shell'.

   ```
   <?xml version="1.0" encoding="UTF-8" ?>
   <!DOCTYPE xsl:stylesheet>

   <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">



   </xsl:stylesheet>
   ```

   Next we will create the template itself. It is always a good idea to copy the default template from the *Open Toolkit* file into your custom XSL file and modify it. This way you can be careful not to lose essential processing that the *Open Toolkit* may perform to handle flagging, filtering, attributes etc.
3. Open *dita2htmlImpl.xsl* and search for t*opic/keyword*.
4. Copy the template into your custom file inside the xsl:stylesheet element..
   The code should look like this:

   ```
   <xsl:template match="*[contains(@class,' topic/keyword ')]" name="topic.keyword">
    <span class="keyword">
   ```

```
  <xsl:call-template name="commonattributes"/>
  <xsl:call-template name="setidaname"/>
  <xsl:call-template name="flagcheck"/>
  <xsl:call-template name="revtext"/></span>
</xsl:template>
```

Notice the four templates that are being called by the default template. Since we know that no output processing is really performed on keyword elements, we know that these must be some kind of behind the scenes processing and we should probably leave them alone.

5. Insert the following code before the `<span class="keyword">` line:

```
 <xsl:choose>
   <xsl:when test="@otherprops='red'">
```

This is a simple XSLT structure that allows conditional processing. In this case, the processing contained by the `xsl:when` element will be performed if the `otherprops` attribute has the value 'red'.

6. Modify the remaining code until you have the following:

```
<xsl:template match="*[contains(@class,' topic/keyword ')]" name="topic.keyword">
  <xsl:choose>
    <xsl:when test="@otherprops='red'">
      <span class="red_keyword">
        <xsl:call-template name="commonattributes"/>
        <xsl:call-template name="setidaname"/>
        <xsl:call-template name="flagcheck"/>
        <xsl:call-template name="revtext"/>
      </span>
    </xsl:when>
    <xsl:otherwise>
      <span class="blue_keyword">
        <xsl:call-template name="commonattributes"/>
        <xsl:call-template name="setidaname"/>
        <xsl:call-template name="flagcheck"/>
        <xsl:call-template name="revtext"/>
      </span>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

This template works by applying a different CSS class to the keyword depending upon the value of the `otherprops` attribute.

7. Save the *custom_html.xsl* file.
8. Add two new named styles to your *thanksgiving.css* file.

   .red_keyword should set the color style to red and *.blue_keyword* should set the color to blue.

```
.red_keyword
{
    color: red;
}
.blue_keyword
{
    color: blue;
}
```

9. Browse to the *xsl* directory in the *DITA Open Toolkit*.
10. Make a copy of the *dita2xhtml.xsl* file and call it *dita2xhtml_custom.xsl*.
11. Open the copy and add the following line to the end of the list of `xsl:import` elements.

```
<xsl:import href="xslhtml/custom_html.xsl"></xsl:import>
```

Caution: You must add your import instruction after the import of the file containing the default version of your template (*dita2htmlImpl.xsl* in this case). Otherwise the default template will overwrite your custom one.

12. Save the copy and close it.

13. Open the *thanksgiving_html.xml* ANT build file and add a new property element.
14. Give the property name *args.xsl* and value *${dita.dir}${file.separator}xsl${file.separator}dita2xhtml_custom.xsl.*

    This line instructs the build process to use an alternate stylesheet. You can only override *dita2xhtml.xsl* for XHTML outputs. So you will need to create a custom stylesheet, import it into a custom copy of *dita2xhtml.xsl* and then point the args.xsl parameter to that copy.
15. Open one of the tutorial topic files and wrap a few words or phrases with a keyword element. Set the otherprops attributes to *red* or *blue* to test the custom XSLT
16. Build the XHTML output.

Testing and Debugging XSL Customizations

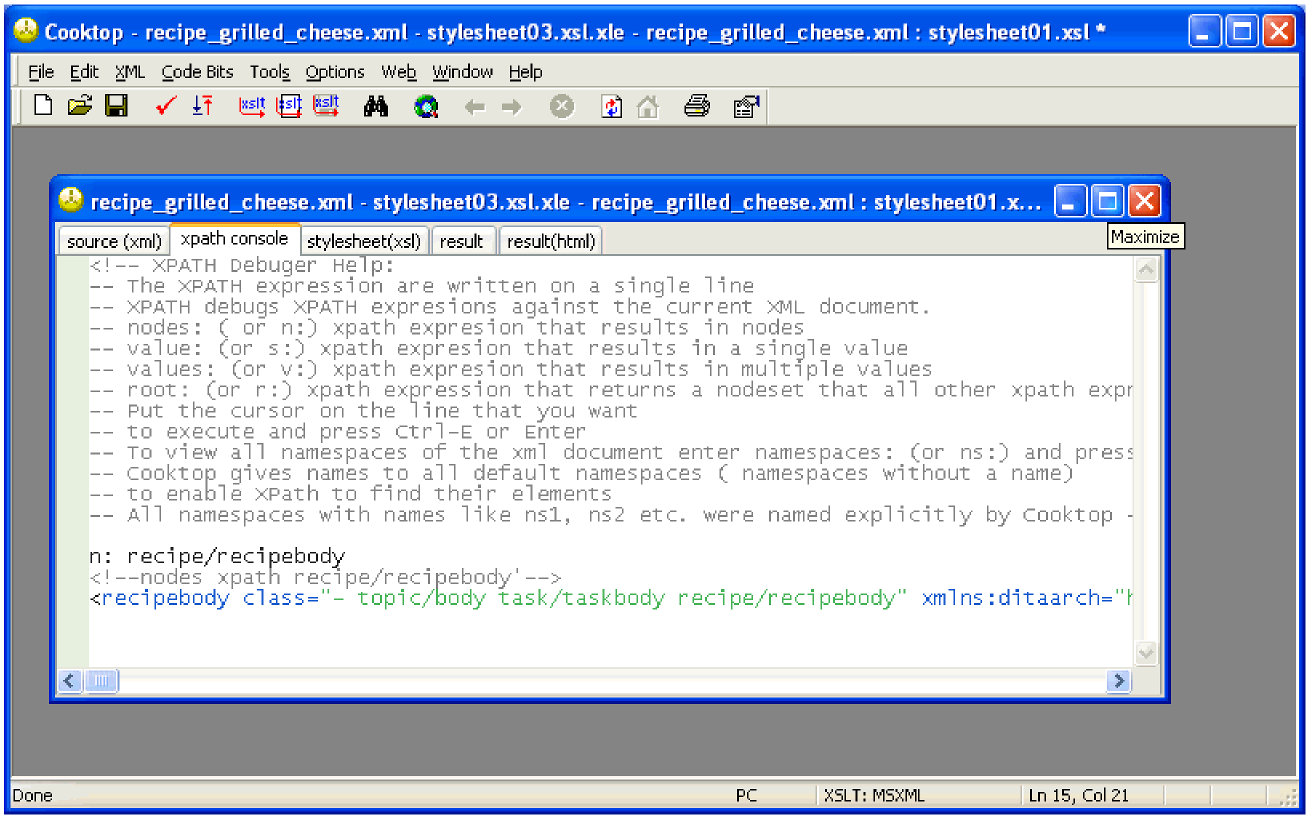
It can be time-consuming to have to build the entire document to test and debug your XSL customizations. As an alternative to this, you can apply the XSL stylesheets directly to a topic file to see if the results are what you expect.

I use an application called *Cooktop* for this purpose. *Cooktop* allows you to load an XML file and an XSL stylesheet, run the transform and view the resulting output and source code. There are various other applications available that can perform the same task however.

You should transform your XML using *dita2xhtml.xsl*. All the other XSL stylesheets are imported from there.

Your custom CSS stylesheets will not be attached to the resulting HTML file when you use this method. You can link it manually by editing the source code or just look at the source code to see if it has been generated correctly.

Cooktop also allows you to test fragments of XSL code against an XML file. This allows you to test the results of new templates individually.


Cooktop - recipe_grilled_cheese.xml - stylesheet03.xsl.xle - recipe_grilled_cheese.xml : stylesheet01.xsl *
File
Edit
XML
Code Bits
Tools
Options
Web
Window
Help
recipe_grilled_cheese.xml - stylesheet03.xsl.xle - recipe_grilled_cheese.xml : stylesheet01.x...
Maximize
source (xml)
xpath console
stylesheet(xsl)
result
result(html)
Done
PC
XSLT: MSXML
Ln 15, Col 21

```
<!-- XPATH Debuger Help:
-- The XPATH expression are written on a single line
-- XPATH debugs XPATH expresions against the current XML document.
-- nodes: ( or n:) xpath expresion that results in nodes
-- value: (or s:) xpath expresion that results in a single value
-- values: (or v:) xpath expresion that results in multiple values
-- root: (or r:) xpath expression that returns a nodeset that all other xpath expr
-- Put the cursor on the line that you want
-- to execute and press Ctrl-E or Enter
-- To view all namespaces of the xml document enter namespaces: (or ns:) and press
-- Cooktop gives names to all default namespaces ( namespaces without a name)
-- to enable XPath to find their elements
-- All namespaces with names like ns1, ns2 etc. were named explicitly by Cooktop -

n: recipe/recipebody
<!--nodes xpath recipe/recipebody'-->
<recipebody class="- topic/body task/taskbody recipe/recipebody" xmlns:ditaarch="h
```

Figure 34: Cooktop XSL Debugger

Customizing Idiom FO (pdf2) Output

Before attempting this section of the tutorial, you should have a good understanding of XSLT and XSL-FO.

http://www.w3schools.com/xslfo/default.asp

http://www.w3schools.com/xsl/default.asp

XSLT is used to transform DITA XML into XSL-FO. The XSL-FO is processed by the XEP RenderX FO processor into a PDF file.

There are several way in which you can customize pdf2 output, depending upon the desired results.

- Edit the vars file: The vars file is used to store common strings, such as product names and text used in running headers and footers. There are multiple vars files. Each applies to a different language. If you are only producing documentation in English, you only need to change the English vars file.
- Modify attribute sets: Attribute sets play a similar role to CSS. They are a library of named style definitions that can be applied to XSL-FO elements when your XML is transformed.
- Modify master page layouts: page layout definitions are stored in a single file and can be modified. Properties such as page size, margins, regions etc can be defined. You can have multiple layouts for different sections of the document (contents, chapter, index etc). You can also define the sequence of pages in different sections. These determine the master page layouts to use for the first, last, odd and even pages.
- Modify XSLT transforms: As with XHTML, you can modify the XSLT transforms to process DITA elements in a different way.

The Idiom FO plug-in has a customization architecture that allows you to override the default processing without modifying the plug-in files themselves. This architecture can be used with vars, attributes, layouts and xslt stylesheets.

How To Edit the Vars File

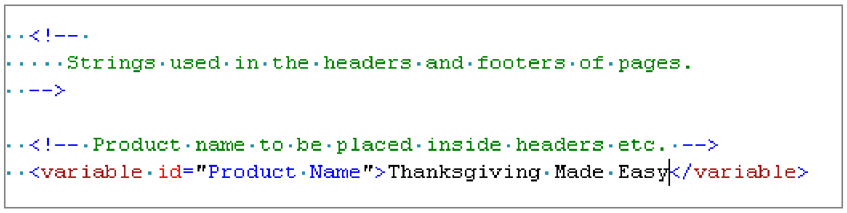
The vars files contain many useful settings that you can customize easily. Each one is commented in the default file, so you can tell what does what. Currently, I edit the product name manually when I need to change it in my pdf output. I suspect you could create an XSL transform to update the custom vars file and include it in your build process if you wanted to eliminate manual changes..

1. Copy the file *en_US.xml*, which is located in *<dita>\demo\fo\cfg\common\vars* into *<dita>\demo\fo\Customizations\common\vars*.

   All customizations reside in the appropriate sub-directory of the *Customization* directory.

2. Open the copy.

   The first variable defined in the file is "Product Name". If you look down the file, you will see that this variable is re-used throughout the file in variable that define text for headers. You only need to edit the *Product Name* variable to update all these.

```
  <!-- 
     Strings used in the headers and footers of pages.
  -->

  <!-- Product name to be placed inside headers etc. -->
  <variable id="Product Name">Thanksgiving Made Easy</variable>
```

   Figure 35: Editing the vars file

3. Edit *Product Name* so that its value is *Thanksgiving Made Easy*.
4. Save the file.
5. Build your pdf2 output.

Notice that the product name has been updated in the page headers.

PDF Attribute Sets

Attribute sets are named sets of styles and properties that can be applied to XSL:FO elements. Attribute sets are stored in their own files, similar to CSS which makes it easy to modify them without having to change the XSL transforms themselves.

The default attribute sets are stored in *<dita>\demo\fo\cfg\fo\attrs*. As always, you should create a custom override file rather than modifying the default files themselves.

Attribute File	**Description**
commons-attr.xsl	Contains attributes for standard body elements such as paragraphs, headings, images, notes, keywords etc.
lists-attr.xsl	Contains attributes for list items including ordered, unordered and specialized lists for task topics.
tables-attr.xsl	Contains attributes for tables and simpletables.
static-content-attr.xsl	Contains attributes for headers and footers.
index-attr.xsl	Contains attributes for index sections.
toc-attr.xsl	Contains attributes for tables of contents sections.
front-matter-attr.xsl	Contains attributes for the front cover of the pdf document.
hi-domain-attr.xsl, pr-domain-attr.xsl, ui-domain-attr.xsl, sw-domain-attr.xsl	Contains attributes for elements in specialized domains provided with the DITA Open Toolkit.

Syntax of Attribute Sets

You can look at the file *commons-attr.xsl* to see the attribute syntax. The element xsl:attribute-set is the containing element for each set. It takes a name attribute which defines how the set is referenced in the XSLT stylesheets. Each attribute is defined using the xsl:attribute element. This element has a name attribute which identifies the name of the attribute and a value attribute which identifies its value.

```
<xsl:attribute-set name="__border__top">
      <xsl:attribute name="border-top-color">black</xsl:attribute>
      <xsl:attribute name="border-top-width">thin</xsl:attribute>
</xsl:attribute-set>
```

How To Customize Attribute Sets

1. Create a file called *custom_attr.xsl* in the *<dita>\demo\fo\Customization\fo\attrs*.
2. Enter the following code:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:fo="http://www.w3.org/1999/XSL/Format"
   version="1.0">
  <xsl:attribute-set name="topic.title">
   <xsl:attribute name="font-family">Sans</xsl:attribute>
   <xsl:attribute name="border-bottom">3pt solid black</xsl:attribute>
   <xsl:attribute name="margin-top">0pc</xsl:attribute>
   <xsl:attribute name="margin-bottom">1.4pc</xsl:attribute>
   <xsl:attribute name="font-size">18pt</xsl:attribute>
   <xsl:attribute name="font-weight">bold</xsl:attribute>
   <xsl:attribute name="color">darkblue</xsl:attribute>
   <xsl:attribute name="padding-top">1.4pc</xsl:attribute>
   <xsl:attribute name="keep-with-next.within-column">always</xsl:attribute>
  </xsl:attribute-set>
</xsl:stylesheet>
```

This modifies the top level title attribute set, called topic.title, so that the text is dark blue.

Note: You only need to include attribute sets that you are overriding in this file.

3. Save the file.
4. Browse to the *Customization* folder and open the *catalog.xml* file in a text or source code editor.

   The catalog file contains entries that identify which files to use as overrides. If this is a new install of the Idiom plug-in, the file will be named *catalog.xml.orig*. Rename it to *catalog.xml* before proceeding.
5. Locate the 'custom attributes' section and un-comment the line.
6. Modify the uri attribute so that it points to your custom file.
   The uncommented line should be:

   ```
   <uri name="cfg:fo/attrs/custom.xsl" uri="fo/attrs/custom_attr.xsl"/>
   ```

   Note: Even though the name value looks like a file path, it doesnt change even if you modify the filename. Only the uri attribute needs to be edited.
7. Save the *catalog.xml* file.

Customizing XSLT for PDF2 Output

XSLT can be overridden for PDF output in the same way as for XHTML output.

The default XSLT stylesheets are stored in *<dita>\demo\fo\xsl\fo*. As always, you should create a custom override file rather than modifying the default files themselves. The following table provides an overview of the contents of some of the main stylesheets. Other stylesheets provide transforms for elements in the specialized domains provided with the *DITA Open Toolkit*.

XSLT Stylesheet	**Description**
commons-xsl	Transforms for common body elements.
links.xsl	Transforms for related-links sections and xref.
lists.xsl	Transforms for un-ordered and ordered lists, steps sections in tasks, linklists etc.
tables.xsl	Transforms for tables and simpletables.
front-matter.xsl	Transforms to create the front page of the PDF.
preface.xsl	Transforms to create the preface section.
toc.xsl	Transforms to create the Table of Contents section.
index.xsl	Transforms to create the Index section
bookmarks.xsl	Transforms to create the PDF bookmarks.
static-content.xsl	Transforms to create headers and footers for various sections of the PDF.

Like vars and attributes, custom XSL transforms are placed in an override file in a sub-directory of the *Customization* directory and an entry added to the *catalog.xml* file.

An Example XSLT Template Walkthrough

Search for 'topic/image' in the file *commons.xsl* (in the *<dita>\demo\fo\xsl\fo* directory). This should locate the template used to display images in your PDF output. The template contains code that governs the placement of the image according to its placement attribute and whether it is in a fig element.

```
<xsl:template match="*[contains(@class,' topic/image ')]">
    <!-- build any pre break indicated by style -->
```

```
        <xsl:choose>
            <xsl:when test="parent::fig">
                <!-- NOP if there is already a break implied by a parent property -->
            </xsl:when>
            <xsl:otherwise>
                <xsl:if test="not(@placement='inline')">
                    <!-- generate an FO break here -->
                    <fo:block>&#160;</fo:block>
                </xsl:if>
            </xsl:otherwise>
        </xsl:choose>

        <xsl:choose>
            <xsl:when test="not(@placement = 'inline')">
<!--                <fo:float xsl:use-attribute-sets="image__float">-->
                    <fo:block xsl:use-attribute-sets="image__block" id="{@id}">
                        <xsl:call-template name="placeImage">
                            <xsl:with-param name="imageAlign" select="@align"/>
                            <xsl:with-param name="href" select="@href"/>
                            <xsl:with-param name="height" select="@height"/>
                            <xsl:with-param name="width" select="@width"/>
                        </xsl:call-template>
                    </fo:block>
<!--                </fo:float>-->
            </xsl:when>
            <xsl:otherwise>
                <fo:inline xsl:use-attribute-sets="image__inline" id="{@id}">
                    <xsl:call-template name="placeImage">
                        <xsl:with-param name="imageAlign" select="@align"/>
                        <xsl:with-param name="href" select="@href"/>
                        <xsl:with-param name="height" select="@height"/>
                        <xsl:with-param name="width" select="@width"/>
                    </xsl:call-template>
                </fo:inline>
            </xsl:otherwise>
        </xsl:choose>

        <!-- build any post break indicated by style -->
        <xsl:choose>
            <xsl:when test="parent::fig">
                <!-- NOP if there is already a break implied by a parent property -->
            </xsl:when>
            <xsl:otherwise>
                <xsl:if test="not(@placement='inline')">
                    <!-- generate an FO break here -->
                    <fo:block>&#160;</fo:block>
                </xsl:if>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
```

The first section inserts a break if the placement attribute is not inline, or if the image is in a fig element. If the image is in a fig element, the template for fig should already have supplied the break.

The next section specifies a\n attribute set to use, depending upon the placement attribute. It then calls the *placeImage* template to actually output the image using the correct alignment, size and image source. Notice that the placeImage template is called from inside an <fo:inline> element for an image with placement attribute 'inline' and in a <fo:block> element for images with placement 'break'.

The final section adds another break after the image if it is not an inline image or in a fig element.

This template demonstrates some of the common aspects of XSLT such as conditional structures (choose-when-otherwise) and calling named templates. It also demonstrates how attribute sets can be applied to control output.

Note: The template *placeImage* is just below this one in the commons.xsl file if you want to look at it.

How To Customize XSLT for PDF2 Output

In this section, we will create custom XSL for pdf2 output that has the same effect as the custom XSL we created for XHTML output. Keywords will be displayed in red or blue according to the otherprops attribute.

1. Browse to *<dita>/demo/fo/Customization/fo/xsl*.
2. Make a copy of the *custom.xsl.orig* file.
3. Rename the copy to *custom_pdf.xsl*.
4. Open the file in an XML source code editor.
   You should see the following code:

   ```
   <?xml version='1.0'?>
   <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:fo="http://www.w3.org/1999/XSL/Format"
     version="1.1">

   </xsl:stylesheet>
   ```

   This provides an empty 'shell' for your custom templates.
5. Paste the following code between the stylesheet elements.

   ```
   <xsl:template match="*[contains(@class,' topic/keyword ')]">
     <xsl:choose>
       <xsl:when test="@otherprops='red'">
         <fo:inline xsl:use-attribute-sets="keyword_red" id="{@id}">
           <xsl:apply-templates/>
         </fo:inline>
       </xsl:when>
       <xsl:otherwise>
         <fo:inline xsl:use-attribute-sets="keyword_blue" id="{@id}">
           <xsl:apply-templates/>
         </fo:inline>
       </xsl:otherwise>
     </xsl:choose>
    </xsl:template>
   ```

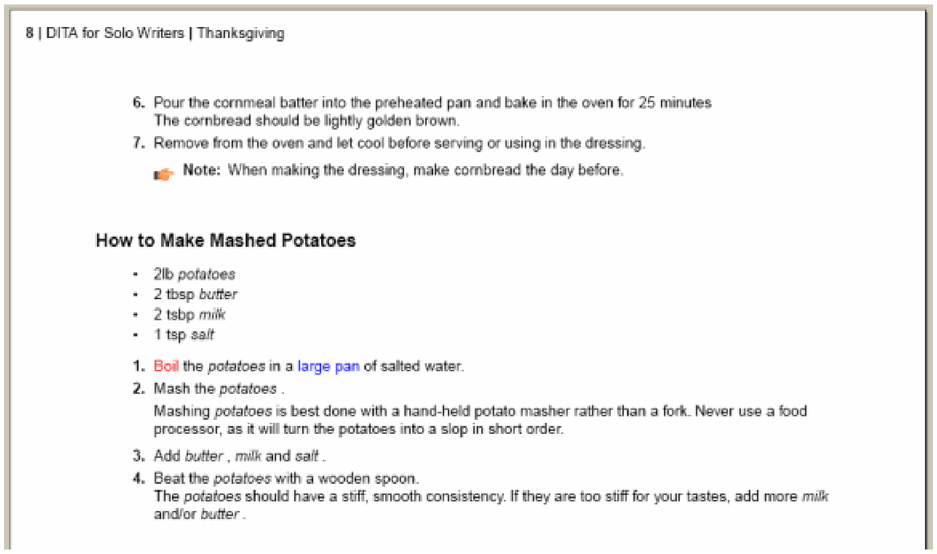
   This XSL works in a similar way to the XHTML version, except that the transformed code is XSL-FO rather than XHTML..

   Note how the use-attribute-sets attribute is used to assign the specified attribute set to the XSL-FO element.

   Next, we need to create an entry in the catalog.xml file to point to the override XSL file.
6. Browse to the *Customization* folder and open the *catalog.xml* file in a text or source code editor.
7. Locate the 'Custom XSL code entry' section and un-comment the first line.
   The uncommented line should be:

   ```
   <uri name="cfg:fo/xsl/custom.xsl" uri="fo/xsl/custom_pdf.xsl"/>
   ```
8. Save the *catalog.xml* file.

   The *catalog.xml* file is used to map override files for attributes and XSLT transforms.
9. Build the PDF2 output and check that the override. is working.



8 | DITA for Solo Writers | Thanksgiving

6. Pour the cornmeal batter into the preheated pan and bake in the oven for 25 minutes
   The cornbread should be lightly golden brown.
7. Remove from the oven and let cool before serving or using in the dressing.
   Note: When making the dressing, make cornbread the day before.

How to Make Mashed Potatoes

- 2lb *potatoes*
- 2 tbsp *butter*
- 2 tsbp *milk*
- 1 tsp *salt*

1. Boil the *potatoes* in a large pan of salted water.
2. Mash the *potatoes* .
   Mashing *potatoes* is best done with a hand-held potato masher rather than a fork. Never use a food processor, as it will turn the potatoes into a slop in short order.
3. Add *butter* , *milk* and *salt* .
4. Beat the *potatoes* with a wooden spoon.
   The *potatoes* should have a stiff, smooth consistency. If they are too stiff for your tastes, add more *milk* and/or *butter* .

Figure 36: PDF output with the XSL customization

Specializing Information Types

The *DITA Open Toolkit* user guide says of information types:

> Topics that answer different kinds of questions can be categorized as different information types.

Concept, *Task* and *Reference* are examples of information types. Tasks, for example, answer the question 'how do I do it?'. Specialized information types answer more specific questions.

In this section of the tutorial we are going to develop a *Recipe* specialization. This type answers the question 'how do I make this dish?' which is a more specific version of 'how do I do it?'. This tells us that the *Recipe* type is a specialized version of a task. A new information type must always be a specialized version of an ancestor type.

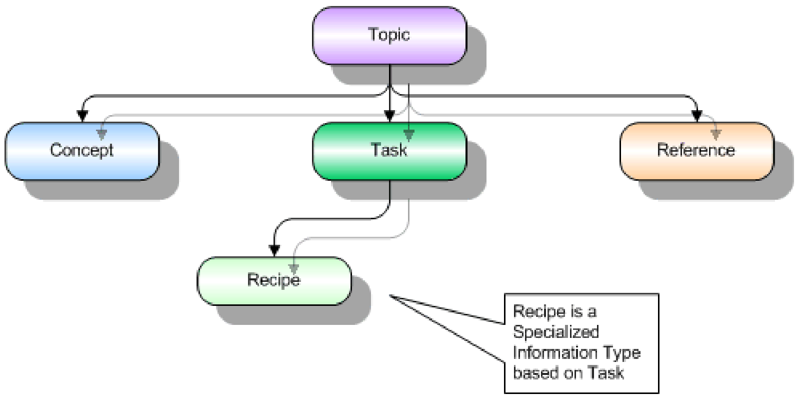


Figure 37: Information types

Specialized information types help you to present content in a consistent way. This helps users absorb the information more readily.

Specialized information types also benefit writers, since it makes it easier to remain consistent when structuring topics. You can also use a vocabulary specific to the type of content. In the recipe type, for example, the ul element works perfectly well for a list of ingredients, but an element called ingredientlist is much more intuitive for content writers (even if thats you!). It also allows XSL transforms to be created for the ingredientlist element. In this way, you can have processing and formatting just for ingredient lists that doesn't apply to other types of un-ordered list. However, the specialization hierarchy means that you don't have to process ingredientlist specifically, since it can still be handled by the processing for it's ancestor element, ul.

Each information type specialization is comprised of two main files.

- The .dtd file which is a 'shell' dtd. Its main purpose is to integrate any required specialized domains and base information types.
- The .mod file which defines the elements and structure of the new information type.

You can specialize map information types as well as topic types. The bookmap, for example, is a specialized map.

Designing the Recipe Topic Type

Attention: You will need to download the specialization files to complete this section.

Most recipes, regardless of where you find them, follow the same basic pattern. The name of the recipe at the top, perhaps followed by a short description of the dish, a list of ingredients then the instructions for making it. Sometimes you may get a serving suggestion or some nutritional information at the end.

A person who is planning to create a new recipe book is unlikely to divert significantly from this structure. If you were suddenly to decide to put the ingredient list at the end, it would be illogical and confusing. Specialization allwos you to define the structure so that it is logical and consistent with reader's expectations.

```
<recipe>

 <title></title>

 <ingredientlist>
  <ingredient></ingredient>
  <ingredient></ingredient>
  <ingredient></ingredient>
 </ingredientlist>

 <instructions>
  <instruction></instruction>
  <instruction></instruction>
  <instruction></instruction>
 </instructions>

 <serving></serving>
 <nutrition></nutrition>

</recipe>
```

Figure 38: Structure of a recipe

Table 3: Recipe Topic Type Elements

Base Type - Topic	Base Type - Task	Specialized Type - Recipe
(topic.mod)	(task.mod)	(recipe.mod)
topic	task	recipe
title		
body	taskbody	recipebody
section	context	description
ul		ingredientlist
li		ingredient

Base Type - Topic	**Base Type - Task**	**Specialized Type - Recipe**
ol	steps	instructions
li	step	instruction
section	postreq	serving
		nutrition
related-links		

The table above lists the elements that we want to include in a recipe topic, based on the recipe structure described at the beginning of this section of the tutorial. This table demonstrates an important aspect of specialization. The new information type is *specialized* from an existing type, either the generic *topic* type or one of the base types specialized from topic in the *Open Tookit* (*concept, reference, task*). A recipe is a specific type of task. In fact we used task topic types for the recipes in the previous sections and it worked reasonably well. So we can base our new type from the task type. This is an important concept. By specializing new information types rather than creating completely new ones, your content will still be compatible with DITA implementations that do not include your specialization.

In the table, you can see the base element of each new element in the recipe information type. Blank cells to the right mean the element is not specialized any further. i.e title is used as is from the topic information type. ingredientlist is specialized directly from ul. You may wonder why it isnt specialized from steps-unordered in the task type. The reason is that our simple ingredient elements do not need the more complex structure of the step element, which are the children of steps-unordered. Our instructions however, do need the more complex step structure so ingredients/ingredient are specialized from steps/step.

How To Create the Recipe Module

In the tutorial files for this section, there is a template file and completed file for both recipe specialization files. You can complete the tutorial using the template as a starting point and the completed file as a guide. I use *Visual Studio Express Web Developer* as my editor but any good XML source editor will work.

1. Open the recipe_template.mod file in your XML editor. You may also want to open the final recipe.mod file for reference also.

   The first section of all DITA DTD files in the *Open Toolkit* is a set of comments containing information about the specialization module. This is based on the same section from the Open Toolkit .mod files. It is worth copying and modifying this section into new modules for your own and others' benefit.

   In particular, be sure to include your own public identifier for the module. This will be used later in file catalogs to help maintain and resolve relative links. Use the same pattern as the Open Toolkit for your public ids, so instead of *PUBLIC "-//OASIS//ELEMENTS DITA Task//EN"* (from task.mod), I use *PUBLIC "-//LD//ELEMENTS DITA Recipe//EN"*. The general syntax for Formal Public Identifiers in DOCTYPE declarations is *-//owner//keyword description//language*. You can read more about DOCTYPE at the web sites listed in the **Web Links** section below.

2. The section 'ARCHITECTURE ENTITIES' section defines the arch-att entity and the namespace for the DITAArchVersion attribute. arch-att is a set of attributes that define the version of DITA being used. These attributes must be present on the topic level element (recipe in this case). Leave this section as it is.

   The next section is labelled 'SPECIALIZATION OF DECLARED ELEMENTS'. In this section, we declare an entity which is used to define how (or if) topics of different types can be nested. This entity must be named using the following pattern: *topic-level-element*-info-types, so in our case it will be recipe-info-types.

3. Edit the SPECIALIZATION OF DECLARED ELEMENTS section so that it matches the following code:

   ```
   <!ENTITY % recipe-info-types
                    "%info-types;"                    >
   ```

   This is the default declaration. We will define it with the actual topic nesting behaviour that we want to implement in the .dtd file in the next section. This default declaration is required in the .mod however.

4. In the next section, 'ELEMENT NAME ENTITIES', we must define an entity for every new element in the specialiization using the following code:

   ```
   <!ENTITY % recipe "recipe"                    >
   <!ENTITY % description   "description"                      >
   <!ENTITY % ingredients      "ingredients"                      >
   <!ENTITY % recipebody    "recipebody"                      >
   <!ENTITY % ingredientlist  "ingredientlist"                      >
   <!ENTITY % ingredient  "ingredient"                      >
   <!ENTITY % instructions "instructions"                      >
   <!ENTITY % instruction "instruction"                      >
   <!ENTITY % serving "serving"                      >
   <!ENTITY % nutrition "nutrition"                      >
   ```

   The next section defines the included-domains attribute. This entity is used as the value of the domains attribute on the topic level element (recipe). This element specifies which specialized domains should be included in this information type. Like the topic nesting entity, this one is redefined in the .dtd to actually include specialized domains into the information type. The declaration here is a default and is required in the .mod file.

   In the next section, ELEMENT DECLARATIONS, we finally get to create the new elements. The easiest way to do this is to start by copying the definition of the new element's closest ancestor into your .mod file. This gives you a base to start from.

How to Create the Element Declarations in recipe.mod

When declaring elements, its important to remember that the new content model must be more restrictive than its ancestor's. The recipe elements' children, for example, must map to elements that are children of the task element, or to elements specialized from them.

1. Open the *task.mod* file from the *Open Toolkit dtd* directory.
2. Copy the declaration for the task element from task.mod into your 'ELEMENT DECLARATIONS' section. You should have the following code:

   ```
   <!--                LONG NAME: Task                    -->
   <!ELEMENT task         ((%title;), (%titlealts;)?,
                          (%shortdesc; | %abstract;)?,
                          (%prolog;)?, (%taskbody;)?,
                          (%related-links;)?, (%task-info-types;)* )  >
   <!ATTLIST task
            id        ID                         #REQUIRED
            conref     CDATA                        #IMPLIED
            %select-atts;
            %localization-atts;
            %arch-atts;
            outputclass
                      CDATA                      #IMPLIED
            domains    CDATA             "&included-domains;"    >
   ```

3. Change the comment to read 'LONG NAME: Recipe'. This is for readabilty only as far as I know.
   The !ELEMENT section declares the name of the new element, which elements can be it's children and the quantity and number of those child elements that can be present.
4. Change the word 'task' imediately following !ELEMENT to 'recipe'.
   This names our new element.
5. The code in parentheses defines the children of the recipe element.

   A comma ',' separating two child elements indicates that they must be present in the order in which they are listed in the declaration. A vertical line '|' indicates that either element can be present.

   One of three different characters can appear immediately following an element or group of elements:

   - ?: Either 1 or 0 of the preceding element(s) can be present.
   - +: 1 or more of the preceding element(s) can be present. There must be at least one.

- *: 0 or more of the preceding element(s) can be present.

If there is no preceding character, the element or group of elements must appear once.

All elements are referred to by the ENTITY that was declared for them. These are declared either in the ELEMENT NAME ENTITIES section of this mod file or in the .mod file for the ancestor information type in which they are defined. i.e %title; is defined in the base information type.

Before editing it, lets take a look at the definition for the task element and see how it matches up with what we already know about task topics. The first element title must appear once. titlealts can appear once but is not required. shortdesc or abstract are defined as a group by their surrounding parentheses. The entire group can appear 0 or 1 times. Effectively, this means that you can have one shortdesc or one abstract or neither but not both. Pay attention to parentheses and the positions of the quantity modifiers when dealing with groups of elements. Following that, you can have a single, optional prolog element, a single taskbody element, which is required, followed by a single, optional related-links element.

Finally, the task-info-types is the topic nesting definition described earlier. By referencing this entity here, it allows any topic type that it defines to be nested inside the topic-level element recipe.

Note: The actual value of the topic nesting entity is defined in the .dtd file. The .mod file contains the default decalaration.

6. The major difference between the task element and our recipe element, is that the main body element will be recipebody. We are also going to remove the abstract element from the declaration as it isn't really appropriate to recipes. Finally we need to supply the topic nesting entity for our specialization which is recipe-info-types.

   After making these changes, you should have the following !ELEMENT declaration for recipe:

```
<!ELEMENT recipe     (%title;, (%titlealts;)?, (%shortdesc;)?,
                      (%prolog;)?, (%recipebody;)?, (%related-links;)?,
                      (%recipe-info-types;)* )                    >
```

How to Declare Attributes in recipe.mod

After the element is declared, the next task is to declare the element's attributes. The !ATTLIST declaration defines the attributes for the new element. The attribute list should be the same or a subset of the attribute list of the ancestor element. In this case task is the ancestor element.

```
<!ATTLIST task
         id        ID                          #REQUIRED
         conref    CDATA                            #IMPLIED
         %select-atts;
         %localization-atts;
         %arch-atts;
         outputclass
                CDATA                          #IMPLIED
         domains    CDATA               "&included-domains;"    >
```

Figure 39: !ATTLIST for task in task.mod

1. Change 'task' to 'recipe' immediately following !ATTLIST. This is the name of the element that we are declaring attributes for.

   Some groups of attributes are referenced by a single entity name because those groups are used in a large number of elements. A few examples are:

   - **select-atts:** a group of attributes used for filtering and flagging.
   - **id-atts:** attributes used to identify or reference a topic or element.
   - **univ-atts:** a group of universal attributes, including the select-att and id-att groups, that are available to a large number of DITA elements.

   There are several other attribute groups defined in the DITA DTD. Each is described in the *DITA Reference Guide*.

We are not going to make any changes to the attributes for recipe. It will be the same as for task.

2. Using the example *recipe.mod* as a guide, complete the remaining element declarations. You should have a declaration for every new element that you created an entity for in the ELEMENT NAME ENTITIES section.

   There are some pre-defined content models that can be used for elements specialized from certain general elements such as section.; section.notitle.cnt for example, defines a standard content model for sections that have no title. section.cnt is available for sections with a title and body.cnt for body sections. All three of these are defined in *topic.mod*. Other predefined element content entities can be found in *commonElements.mod*.

   For most of the new elements in *recipe.mod*, the attributes are the same as in the ancestor element. One exception is the spectitle attribute on ingredientlist, description, instructions,serving and nutrition. spectitle is used to provide a fixed label or heading for a specialized element. This allows you to create sections with fixed headings.

   Note: The ingredientlist spectitle is not processed by default in the *Open Toolkit*. You have to create a custom XSL transform for the topic/ul element that calls the sect-heading template if spectitle is present. There is an example of this in the *custom.xsl* file in *dita-tutorial.zip*. Uncomment the 'topic.ul' template to use it.

3. The last section of the .mod file defines the content of the class attribute. This is a very important step. the class attribute ensures that XSL transforms can be 'specialization aware'. That means that topics of your new specialized type can still be processed by stylesheets that only know about the base topic types. For example, recipe topics can be processed as tasks if there are no specialized stylesheets.

   The class attribute provides the link between the new element and its base types.

   ```
   <!ATTLIST recipe %global-atts; class CDATA "- topic/topic task/task recipe/recipe " >
   ```

   The class attribute value always starts with a '-' character in an information type specialization module ('+' in a domain specialization module). This followed by one or more space-delimited values followed by a final space. Each value uses the pattern *module_name/element_name*. The most general base element is first in the list.

4. Using the completed recipe.mod as a guide, add an !ATTLIST declaration for each new element that adds global-atts and the class attribute with the correct value.
5. Save the file as *recipe.mod* in the *DITA Open Toolkit dtd* directory.

How To Create the Recipe Shell DTD

The final step in creating an information type specialization is to create the 'shell' .dtd file to integrate your module into the DITA DTD.

1. Open the *recipe_template.dtd* file in your XML editor. You may also want to open the final *recipe.dtd* file for reference also.

   As in the .mod file, the first section is a set of comments containing information about the specialization DTD. Edit these comments to reflect the name and purpose of your specialization. Make sure that you edit the public identifier for the DOCTYPE declaration.

   The first section of the .dtd, 'DOMAIN ENTITY DECLARATIONS', declares an entity for each specialized domain that you want to be available to recipe topics. This entity is then used to include the domain entity file.

2. For recipe topics, we will include the highlighting domain so that b, i, u elements etc are available.

   Each specialized domain is referred to by a short identifier followed by -d. This gives us hi-d for the highlighting domain. -dec is appended to this to show that the entity refers to the entity declaration file. The full code for this section is as follows:

   ```
   <!ENTITY % hi-d-dec     PUBLIC
   "-//OASIS//ENTITIES DITA Highlight Domain//EN"
   ```

```
"highlightDomain.ent"                                  >
%hi-d-dec;
```

!ENTITY declares the entity and the last line includes the file referenced by hi-d-dec into the .dtd. You can include additional domains in the same way.

In the sample tutorial file *recipe.dtd*, several other domains are included. We will create and include these domains in the next few sections of this tutorial.

The next section, 'DOMAIN EXTENSIONS' requires you to declare an entity for each base element that is extended by the domain(s) that you just included. This can be a little complicated.

To find out which base elements have been extended by a domain, examine the domain .mod file.

3. Open *highlightDomain.mod*.

4. Scroll down to the bottom of the file until you see the class attribute declarations. These are similar to the class attribute declarations in *recipe.mod*.

Tip: The + character at the beginning of the class attribute value indicates that this is a domain specialization not an information type.

The class values tell you which elements have been extended. In this case, all the elements in the domain were extended from *topic/ph*, so that is the only extension that we need to declare in the .dtd. The declaration needs to define the element name and each domain that the element is extended in. There is only one domain in our dtd so you only need to define one.

```
<!ENTITY % ph          "ph     | %hi-d-ph;"                    >
```

task.dtd gives an example of more complex domain extension declarations for types that include multiple domains.

The ' NESTING OVERRIDE' is where we can override the default topic nesting behaviour using the entity that we declared in the .mod file. We are only going to allow other recipe topics to be nested within a parent recipe topic.

Tip: Define the entity as 'no-topic-nesting' to disable topic nesting completely for this type:

```
<!ENTITY % recipe-info-types
                       "no-topic-nesting"                              >
```

5. Enter the following code:

```
<!ENTITY % recipe-info-types
                       "recipe"                              >
```

6. In the 'DOMAINS ATTRIBUTE OVERRIDE' section, we re-define the included-domains entity so that it references those domains that we have already declared as included in this type. For the recipe type, this is just the highlighting domain.

```
<!ENTITY included-domains "&hi-d-att;"                   >
```

7. The next section of the .dtd is 'TOPIC ELEMENT INTEGRATION'. Here we include the base topic types and the specialization module so that we can access the elements that they contain.

```
<!--                    Embed topic to get generic elements        -->
<!ENTITY % topic-type   PUBLIC
"-//OASIS//ELEMENTS DITA Topic//EN"
"topic.mod"                                                >
%topic-type;

<!--                    Embed reference to get specific elements  -->
<!ENTITY % task-typemod
                        PUBLIC
"-//OASIS//ELEMENTS DITA Task//EN"
"task.mod"                                            >
 %task-typemod;

<!--                  Embed recipe module to get specific elements  -->
<!ENTITY % recipe-typemod
```

```
                    PUBLIC
"-//LD//ELEMENTS DITA Recipe//EN"
"recipe.mod"                                  >
 %recipe-typemod;
```

Each entity defines the public identifier for the DOCTYPE declaration and the path to the .mod file (relative to the location of the .dtd).

8. Finally, in 'DOMAIN ELEMENT INTEGRATION', we include the domain specialization modules for the domains that we want to include.

```
<!ENTITY % hi-d-def     PUBLIC
"-//OASIS//ELEMENTS DITA Highlight Domain//EN"
"highlightDomain.mod"                                  >
%hi-d-def;
```

First we declare an entity with the module's public identifier and path to the file. Then we reference that entity to include it. Notice that for this entity name we use the domain identifier (xx-d) with -def appended to it to show that this is the definition file.

9. Save the file as r*ecipe.dtd* in the *DITA Open Toolkit dtd* directory.
10. You can test the specialization by creating a new topic that references the recipe DTD.

Specializing Domains and Attributes

In the next two sections of the tutorial, we are going to implement two different specializations with some specific structural and vocabulary requirements in mind:

- First, I want to be able to display temperatures in farenheit or celsius
- Secondly, I want to be able to output versions suitable for low fat diets and a regular version.

Domain specialization will allow us to create new elements that can be used to tag temperatures and food items. The new elements in our cooking will be:

- fooditem
- temperature

This domain applies most obviously to the recipe topic type. But it could also be combined with other types if needed. For example, you could perhaps create a recipe_info type that combined the cooking domain with the concept information type.

We can also specialize the props attribute to define the new metadata we need in order to filter the output. Our new attributes will be:

- measuresystem: possible values imperial, metric
- fatcontent: possible values low, high

Any attribute that is created as a specialization of the props attribute is a *selection* attribute which can be used to filter of flag content. You could also use the otherprops attribute to supply this information and avoid specialized attributes. However, the specialization allows us to use attribute names that make more sense both to yourself and other writers who may need to work with the content.

How To Create the Cooking Domain Entity File

Each domain specialization includes a .ent file in which the element and domain entities are declared; and a .mod file in which the specialized elements are declared.

The tutorial files contain template files for you to use with this section and also completed files for reference.

1. Open the *cooking_template.ent* template. You can also open the completed *cooking.ent* file for reference if you wish.

The first section of the file is the descriptive comments section. It is worth making sure this information is accurate and up to date for all specialization files that you create. Make sure that you update the public identifier as it will be used later.

2. In the ELEMENT EXTENSION ENTITY DECLARATIONS section, we declare an entity for each base element that is extended by our domain.

   Both the fooditem and temperature elements are specialized types of phrase, so both are extended from the ph element.

   ```
   <!ENTITY % ckg-d-ph      "fooditem|temperature"                    >
   ```

   Note that we are using 'ckg' as the domain identifier. Each domain should have an identifier, usually consisting of 2-3 letters, which can be used to uniquely identify the domain.

3. In the DOMAIN ENTITY DECLARATION section, we must declare an entity for the domain which defines all of its dependencies. This domain only extends the single element ph, which is part of the topic information type. Therefore it has a single dependency on topic.

   ```
   <!ENTITY ckg-d-att "(topic ckg-d)">
   ```

4. Save the file as *cooking.ent* in the *DITA Open Toolkit* dtd directory.

How To Create the Cooking Domain Definition Module

1. Open the *cooking_template.mod* template. You can also open the completed *cooking.mod* file for reference if you wish.
   The first section of the file is the descriptive comments section. Make sure that this information is accurate and up to date for all specialization files that you create. Make sure that you update the public identifier, as it will be used later.
2. In the 'ELEMENT NAME ENTITIES' section, we will declare an entity for each new element:

   ```
   <!ENTITY % fooditem   "fooditem"                              >
   <!ENTITY % temperature   "temperature"                         >
   ```

3. Next we define the fooditem and temperature elements by copying the definition for ph from the *commonElements.mod* file. You can use the base element definition as a starting point for your specialization, just as when defining new elements in an information type.

   Remember: The attributes for the specialized element must be the same or a subset of the attributes for the base element. In this case the attributes are the same as for the base element, ph.

   ```
   <!--                    LONG NAME: Food Item          -->
   <!ELEMENT fooditem     (%words.cnt;)*       >
   <!ATTLIST fooditem
             keyref     CDATA                          #IMPLIED
             %univ-atts;
             outputclass
                       CDATA                        #IMPLIED    >

   <!--                    LONG NAME: Temperature          -->
   <!ELEMENT temperature     (%words.cnt;)*       >
   <!ATTLIST temperature
             keyref     CDATA                          #IMPLIED
             %univ-atts;
             outputclass
                       CDATA                        #IMPLIED    >
   ```

   Tip: words.cnt entity defines a content model for elements that contain only text. It is defined in *commonElements*.mod

4. In the 'SPECIALIZATION ATTRIBUTE DECLARATIONS' section, we add the class attribute value. This attribute allows the specialized elements to be transformed by templates for a more generalized base element if one doesnt exist specifically for the specialized element.

   ```
   <!ATTLIST fooditem   %global-atts;  class CDATA "+ topic/ph ckg-d/fooditem "    >
   <!ATTLIST temperature   %global-atts;  class CDATA "+ topic/ph ckg-d/temperature "    >
   ```

5. Save the file as *cooking.mod* in the *DITA Open Toolkit* dtd directory.

How To Create the Extended Attribute Domain Entity File

You need to create an .ent entity file for each specialized attribute.

1. Open the *measuresystemAtt_template.ent* file. You can also open the completed *measuresysetmmAtt.ent* file for reference.
2. The file begins with the usual comments. Edit these as appropriate.
3. In the 'ELEMENT NAME ENTITIES' section, we declare an entity to define the new attribute.

   ```
   <!ENTITY % measuresystemAtt-d-attribute "measuresystem CDATA #IMPLIED"                              >
   ```

   Attributes specialized from props will always have CDATA #IMPLIED as their allowed content.
4. in the 'DOMAIN ENTITY DECLARATION' section, we declare an entity that defines the attribute domain and allows the new attribute to be picked up by DITA processing. The leadeing 'a' indicates that this is an attribute.

   ```
   <!ENTITY measuresystemAtt-d-att "a(props measuresystem)"                              >
   ```

5. Save the file as *measuresystemAtt.ent* in the *DITA Open Toolkit* dtd directory.
6. Open *fatcontentAtt_template.ent* and create the entity file for this attribute in the same way as you did for measuresystem.

Integrate Domains and Attributes into the Shell DTD

1. Open *recipe.dtd* from the *DITA Open Toolkit* **dtd** directory.
2. First, you must add references to all entity declaration files in the 'DOMAIN ENTITY DECLARATIONS' section.

   ```
   <!ENTITY % ckg-d-dec PUBLIC  "-//LD//ENTITIES DITA Cooking Domain//EN"
     "cooking.ent">
   %ckg-d-dec;

   <!ENTITY % measuresystemAtt-d-dec PUBLIC
   "-//LD//ENTITIES DITA measuresystem Attribute Extension Domain//EN"
     "measureSystemAtt.ent">
   %measuresystemAtt-d-dec;

   <!ENTITY % fatcontentAtt-d-dec PUBLIC
   "-//LD//ENTITIES DITA fatcontent Attribute Extension Domain//EN"
     "fatcontentAtt.ent">
   %fatcontentAtt-d-dec;
   ```

   These references follow the same pattern as the entity for the Highlight domain.
3. Update the 'DOMAIN EXTENSIONS' section so that it references all extended base elements in the new specialized domain.

   ph is the only element extended by the new domain and it already has an entry for the Highlight domain. All you need to do is add the cooking domain entity to the list:

   ```
   <!ENTITY % ph          "ph      | %hi-d-ph;| %ckg-d-ph;" >
   ```

4. Next, you must define the props-attribute-extensions entity to reference the two specialized attributes:
   ```
   <!ENTITY % props-attribute-extensions  "%measuresystemAtt-d-attribute;
                                  %fatcontentAtt-d-attribute;">
   ```
5. In the 'DOMAINS ATTRIBUTE OVERRIDE' section, add references to domain entities for the domain and attributes:
   ```
   <!ENTITY included-domains "&hi-d-att;&ckg-d-att;&measuresystemAtt-d-att;&fatcontentAtt-d-att;">
   ```
6. Finally, add a reference to the new domain's definition module to the 'DOMAIN ELEMENT INTEGRATION' section''
   ```
   <!ENTITY % ckg-d-def PUBLIC "-//LD//ELEMENTS DITA Cooking Domain//EN"
     "cooking.mod">
   %ckg-d-def;
   ```
7. Save the file.
   You are now ready to create and build some recipe topics. Create some recipes and add them to the tutorial ditamap. Try building PDF and HTML outputs.

   You can either create custom templates in *XMLMind* for authoring specialized topic types or you can create them using the default template or in a source editor.

XML Catalogs

An xml catalog creates a mapping between a generic identifier and a location on the local machine. The file *catalog-dita.xml* in the root directory of the *Open Toolkit* provides mappings for the DITA DTD files. This means that you don't have to modify your DOCTYPE reference in your XML files whenever you move the location of your content or create new DITA content in a different location.

DOCTYPE Declarations

This is an example of a DOCTYPE declaration:

```
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "../dtd/task.dtd">
```

The first item, 'task' indicates the root element of the document.

'PUBLIC' indicates that this declaration is using a public identifier.

The third item specifies the public id. This is used to locate the correct mapping in the catalog file.

The final item is a system identifier. This provides the location of the file as a path or uri and is used if the public ID cannot be located in the catalog.

Catalog Entries

In the Open Toolkit catalog, all the entries are contained in a group element:

```
<group xml:base="dtd/">
```

xml:base provides a base location for catalog entry mappings that are children of the group element. This is relative to the catalog file location.

This is the corresponding catalog entry for the DTD referenced in the DOCTYPE declaration above:

```
<public publicId="-//OASIS//DTD DITA Task//EN" uri="task.dtd"></public>
```

The uri is relative to the location specified in xml:base for the entry's parent group element.

catalog-dita-template.xml

This file is copied to the catalog each time you build output using the *Open Toolkit*.

Specialization Files

You should add entries to the catalog template (*dita-catalog-template.xml*) for each new file used by your specialization. The *dita-catalog.xml* file is updated using the contents of *dita-catalog-template.xml* each time that you build. For example, you entries for the recipe information type files would look like this:

```
<public publicId="-//LD//DTD DITA Recipe//EN" uri="recipe.dtd">
 </public>

 <public publicId="-//LD//ELEMENTS DITA Recipe//EN" uri="recipe.mod">
 </public>
```

catalog-ant.xml

You should also add catalog entries for each specilaization file to the catalog-ant.xml file, which also resides in the *DITA Open Toolkit* directory. These entries have a slightly different syntax. For example:

```
 <dtd publicId="-//LD//DTD DITA Recipe//EN"
      location="${dita.dtd.dir}${file.separator}recipe.dtd" />
  <dtd publicId="-//LD//ELEMENTS DITA Recipe//EN"
      location="${dita.dtd.dir}${file.separator}recipe.mod" />
```

Configuring XMLMind for Specialized Information Types

The DITA Add On for *XMLMind* comes configured with templates and a stylesheet for the base topic types. You can customize *XMLMind* to provide similar support for your own specialization. *XMLMind* comes with detailed documentation on this which can be found in the *docs* directory in the *XMLMind* install directory. The *configure* and *csssupport* documents are particularly useful.

Use the existing files in the DITA add on as a starting point. Create a recipe template based on the existing task template. These can be found in the template directory of the DITA add on directory. You can then add a line to reference your specialization and template to the *dita.xxe* configuration file and the *dita-common.incl* file. Make sure that you copy your specialization dtd and mod files to the DITA add on *dtd* directory. Elements from specialized information types and domains, as well as extended attributes will become available from the XMLMind interface.

You can also use *XMLMind's* extended css syntax to define the visual layout used when editing specialized topic types.

Using DITAVAL Files to Filter and Flag

Filtering and flagging, sometimes referred to as *Conditional Processing* allow you to exclude or flag certain content based upon the values of certain attributes.

Selection attributes are used to filter and flag content. These attributes are:

- Audience
- Platform
- Product
- Rev
- Status
- Otherprops
- Attributes specialized from props

Otherprops can be used for values that don't fit well in any of the other attributes. As we saw in the Specialization section, props can be extended to provide selection attributes specific to your requirements. Both measuresystem and fatcontent can be used to filter and flag content in a recipe.

Note: You can use filtering on phrase-level elements and it appears to work fine. However, flagging only seems to work on block level elements.

Filtering Logic

These attributes can each take multiple values, separated by a space.

For example, audience="users expert"

All the values supplied for an attribute must be set to exclude for the element to be excluded. However, only one attribute needs to evaluate to exclude the element. Consider this element:

```
<fooditem measuresystem="imperial metric" fatcontent="high">1 small bar of chocolate</fooditem>
```

If you excluded imperial measurement and high fat content, this element would be excluded because all of the values assigned to fatcontent were excluded, even though only one of measuresystem's values was excluded.

If you only excluded imperial measurement, the element would not be excluded because neither attribute evaluates to exclude.

If you excluded imperial and metric measurement, the element would be excluded because measuresystem would evaluate to 'exclude'.

Flagging Logic

Whenever a flagged value appears in an attribute, the element will be flagged. If there are multiple flagged values, the first one will be flagged.

You can assign different text or a different image to be used to flag each attribute/value pair. For example, you can specify different images to flag high fat content and low fat content.

Note: Flagging is not currently implemented for pdf2 output.

Creating Filtering and Flagging Rules

Filtering and flagging rules are specified in a separate XML file. This file is then referenced from your ANT build script by setting the dita.input.valfile property. You will sometimes see these files referred to as filter files or val/ditaval files.

How To Create a Ditaval File

Before we do that, you will need to create some sample content to work with. Try creating a few recipe topics and provide measuresystem and fatcontent attributes. Provide cooking temperatures by creating two temperature elements, one with measuresystem set to imperial and the other with it set to metric.

Remember: You can only flag block level elements. Since fooditem is a phrase element, you will need to add the fatcontent attribute to the ingredient element for flagging.

Set the audience attribute to novice in the 'How To Boil Potatoes' and 'How To Check Meat Temperature' topics.

1. Create a new XML file in your XML source editor.
2. Add a root element val with no attributes.
3. Add a prop element with the following attributes:

att	The name of the attribute to filter or flag: *measuresystem*
val	The value the attribute should have for it to be filtered/flagged: *metric*

action	Whether to filter or flag: *exclude*

The complete code:

```
<?xml version="1.0" encoding="UTF-8"?>
<val>
  <prop action="exclude" att="measuresystem" val="metric" />
</val>
```

This entry will exclude the content of all elements that have the measuresystem attribute set to *metric* from the output.

4. Create a similar prop element to exclude the 'novice' topics.

```
<prop action="exclude" att="audience" val="novice" />
```

5. Next we will create the flagging entry. Add a second prop element and add the following attributes:

att	'fatcontent'
val	'low'
action	'flag'
img	The name of the image to use as the flag marker: 'smile.gif'
alt	Alternative text for the image. This is displayed if the image cannot be. It is used as the mouseover hint for some output types: 'Low Fat!'

The complete code:

```
<?xml version="1.0" encoding="UTF-8"?>
<val>
  <prop action="exclude" att="measuresystem" val="metric" />
  <prop action="flag" att="fatcontent" val="low" img="../smile.gif" alt="Low Fat!"/>
</val>
```

6. Create a *filters* directory in the *tutorial* content directory and save the ditaval file as *tutorial.ditaval*.
7. Create the *smile.gif* image or use the one in the tutorial zip file and copy it into the same directory as the ditaval file.
8. Finally, we need to instruct the build to use the ditaval file. Add the following property to your build file for XHTML (it should work in the same way for other output types):

```
<property
name="dita.input.valfile"
value="${dita.dir}${file.separator}tutorial${file.separator}filters${file.separator}tutorial.ditaval"
></property>
```

You can also add an instruction to your build file to automate the copying of your flag images since the *Open Toolkit* does not do this for you. The images need to be copied to the output directory. So far as I can tell, the src attribute in the HTML img tag is always set to point to the file in the same directory. E.g `<img src="smile.gif">`.

9. To copy all .gif files from the filters directory to the output directory, add the following instruction to your build file. It should be copied into the target but outside the ant call.

```
<copy todir="tutorial${file.separator}out${file.separator}html">
    <fileset dir="${basedir}${file.separator}tutorial${file.separator}filters">
      <include name="**/*.gif"></include>
    </fileset>
  </copy>
```

It is fairly self-explanatory and should be easy enough to modify to your own needs. The fileset element is used to define a set of files to copy and the include element defines the filter to use to select files from the fileset directory.

10. Build the XHTML output and check that you get the expected results.

11. Comment out the exclude for novice audiences in the ditaval file and rebuild. Note that the novice topics are included once again.

Content Inclusion

DITA allows you to easily re-use content. You have already seen how topics can be re-used in maps files but DITA also allows you to reuse content at the element level.

IDs

To re-use an element, you must give it an ID. The ID is required on topic level elements but is optional on others. You only need to assign an ID to elements that you want to reference elsewhere.

Topic IDs must be unique across a document. Element IDs must only be unique within a topic because they can be referred to in the form *topic id/element id*.

A full reference to an element consists of a URI of the file containing the topic, topic ID and element ID:

```
concept_thanksgiving.xml#thanksgiving_intro/history
```

You can use relative URIs providing that they can be resolved or a full absolute URI, which may point to a remote web site.

Maintaining Document Validity

The conref processing in the *DITA Open Toolkit* works in such a way that you cannot produce documents that are invalid against the parent topic type's DTD.

For example, you cannot reference a taskbody element from a conbody source element in a concept topic because taskbody is invalid in a concept topic.

To acheive this, content referencing is restricted to elements of the same type. Although the DITA Architectural Specification suggests that you should be able to reference elements that are specialized versions of the source element, I havent found any circumstance in which the source and target elements can be of different types when using the *Open Toolkit*.

How To Use Conref

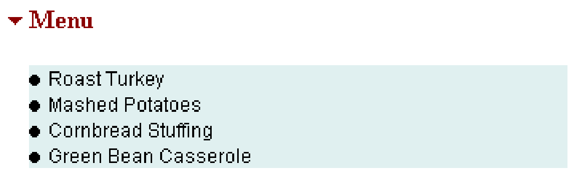
1. Create a new concept topic with the title "Thanksgiving Menu".
2. Create an introductory paragraph similar to the following:

   The Thanksgiving Menu that we have chosen is a very traditional one. You may want to add a traditional dessert, such as Pumpkin Pie.
3. Next, add an un-ordered list to display the menu. It should look similar to this:
   - Roast Turkey
   - Mashed Potatoes
   - Cornbread Stuffing
   - Green Bean Casserole
4. Select the ul element.
5. Set the id attribute to *menu_list*.
6. Set the id attribute of the task element to *thanksgiving_menu*.
7. Save the topic as *concept_thanksgiving_menu*.xml in the *tutorial* directory.
8. Open the 'Thanksgiving Dinner' introductory topic (*concept_making_dinner.xml*).
   We are going to insert the menu into this topic using a conref attribute.

9. Add a new section element to the topic and give it the title 'Menu'.
10. Add a ul element to the section.
11. Select the ul element and enter this value for the conref attribute:

    ```
    concept_thanksgiving_menu.xml#thankgiving_menu/menu_list
    ```

12. In *XMLMind*, save the file, close it then re-open it to see the included content.



- Roast Turkey
- Mashed Potatoes
- Cornbread Stuffing
- Green Bean Casserole

Figure 40: Included content using conref in XMLMind

Troubleshooting

Enable Logging	Add '-logger org.dita.dost.log.DITAOTBuildLogger' to the end of your ant command when you build output to create a logfile in the output directory. This makes it easier to review errors that occurred during the build.
Multiple File Search and Replace	A tool that allows you to search multiple files in one operation is very useful in many troubleshooting scenarios. Microsoft Visual Studio Web Developer Express allows you to do this and is availble as a free download from Microsoft. There are almost certainly a number of other tools that provide the same feature.
Problems with Output Directory Structure	If you find that the Open Tookit is placing your output into one or more sub-directories rather than directly into the output directory that you specified, check you DITA map file for topic references that use '../' to move back up the directory structure. Topic files should be in subdirectories (or the same directory) as the map file to avoid this.
'[DOTA002F][FATAL] Invalid Input. Please provide the correct input.	This error message usually indicates that the path to the ditamap in your build file is incorrect. Check your *args.input* property setting.

XMLMind Tips and Tricks

XMLMind Editing Tips

Use the DITA toolbar

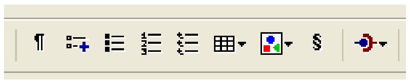


Figure 41: XMLMind DITA Toolbar

The DITA toolbar is displayed whenever you are editing a DITA topic type. It provides shortcuts to Highlight domain elements such as italics (i) and bold (b). It also has buttons for inserting lists, tables, images and sections. You can insert a conref using a two-step function: select a source element with an id attribute and select the option to copy the reference. You then select the target element and choose the second option to copy the correct reference into the conref attribute.

Use the Node Path Selection Bar

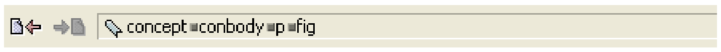


Figure 42: Node Selection Bar

Right-clicking on a node opens a menu that allows you to delete it or to add the same tag before or after it. You can also CTRL+Click on a node on the bar to add the same element as a sibling. This is useful when creating lists and tasks. Pressing SHIFT+Click will insert the same element before the current one. Remember that deleting a tag will delete all its children. A red box is displayed around any visible content contained by the selected tag, including the content of all its children.

Use Selection Functions

You can also use the arrow buttons on the toolbar or the **Select** menu to move through tags. The left and right arrows move through siblings whilst the up and down arrows move through parents and children. Additional options on the **Select** menu allow you to extend the current selection to the preceding or following sibling.

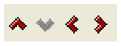


Figure 43: Selection buttons on the toolbar

Finally you can use the **Find Element** feature either from the toolbar or from the **Select** menu to find elements with a specified name.

Use Shortcut Keys

XMLMind has shortcut keys for most functions. When inserting, wrapping, converting or replacing element, you can type the first few letters of the element name to select it, then press Enter to perform the function. You can use this in conjunction with the keyboard shortcuts below to quickly add elements.

CTRL + I	Insert an element as a child of the current one.
CTRL+J	Insert an element after the current one.
CTRL+H	Insert an element before the current one

Select **Mouse and Key Bindings** from the *XMLMind* **Help** menu for a full list. You may never need to use the mouse again!

Glossary

Term	**Defintion**
Block Element	
DITA	
Phrase Element	

Index