

Learning DITA

Table of Contents

Course I. Introduction to DITA.....	8
Lesson 1: What is DITA?.....	8
Markup languages.....	8
DITA is structured authoring.....	9
What is a DITA topic?.....	10
Lesson 2: DITA topics.....	10
The generic topic type.....	11
Concept topics.....	11
Task topics.....	12
Reference topics.....	13
Glossary entry topics.....	13
Specialization.....	14
Lesson 3: Metadata.....	14
What is metadata?.....	15
How do you use metadata?.....	16
Lesson 4: Creating DITA content.....	18
Creating a topic in a text editor.....	18
Creating a topic in oXygen.....	18
Notes, cautions, and warnings.....	20
Bulleted and numbered lists.....	21
Block versus inline elements.....	22
Lesson 5: Tables.....	23
The simpletable elements.....	24
The table element.....	25
Best practices for tables.....	27
Lesson 6: Creating relationships among topics.....	28
Map files.....	28
Transforming map files into outputs.....	29
Cross-references.....	30
Related links.....	31
Relationship tables.....	31
Reusing content with conrefs.....	32
Course II. The DITA Concept Topic Type.....	34
Lesson 1: Creating a concept topic.....	34
Housekeeping and sample files.....	34
Creating a new concept topic.....	35

Adding basic elements to a concept topic.....	37
Lesson 2: Images and tables.....	40
Adding images to a concept topic.....	41
Adding simple tables to a concept topic.....	42
Adding tables to a concept topic.....	45
Lesson 3: More Elements.....	52
Adding inline styling.....	53
Adding other elements.....	56
Lesson 4: Advanced elements.....	60
Adding advanced elements.....	61
Adding more advanced elements.....	64
Lesson 5: XML overview and best practices.....	67
Separate content from formatting.....	67
Be consistent.....	68
Create semantically rich content.....	69
Course III. The DITA Task Topic Type.....	70
Lesson 1: Creating a task topic.....	70
Housekeeping and sample files.....	70
Strict or general tasks.....	71
Creating a new task topic.....	72
Adding introductory elements to a task topic.....	73
Lesson 2: Creating the steps.....	74
First steps.....	75
Giving the reader choices.....	77
A table of choices.....	79
Getting more detailed.....	81
Showing examples and results.....	83
Interrupting the flow.....	85
Lesson 3: Finishing up the task.....	87
Task results.....	88
Making an example.....	88
What's next?.....	89
Lesson 4: Best practices for tasks.....	91
Plan your tasks.....	91
Provide appropriate context.....	92
Use a reasonable number of steps.....	92
Use substeps appropriately.....	92
Keep an eye toward reuse.....	93

Course IV. The DITA Reference and Glossary Topic Types.....	94
Lesson 1: Creating a reference topic.....	94
Housekeeping and sample files.....	94
About reference topics.....	95
Creating a new reference topic.....	96
Describing the syntax.....	98
Adding a properties table to a reference.....	99
Lesson 2: Best practices for reference topics.....	103
Create templates.....	104
Use relationship tables for links.....	104
Use DITA domain elements.....	104
Lesson 3: Creating a glossary entry.....	105
About glossary entry and glossary group topics.....	106
Creating a new glossary entry.....	107
Creating a new glossary group topic.....	108
Organizing glossary entries in a map.....	110
Lesson 4: Best practices for glossary entries and groups.....	113
Naming files and ids.....	113
Building a useful glossary.....	113
Adding a glossary to a bookmap.....	114
Manage glossary creation.....	114
Course V. Using DITA Maps and Bookmaps.....	115
Lesson 1: Creating a map.....	115
Housekeeping and sample files.....	115
Creating a new DITA map.....	116
Adding topic references to a map.....	117
Adding map references to a map.....	120
Lesson 2: Creating a bookmap.....	122
Overview: maps vs. bookmaps.....	122
Creating a new bookmap.....	123
Adding chapters to a bookmap.....	124
Lesson 3: Advanced DITA map and bookmap concepts.....	127
Adding metadata to a map.....	128
Adding metadata to a bookmap.....	130
Frontmatter, backmatter, and booklists.....	133
Adding a navtitle to a bookmap.....	135
Adding a topichead to a bookmap.....	137

Lesson 4: Relationship tables.....	138
About relationship tables.....	139
Creating a relationship table.....	140
Adding rows to the relationship table.....	142
Advanced relationship tables.....	143
Best practices for relationship tables.....	145
Course VI. Introduction to Reuse in DITA.....	147
Lesson 1: Introduction to reuse.....	147
Housekeeping and sample files.....	147
Reasons for reuse.....	148
Analyzing content for reuse.....	148
Types of reuse.....	149
Lesson 2: Creating reusable topics.....	150
Writing for reuse.....	150
Reuse and translation.....	151
Reusing paragraphs.....	153
Reusing steps.....	154
Reusing list items.....	154
Storing your reusable content.....	155
Lesson 3: Reusing topics and maps.....	156
Reusable topics and maps.....	156
Reusing a topic in the same map.....	157
Reusing maps.....	158
Merging maps.....	159
Lesson 4: Using content references.....	160
What is a content reference?.....	160
Conref basics.....	161
The conref attribute.....	162
Element requirements for conrefs.....	163
Reusing across topics.....	164
Attributes of content references.....	164
Best practices for using conrefs.....	165
Course VII. Advanced Reuse in DITA.....	167
Lesson 1: Using conditions.....	167
Housekeeping and sample files.....	167
Review of basic reuse.....	168
Marking conditional elements.....	168
What can you filter?.....	170
The ditaval file.....	171

Flagging content.....	173
Best practices for conditions.....	175
Lesson 2: Using keys.....	176
What is a key?.....	176
Using keys for paths.....	177
Using keys for text.....	178
Advanced key uses.....	179
Best practices for keys.....	180
Lesson 3: Advanced conrefs.....	181
Review of content reference basics.....	181
Conkeyrefs.....	182
Conrefend.....	184
Conref push (conaction).....	186
Best practices for advanced conrefs.....	190
Course VIII. Publishing Output from DITA Sources.....	192
Lesson 1: Choosing a DITA publishing environment.....	192
Housekeeping and sample files.....	192
Determining output requirements.....	193
DITA publishing environment considerations.....	194
Publishing with a standalone DITA OT.....	195
Publishing with the oXygen XML Editor.....	196
Publishing with a component content management system (CCMS).....	196
Lesson 2: Setting up your DITA publishing environment.....	198
Installing the DITA OT.....	199
Installing Java.....	200
Source and output folders.....	201
Testing the DITA OT installation.....	203
Lesson 3: Generating output.....	204
DITA OT commands and parameters.....	204
Generating PDF output.....	205
Generating HTML output.....	206
Troubleshooting common issues with generating output.....	209
Lesson 4: Custom plugins.....	211
Why customize plugins?.....	211
Installing a custom plugin.....	212
Updating a custom plugin.....	213
Best practices for custom plugins.....	215

Course IX. LearningDITA Live 2018 Recordings.....	217
Beginner sessions.....	217
DITA overview.....	217
You got DITA in my minimalism! You got minimalism in my DITA!.....	217
Teaching (and learning) with DITA at the college level.....	217
Getting started with DITA and Adobe FrameMaker 2017 in less than one hour.....	217
Intermediate sessions.....	218
The business value of DITA.....	218
Improving the quality of your DITA documentation.....	218
Localization strategy for DITA.....	218
Managing DITA projects in the real world.....	219
Faster content, better healthcare: improving cancer diagnostics with electronic delivery.....	219
Starting small with structured content management.....	220
Delivering a consistent content experience. Everywhere. Everyday.....	220
When conversion makes sense.....	220
Advanced sessions.....	220
DITA for agile documentation.....	220
DITA specialization overview.....	221
Developing learning content in DITA.....	221
InDesign and DITA.....	221

Course I. Introduction to DITA

Lesson 1: What is DITA?

Objectives

- Present the background and origins of DITA.
- Identify DITA markup.
- Describe what makes a topic reusable.
- Show a minimal topic.

DITA stands for Darwin Information Typing Architecture. It is an open standard, originally created by IBM. IBM donated DITA to [OASIS](#) (the Organization for the Advancement of Structured Information Standards) in 2005.

DITA is an XML-based, tool-independent way to create, organize, and manage content. DITA is built on:

- Topic-based authoring
- Separating content from formatting
- Minimalism
- Structured authoring concepts

The fact that DITA is an open, XML-based standard means that there are a diverse set of tools available to author, edit, format, and store DITA files. As your business needs change, you can use the tool that's most appropriate to your needs.

Markup languages

A markup language is a way of tagging content in a plain text file (a file you might edit with Windows Notepad). The markup language you might be most familiar with is HTML, which is the fundamental markup language for the world wide web.

DITA is built on the XML markup language. XML looks a lot like HTML. They both use angle brackets (< and >) to identify the markup tags (for example <title>). In both languages a forward slash identifies a close tag (</title>)

```
<title>This is a title</title>
```

In HTML and in XML, the tags can also have attributes (in the form `attribute="value"`) that provide further information about the tag:


```
<note type="warning">Keep your arms and legs inside the vehicle at all times.</note>
```

There are two main differences between HTML and XML:

- HTML can be quite forgiving when you forget to close tags or put quotation marks around attribute values; XML is strict in requiring them.
- HTML uses a predefined set of tags (<body>, <p>, , and so on). In XML the tags are defined in a separate file and can be changed and added to by an information architect.

DITA uses this tag naming feature in XML to define its own set of tag names or “elements”. The DITA elements allow you to mark up content using names for things that make sense, such as <note> for notes, <section> for sections, <image> for images, and so on. However, because many HTML tag names make sense, their names are also used for DITA elements, such as <p> for paragraph, and and for unordered and ordered lists.

One other difference in naming: in HTML, the outer-most or “root” tag is <html>. In DITA, the name of the root tag depends on the type of topic you’re creating, such as <concept>, <task>, or <reference> (among others).

Note: It’s useful to know how the markup looks in XML and DITA. However much of the time you’ll be creating DITA content using smart authoring tools that handle the individual tags and attributes for you.

Video: [Overview of HTML markup versus DITA markup](#)

DITA is structured authoring

Structured authoring is a publishing workflow that lets you define and enforce consistent organization of information in documents.

The elements defined in DITA have a very specific hierarchy and relationships. That is, each element in DITA has a specific set of elements that might contain it, and a specific set of elements that can be contained by it. DITA also specifies which elements can (or must) come before or after an element and how many of a particular element are allowed.

For example, a DITA simpletable can contain an optional simpletable head, which must be followed by one or more simpletable rows. The rules prevent a writer from creating a simpletable with a table head in the middle of the rows, or from creating a table that contains no rows at all.

The rules that define this structure are defined by the DITA markup language. The same language can be used by computer programs to make sure that DITA documents are structured correctly. If a particular set of documents requires different rules or different element names, an information architect can define these names and rules.

Although DITA is a widely-used form of structured authoring, there are other structured authoring standards, including DocBook, S1000D, NLM, and more.

What is a DITA topic?

A DITA topic is the basic unit of authoring and reuse.

In topic-based authoring, you create a number of individual topics, each of which each addresses a single idea or answers a single question. These topics can then be used and reused in any order, in a number of different documents. In DITA the topics are organized in *maps*, which are much like a table of contents; the map allows you to specify the order and the hierarchy of the topics.

To make your topics reusable:

- A topic should address a single idea or answer a single question.
- A topic should contain enough information to stand on its own.
- A topic should not assume any context. You should not make assumptions about what comes before or after the topic.
- A single file should contain a single topic.

All DITA topics must have at least a title element and an id attribute for the root topic. That is, the following is a valid DITA topic:

```
<topic id="sample">
  <title>Topic title goes here</title>
</topic>
```

However, a *useful* topic will have additional content.

Lesson 2: DITA topics

Objectives

- Identify the common elements found in all topics.
- Cite the standard topic types defined in DITA.
- Identify the elements typically found in each topic type.
- Define specialization and constraints.

A DITA topic is a building block of content.

One of the fundamental principles of DITA is that different types of content require different containers (topic types). That way the elements in the topic can be specific to the information being described.

All DITA topic types are related to a common or generic topic type. The generic topic type is then “specialized” into a number of specific topic types, each of which holds a specific type of information. The “standard” DITA topic types include:

- Concept—contains conceptual information; answers the question “Why?”

- Task—contains step-by-step procedural information; answers the question “How?”
- Reference—contains reference information; answers the question “What?”
- Glossary entry—defines a single term; answers the question “What does this mean?”

This lesson introduces the generic topic type, then describes each of the standard topic types and how you use them.

Additional reading

[DITA Style Guide, Chapter 1](#)

[Technical Writing 101, Chapter 7](#)

[DITA Topics, contributed by Pam Noreault, Tracey Chalmers, and Julie Landman](#)

The generic topic type

All DITA topic types are based on a single generic topic type. The generic topic type is used as the basis to create the specific topic types. Many of the elements and organizations in the generic topic are also used by the specific topic types.

All DITA topics have this general structure:

```
<topic id="sample">
  <title>Topic title goes here</title>
<shortdesc>A short description</shortdesc>
  <body>
    (Most of the elements go here.)
  </body>
</topic>
```

- The <shortdesc> (short description) element provides a 2-3 sentence summary of the topic content.
- The <title> is the only required element in a topic.
- The <body> contains the bulk of the information in the topic. In the specific topic types the body element has a related name, such as <conbody> for concepts and <taskbody> for tasks.

Note: Although you can use the generic topic type for authoring content, it’s much better to use one of the specific topic types, such as concept, task, and reference.

There are other elements in the generic topic type, but these are the common ones.

Concept topics

A concept topic answers the question “Why?” It provides background information about a subject that the reader needs to know.

Concepts usually contain paragraphs of text and lists, but also include notes, tables, and graphics needed to understand the ideas behind a particular subject.

Common elements used in a concept topic include:

- `<conbody>` (the body of the concept topic)
- `<p>` (a paragraph)
- `` (an unordered or bulleted list)
- `` (an ordered or numbered list)
- `` (a list item inside a `` or ``)
- `<fig>` (a figure, including an optional title)
- `<image>` (a graphic inside a figure, or inline in text)
- `<section>` (a subdivision in the topic, with an optional title)

Task topics

Understand the purpose of the DITA task type

- Identify key components of a task
- Create a DITA task

A task topic answers the question “How do I?” It includes step-by-step instructions to complete a procedure. DITA also allows step results, graphics, notes, and one level of substeps.

Technical content is often heavy on tasks.

DITA provides two task types:

- Strict task
- General task

The strict task requires all elements to appear in a specific order, only allows one `<example>` element, and has two very formal elements for steps (`<steps>` or `<steps-unordered>`). The “general task,” also known as the “loose task” allows more flexibility in the order of the elements, allows multiple `<example>` elements, and allows a `<steps-informal>` element for the steps, which can contain much more varied content. Strict tasks are appropriate for items that require step-by-step instructions; general tasks are useful for process overviews.

Common elements used in a strict task topic include:

- `<taskbody>` (the body of the task topic)
- `<steps>` (the sequence of actions)

- <step> (each individual action)
 - <cmd> (the action the user takes; this is a required element in a <step>)
 - <info> (additional information about the step)
 - <stepresult> (what happens after performing an action)
 - <stepxmp> (an example of how to do the step)
- <example> (an example of how to do the entire task)

Video: [Code overview of DITA task topic](#)

Reference topics

A reference topic answers the question “What is it?” A reference topic typically contains descriptive facts, such as the syntax of a command or API function call, a table listing operational characteristics and tolerances of a device, or a table identifying items on a software screen.

Reference topics do not include steps or background information. Reference topics are similar to dictionary entries and provide facts only.

Common elements used in a reference topic include:

- <refbody> (the body of the reference topic)
- <section> (a subdivision in the reference topic, with an optional title)
- <table> (a table)
- <fig> (a figure, including an optional title)
- <properties> (a list of properties)
- <refsyn> (a syntax diagram)

Video: [Code overview of the DITA reference topic](#)

Glossary entry topics

A glossary entry topic answers the question “What does this word or phrase mean?” Glossary topics typically contain one term, along with one or more definitions.

Common elements used in the glossary topic are:

- <glossentry> (the glossary entry topic type)
- <glossterm> (the word or phrase)
- <glossdef> (the definition of the glossary term)

Video: [Code overview of the DITA glossary topic](#)

Specialization

Specialization lets an information architect create elements and attributes that fit your organization better than what is provided in default DITA. The elements created through specialization are based on existing elements in DITA.

Organizations often specialize to support their unique requirements, such as:

- Creating new elements to contain specific information in a specific order
- Creating new attributes to identify specific information
- Creating new names for elements that are more relevant to authors.

For instance, you might require a set of language-specific `<codeblock>` elements, such as `<codeblock-java>` or `<codeblock-php>` to distinguish between Java and PHP code examples. This could also be accomplished by creating a new language attribute for the `<codeblock>` element: `<codeblock language="java">` or `<codeblock language="php">`.

Important: Be careful with specialization. When you specialize, you make tags more specific, but specialization adds to the cost of implementation. You must balance the value gained from specialization against the cost of implementing and maintaining specializations.

In addition to specialization, *constraints* allow information architects to eliminate elements that you do not need or want to use. For example, if your organization is not documenting software code, your information architect could eliminate `<codeblock>` and `<codeph>` (code phrase). Constraining reduces the number of elements presented to the authors, which can make their lives a little easier.

Lesson 3: Metadata

Objectives

- Define metadata.
- Describe uses of metadata.
- Show how to add metadata to topics.

Metadata means “beyond data” or data about data. In XML content, metadata lets you classify and manipulate information. In DITA, you can assign metadata to topics, individual elements, and more.

Additional reading

[DITA Style Guide, Chapter 8, Metadata, conditional processing, and indexing](#)

[Language identification FAQ from opentag.com \(addresses the use of xml:lang for language identification\)](#)

What is metadata?

Metadata provides information about information. For example, word processing applications often have document properties, which tell you who created the file and on what date it was last modified. The author and modification date in the document properties are not part of the text that is displayed. Document properties are metadata about the document itself.

Metadata provides different ways to classify information. For example, you can:

- Label information with a document type, like “white paper” or “proposal”
- Associate information with a specific product number
- Indicate a user level, like “beginner,” “intermediate,” or “expert”
- Indicate the audience for the content, like “manager” or “employee”
- Provide status information for a topic, like “draft,” “review,” or “final”
- Provide an expiration date when information is no longer valid

After you assign metadata to your content, you (or your reader) can use it to:

- Find information efficiently
- Support governance effort (control when information is made public or removed from a web site)
- Personalize information (your reader could request only beginner information)
- Filter information for different delivery variants (such as Windows or Macintosh)
- Reuse information efficiently
- Indicate the language used in content
- Manage project status

In DITA and other XML formats, metadata is encoded inside your topics, either at the topic level or on specific elements. You can manipulate DITA content using those metadata values.

DITA requires one piece of metadata; every topic must have an id attribute. These IDs are often assigned automatically by authoring tools or content management systems. Here is an example of a topic with an id attribute:

```
<topic id="xyz">
  <title>Your title here</title>
  <body>
    ....
  </body>
</topic>
```

Other elements can use the id attribute. For example the id attribute on a table allows you to create a cross-reference to it. Most authoring tools automatically assign IDs to elements that might be cross-referenced.

Your organization or your DITA setup may require additional metadata. It's common to include the language of the topic with the xml:lang attribute.

```
<topic id="xyz" xml:lang="en-us">
<topic id="abc" xml:lang="de-de">
```

As a general rule, your readers do not see the metadata; they see the results of filtering or use the metadata when they search the content.

How do you use metadata?

You can assign metadata to DITA content in several different locations:

- At the topic level
- At the element level
- At the map file level (a map file lets you collect multiple topics to create a document, help system, and so on; a later module explains map files in more detail)

At the topic level, DITA provides a <prolog> element in which you can store metadata for the entire topic. Here is an example of basic topic metadata:

```
<topic id="xyz">
  <title>Metadata example</title>
  <prolog>
    <author>Sarah O'Keefe, Scriptorium</author>
    <critdates>
      <created date="2015-05-01"/>
    </critdates>
  </prolog>
  <body>
    <p>Body content goes here</p>
  </body>
</topic>
```

The author is specified in the <author> element, and <critdates> provides a spot for <created> and <revised> date elements.

Some useful prolog elements are:

- <author> (the content author)
- <critdates> (critical dates, such as <created> and <revised>)
- <copyright> (copyright year <copyyear>, and copyright holder <copyrholder>)

- <vrml> (product version, release, and modification information)

Important: Use the <prolog> metadata only for system information, such as the author and created/revised dates. DITA doesn't use <prolog> metadata to filter topics.

At the element level, you usually use attributes in elements to specify metadata. Here is an example:

```
<step>
  <cmd>Locate the duckling mash box.</cmd>
  <info audience="novice">Consult the side of the duckling mash box to determine
    how much mash your ducklings need.</info>
</step>
<step>
  <cmd>Measure out the mash for your ducks. </cmd>
</step>
<step>
  <cmd> Pour in the blender.</cmd>
</step>
<step>
  <cmd>Put in feeding pan.</cmd>
</step>
```

Only novices need to be reminded that the box provides measurements. When you generate your output, you can suppress the <info> element with the audience = "novice" metadata for an expert-level audience.

At the map level, you can specify metadata in the <topicref> elements that reference topics. This allows you to suppress entire topics when you generate output. Here is an example:

```
<topicref href="abc.dita">
<topicref href="def.dita" audience="novice">
```

By default, DITA provides you with three attributes that support filtering or conditional processing. They are:

- audience
- product
- platform

If you need additional or different filtering attributes, your information architect will need to specialize to define additional metadata. Some common requirements are:

- customer, for customer-specific information
- region, for information that applies only to specific geographic areas
- product-family, for information that applies to a group of products

Lesson 4: Creating DITA content

Objectives

- Describe how to create a DITA topic in a text editor.
- Describe how to create a DITA topic in the <oXygen/> XML editor.
- Describe how to create notes, cautions, and other admonishments.
- Describe how to create bulleted and numbered lists.
- Distinguish block and inline elements.

Creating a topic in a text editor

DITA files are XML, and XML is plain text. Therefore, you can create an XML file in any text editor.

A basic DITA topic needs an XML declaration, a DOCTYPE declaration, a topic, an id attribute on the topic, and a title. That's it.

Here is a minimal valid topic:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="myfirsttopic">
  <title>Hello world</title>
</topic>
```

This topic is valid, but it is not particularly useful because it doesn't have any body content. To make a topic useful, you need something more like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="myfirsttopic">
  <title>Hello world</title>
  <body>
    <p>Here is a paragraph</p>
    <ul>
      <li>bulleted lists are nice</li>
      <li>especially if you have at least two items</li>
    </ul>
    <note>And don't forget the notes.</note>
  </body>
</topic>
```

Creating a topic in oXygen

Reading a DITA file in a text editor can be a bit of a challenge. The important information doesn't stand out, and the hierarchy of the elements may not be apparent (because spaces and carriage returns have

no meaning to XML). Most people find it much easier to use a visual editor (such as SyncROSoft <oXygen/>) to edit DITA files, where the content of the DITA file is presented in a moderately formatted form.

Note: A visual editor such as <oXygen/> may look like a word processor, but the formatting that you see on screen is just one possibility of the way the final output will be presented. Do not rely on the appearance of text (line ends, fonts, character styles, and so on) as the way your content will look when published.

To create a topic in oXygen, follow these steps:

1. Click the New icon ()

2. Navigate to Framework Templates/DITA/Topic, click the Topic entry, then click Create. oXygen creates a new topic with some placeholder content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="topic_xks_f1r_bs">
  <title></title>
  <body>
    <p></p>
  </body>
</topic>
```

oXygen creates a unique topic ID automatically. You can change it to something meaningful if you want to, but remember that IDs need to be unique and must adhere to strict rules. In the Author view, with tags turned on, you see this:



You can now click inside the `p` (which is a placeholder for a `<p>` tag) and start typing in your content. To create a new paragraph, press Return twice. If you want to switch to another element, press Return once and choose from the drop-down menu that appears at your cursor location.

Notes, cautions, and warnings

The `<note>` element lets you create notes, cautions, warnings, and other admonishments.

```
<note>
```

```
Most surface-feeding ducks are agile and will take to the air easily. Approach these ducks slowly.
```

```
</note>
```

The `type` attribute for the `<note>` element allows you to indicate different kinds of notes. DITA defines a number of values for the `type` attribute:

- `note` (the default)
- `attention`
- `danger`
- `caution`
- `important`
- `tip`

```
<note type="warning">
Do not feed bread to ducks. It is not a natural food for them and can be unhealthy.

It's much better to feed them grapes or cracked corn.
</note>
```

These are the most common attribute values. There are additional attribute values available, along with a catch-all “other” attribute value, which you use with the `othertype` attribute to specify an alternate note type.

Bulleted and numbered lists

The most common lists used in DITA are unordered (“bulleted”) lists (``) and ordered (“numbered”) lists (``).

Both ordered and unordered lists contain one or more list item (``) element. The `` elements shown in these examples are quite simple. However, an `` element can contain text data and most common block and inline elements.

```
<p>The life cycle of a duck is similar to that of most birds:</p>
<ol>
  <li>Egg</li>
  <li>Hatchling</li>
  <li>Duckling</li>
  <li>Juvenile</li>
  <li>Adult</li>
</ol>
```

Note that the author does not add numbers to this ordered list; as with HTML, the numbers are inserted when the output is generated. To help you visualize things, most visual editors will show you the numbers of items in an ordered list.

An ordered list is useful for showing things that have a sequence, however, if you need to describe a process, you might be better off using the `<steps>` element in a task topic.

You can nest a list by starting a new `` or `` element inside an `` element.

```
<p>Ducks have these features:</p>
<ul>
  <li>Webbed feet for swimming.</li>
  <li>Shorter legs than other waterfowl.</li>
  <li>Distinctive bill.
    <ul>
      <li>Wide and flat for filter feeding.</li>
      <li>Long and narrow for fishing (Mergansers)</li>
    </ul>
  </li>
</ul>
```

When the list is finally output, the bullet character or the number used in the list is determined by the stylesheet being used for output and the position in the list in the hierarchy. If a list is nested, the stylesheet will determine the amount of indentation for the list.

In addition to the unordered list (), DITA defines a simple list (<sl>) element, which you can use for lists that contain just a few words in each item. The item in a simple list uses the <sli> element.

```
<p>There are several types of ducks:</p>
<sl>
  <sli>Tree ducks</sli>
  <sli>Surface-feeding ducks</sli>
  <sli>Bay ducks</sli>
  <sli>Sea ducks</sli>
</sl>
```

Block versus inline elements

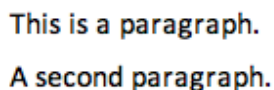
Separation of content and formatting was probably the first thing you learned about XML.

There are a few cases, however, where formatting is built into DITA/XML/HTML content itself. The distinction between block and inline elements is one of those cases:

A *block element* is a paragraph-level element, such as <p> tag (paragraph), (list item), or <codeblock> (software code). As a general rule, a block element is separated vertically on the page from the information before and after it. So if you have two <p> tags in a row, they are separated by a line break when the information is rendered. Note that any line breaks characters in XML itself are irrelevant.

```
<p>This is a paragraph. </p><p>A second paragraph. </p>
```

The result, in oXygen's Author view:



An *inline element* is an element that provides for markup inside a paragraph, such as <tm> (trademark), <i> (italics), <xref> (cross-reference), or <term> (term). Inline elements, by default, do not result in line breaks.

```
<p>An inline element <i>does not</i> result in line breaks.</p>
<p>The line breaks in the XML are
  <i>irrelevant</i>
to the output formatting.</p>
```

oXygen ignores the line breaks in the code when you look at Author view:

An inline element *does not* result in line breaks.

The line breaks in the XML are *irrelevant* to the output formatting.

XML itself does not distinguish between block and inline elements. They use the same markup, so you have to know which elements are blocks and which are inline to figure out how information will be presented. Inline elements are similar to character styles in word processing tools. However, there are other inline elements, such as <fn> (footnote) and <indexterm> (index entry) that are processed differently. For example, <indexterm> entries are processed to create a back-of-the-book index. The text content of <indexterm> does not generally appear at the location where the inline element occurs.

```
<p>An inline element<indexterm>inline element</indexterm> <i>does not</i> result in  
line breaks.</p>
```

Lesson 5: Tables

Objectives

- Distinguish simpletables and tables.
- Describe the elements in a simpletable.
- Describe the elements in a table.
- Present best practices for DITA tables

Tables occupy an uneasy middle ground between formatting and structure. However, tables are often the right container for the content.

DITA include two types of tables:

Simple tables

The <simpletable> element supports basic tables with minimal customization

Tables

The <table> element can support complex tables with spanned rows and columns (straddles) and precise display properties

Most markup languages uses the CALS table model, which was created as part of an initiative of the United States Department of Defense. HTML tables also use the CALS model, so if you are familiar with HTML table markup, you will recognize similarities in DITA tables.

DITA tables are always described row by row.

Simple tables

Simple tables are ideal for basic representation of content within columns and rows. You can use heading rows, but you cannot use a table title, spanned columns or rows, or define column widths.

Tables

The `<table>` element allows for more complex arrangements of tabular content. It allows table titles, spanned columns or rows, multiple groups of rows, and user-specified column widths

Other tables

There are two other very specific table types (based on the `<simplatable>` element):

- `<properties>`, which you use in reference topics to define a list of properties.
- `<choicetable>`, which you use in task topics to present the differences among various choices.

The `simplatable` elements

Simple tables (`<simplatable>`) use the following elements to represent and organize tabular data:

Element	Description	Number
<code><sthead></code>	A header row	There can be at most one header row.
<code><strow></code>	A body row	There must be at least one body row, but the table can contain many rows.
<code><stentry></code>	A simple table entry	Each row can contain one or more <code><stentry></code> elements, but each row (including the header row) should include the same number of <code><stentry></code> elements.

Note: This table is built using a `<simplatable>` element.

The `<stentry>` element can contain text data or other common block and inline elements.

Here is an example of a table with one heading row, two body rows, and two columns.

```
<simplatable>
  <sthead>
    <stentry>heading for column 1</stentry>
    <stentry>heading for column 2</stentry>
  </sthead>
  <strow>
    <stentry>row 1, column 1</stentry>
    <stentry>row 1, column 2</stentry>
  </strow>
  <strow>
    <stentry>row 2, column 1</stentry>
```



```

        <stentry>row 2, column 2</stentry>
    </strow>
</simpletable>

```

Visually, the table looks like this:

heading for column 1	heading for column 2
row 1, column 1	row 1, column 2
row 2, column 1	row 2, column 2

The table element

DITA tables (<table>) use the following elements to represent and organize tabular data:

Element	Description	Number
<title>	Contains the title for the table.	Zero or one.
<tgroup>	Contains the columns specifications, header rows and body rows.	Every table must contain at least one <tgroup> element.
<colspec>	Defines column widths and identifying information.	There should be one <colspec> element per column in your table.
<thead>	Contains the table header rows.	A <tgroup> can contain at most one <thead> element.
<tbody>	Contains the table body rows.	A <tgroup> must contain one and only one <tbody> element.
<row>	Contains a single row in a table.	A <thead> or <tbody> element can contain any number of <row> elements.
<entry>	Contains data for a table cell. Can contain text data or other common block and inline elements.	A <row> element can contain one or more <entry> elements.

This is an example of a minimal DITA table structure (one heading row, one body row, and three columns). Note that in this case, the <tgroup> specifies only the cols attribute, which is required:

```

<table>
  <tgroup cols="3">
    <thead>
      <row>
        <entry>column 1, heading row</entry>

```

```

        <entry>column 2, heading row</entry>
        <entry>column 3, heading row</entry>
    </row>
</thead>
<tbody>
    <row>
        <entry>column 1, body row</entry>
        <entry>column 2, body row</entry>
        <entry>column 3, body row</entry>
    </row>
</tbody>
</tgroup>
</table>

```

The result looks like this:

column 1, heading row	column 2, heading row	column 3, heading row
column 1, body row	column 2, body row	column 3, body row

Here is a more complex table:

```

<table>
    <title>My first table</title>
    <tgroup cols="2">
        <colspec colname="c1" colnum="1" colwidth="1*"/>
        <colspec colname="c2" colnum="2" colwidth="4*"/>
        <thead>
            <row>
                <entry>heading row, column 1</entry>
                <entry>heading row, column 2</entry>
            </row>
        </thead>
        <tbody>
            <row>
                <entry>row 1, column 1</entry>
                <entry>row 1, column 2</entry>
            </row>
            <row>
                <entry namest="c1" nameend="c2">This cell spans two columns.</entry>
            </row>
            <row>
                <entry morerows="1">This cell spans two rows.</entry>
                <entry>row 3, column 2</entry>
            </row>
            <row>
                <entry>row 4, column 2</entry>
            </row>
        </tbody>
    </tgroup>
</table>

```

```

    </tgroup>
</table>

```

Which looks like this:

heading row, column 1	heading row, column 2
row 1, column 1	row 1, column 2
This cell spans two columns.	
This cell spans two rows.	row 3, column 2
	row 4, column 2

Some notes on the table:

- The `<colspec>` element uses attributes to specify the column names (colname), numbers (colnum), and widths (colwidth). In this example, the colwidths are “1*” and “4*”. The asterisk indicates that the settings are proportional, so the first column gets 20% of the available width and the second column gets 80%.
- Each row has a `<row>` element with `<entry>` elements for each cell.
- Notice that row 2 has only a single `<entry>`. That’s because the entry spans across both columns, which is specified by the name-start (namestart) and name-end (nameend) attributes.
- In row 3, the first entry spans rows 3 and 4, so there you have a `morerows` attribute.

The easiest and least stressful way to set up the table code correctly is to use an editor that manages these settings for you. Hand-coding tables is not for the faint-hearted.

Best practices for tables

- Always wrap `<entry>` content in the table in `<p>` tags. Forgetting this can result in the content of the table to format in unexpected ways.
- While the DITA specification allows you to nest tables, it’s a Bad Idea™.
- Whenever possible, consider organizing the content in your table to include more rows instead of more columns. When you send the content to an output, pagination issues can make many columns appear in unexpected ways.
- In many word processing tools, people use tables to visually format content such as indented lists. In DITA, this is not done and can lead to unexpected results.
- Consider localization text expansion in tables. Where possible, design the content for up to 40% text expansion.
- The DITA standard allows graphics in table cells. However, in most cases it’s best to limit these graphics to small icons in table cells.
- Consider the possible outputs when you create tables. Table rendering may be very different (and hard to read) on a small screen, such as a mobile device.

- In some cases you might want to consider using other elements, such as definition lists, which can be rendered either as a table or a glossary-style list, depending on the target output.

Lesson 6: Creating relationships among topics

Objectives

- Describe the purpose and use of a map file.
- Summarize how output is generated.
- Describe the purpose and use of cross-references (xref) and related-links.
- Describe the purpose and use of a relationship table (reltable).
- Describe the purpose and use of a content reference (conref).

Topics stand on their own, but content in one topic often has relationships to other topics. A map file, for example, describes the sequence and hierarchy of topics for a deliverable.

You can create relationships among topics in several different ways:

- Map files
- Conrefs
- Cross-references
- Related links
- Relationship tables

Map files

Map files are how you organize content for delivery. They are like a table of contents: they create sequence and hierarchy among topics. When you generate a PDF file or a help system from a map file, your reader sees the topics in the order and hierarchy established by the map file.

You generally do not add all available topics to a map file—just the ones you want included in a deliverable. Also, you can include the same topic in multiple map files, which is another example of reuse in DITA.

Map files are made up mainly of the following components:

- `topicref` elements, which provide a link to a specific topic
- `mapref` elements, which provide a link to another map

In a map file, you put `topicrefs` in order from top to bottom to indicate sequence. To indicate hierarchy, you nest the topics. Consider the following example:

```

<map>
  <title>My first map</title>
  <topicref href="ducks.dita">
    <topicref href="range.dita"/>
    <topicref href="size.dita"/>
    <topicref href="nests.dita"/>
  </topicref>
</map>

```

For convenience, the code is indented. But what matters is that the first topicref (ducks) encloses the other three files. The range.dita, size.dita, and nests.dita topics are all subordinate to the ducks.dita topic. The result is a table of contents that is structured like this:

- Ducks
 - Range
 - Size
 - Nests

In addition to linking to topics, you can reference map files inside map files. In this approach, the subordinate maps (submaps) are usually a collection of related content. For example, you can create a chapter-level map file for each chapter in a book, and then reference those chapter-level map files inside your main book-level map file.

Instead of a topicref, a reference to a map uses a mapref. If you want to use duck map content as a component of another map, you would insert it into the parent map, as shown here:

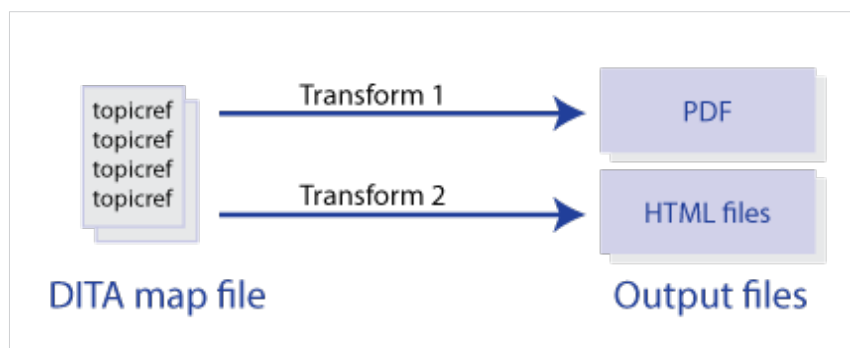
```

<topicref href="fish.dita">
<topicref href="shorebirds.dita">
<mapref href="ducks.ditamap" format="ditamap"/>

```

Transforming map files into outputs

Most content delivery does not use DITA files. Instead, you apply stylesheets to the DITA files to create HTML, PDF, or other formats. This process is called transformation.



The DITA Open Toolkit is a collection of stylesheets, mostly written in Extensible Stylesheet Language (XSL), that provide a starting point for creating output. The default output from an unmodified DITA Open Toolkit transformation is notoriously unattractive. Usually the transformations are enhanced before the output is used in a production environment.

Note: The Open Toolkit is not the only way of generating output from DITA. You can use commercial tools to create output. Some tools provide only publishing support; others also provide authoring support for DITA content.

Transforms are independent of map files, and you can apply multiple transforms to a single map file. For example, you could run the transforms for a PDF file and an online help system on the same map file to get two outputs.

Additional reading

[DITA Open Toolkit](#)

[Getting started with the DITA Open Toolkit, Suite Solutions webinar \(video, 53 minutes\)](#)

Cross-references

Cross-references allow you to create links from text to other locations in text, to figures or tables, or to web sites.

All cross-references use the <xref> element. The href attribute specifies the target of the cross-reference.

The <xref> element can optionally contain text. If text is provided, it is used as the hotspot text for the link. If the <xref> element doesn't contain any text, the output generator uses the title of the target as the hotspot for the link.

Note: Although it might seem like the right thing to do, when creating a cross-reference to a section, figure, or table, the href attribute should specify the section, figure, or table, not the <title> element within the section, figure, or table.

Video: [Code examples of DITA cross-references](#)

You can also use the <xref> element to link to resources outside your DITA topics. For external references, you supply a scope attribute with a value of external and a format attribute, as shown here:

```
<xref href="http://www.scriptorium.com" scope="external" format="html"/>
```

Note: For links to a PDF file, use format="pdf".

Technically, it is possible to use <xref> to link from one topic to another. This inline linking is a Bad Idea™ because you have to create and maintain the link manually. You must specify what you are linking to. When you set up the cross-reference, you have the source and target topics in your map file. But if you reuse the source topic in another map that does not include the target topic, you will get a

broken link in the output generated from that map, and you may not be notified about the problem. Instead of using topic-to-topic links, it's much better to use relationship tables.

Related links

At the end of a topic, you can insert a related-links element. Use related-links to point at additional information that a reader might want.

Related links contain a link and link text. Here is an example:

```
<topic id="sample">
  <title>Sample title</title>
  <body>
    ...
  </body>
  <related-links>
    <link href="http://www.example.com" format="html" scope="external">
      <linktext>Sample link</linktext>
    </link>
  </related-links>
</topic>
```

In the link element, you specify the target (using the href attribute), the format, and the scope. For web links and any other link that is outside the current map file, you must specify scope="external".





The linktext element contains the text that will become the link hotspot.

Note: Avoid using related links for links among the topics in your map file. They suffer from the same limitations as inline cross-references—you can end up with broken links. Use relationship tables instead.

Relationship tables

Relationship tables, or reltables, allow you to describe topic relationships that are not sequential or hierarchical. A reltable is a part of a map file; it can appear at any location in the map, but the convention is to add the reltable at the end of the main map. Each row of the table contains topicref elements that link to related topics.

Here is an example of a reltable row (and the reltable heading row), in which the columns contain topicrefs for concept, reference, and task information. The row shown here contains topicrefs to related topics.

concept	reference	task
 c_about_ducks.dita	 r_breedsofducks.dita  r_goodbreedsforpets.dita	 t_feeding.dita

When you generate output through the DITA Open Toolkit, the relationships described in the reltable are used to create a list of links to the related topics. In the default HTML output, the reltable entries are used to create a Related Topics section at the end of each topic.

In the reltable row shown here:

- The concept topic `c_about_ducks.dita` would have related links to `r_breedsofducks.dita`, `r_goodbreedsforpets.dita`, and `t_feeding.dita`.
- The reference topic `r_breedsofducks.dita` would have links to `c_about_ducks.dita`, `r_goodbreedsforpets.dita`, and `t_feeding.dita`.
- The reference topic `r_goodbreedsforpets.dita` would have links to `c_about_ducks.dita`, `r_breedsofducks.dita`, and `t_feeding.dita`.
- The task topic `t_feeding.dita` would have links to `c_about_ducks`, `r_breedsofducks.dita`, and `r_goodbreedsforpets.dita`.

Note: To illustrate a larger concept, this example contains a simplification: for `r_breedsofducks.dita` and `r_goodbreedsforpets.dita` to link to one another, the cell containing both links must identify them as a “family”. The details relationship table cells attributes will be described in a future course.

The relationships you capture in the reltable are *not* typically shown when you are authoring topics.

Video: [Overview of DITA relationship table \(reltable\)](#)

Start simple with reltables. They can get very complex.

Reltables are preferred over related links or xrefs for links within a map file because of the following factors:

- The topicrefs in the reltable are evaluated against the current map file. If your reltable contains a link to a file that is not included in the map file, that link is not generated in the output. This prevents the broken link problem that can occur with related links and xrefs.
- The reltables are easier to maintain than embedded related links. Each row in a reltable can contain multiple topics and captures their interrelationships. So, if you have eight related topics, it is much easier to create a single row in a reltable that lists those eight topics than it is to create eight slightly different related-links lists (consisting of 8×7 or 56 links) in your eight topics. If you need to remove one topic from the list, the reltable change is done once rather than seven times in the files.

Reusing content with conrefs

In DITA, you use a conref to reuse pieces of content. These content pieces could be admonishments (notes, cautions, and warnings), boilerplate text (such as your company address), and more. For example, a product’s description could be set up as a conref so it is the same across all your documents.

If you are familiar with other authoring tools, conrefs are roughly equivalent to:

- Flare snippets
- RoboHelp embedded topics
- FrameMaker text insets and variables
- HTML server-side includes

Suppose you have the following note in a topic called location.dita:

```
<topic id="topicid">
  ...
  <note id="whatduckslike">Ducks prefer lakes over deserts.</note>
  ...
</topic>
```

To reuse that note in another topic, the conref code is as follows:

```
<note conref="location.dita#topicid/whatduckslike"/>
```

Most DITA authoring tools streamline the process for inserting conref code, so you probably will not type the file path, topic ID, and so on, as shown in the example.

Note: The target element must be the same element type as the element that contains the conref attribute. Thus you can only conref a <note> element from another <note> element; you can only conref an element from another element, and so on.

Video: [DITA conref code example](#)

Tip: It's a good practice to create separate topics that contain reusable elements, rather than just picking a random topic and pointing a conref to that topic. This helps you keep better tabs on the content that is reused and allows you to control when it changes.

Course II. The DITA Concept Topic Type

Lesson 1: Creating a concept topic

Objectives

- Create a concept topic file
- Learn about the concept DOCTYPE
- Add the root elements for a concept
- Add basic content elements, including `<p>`, ``, ``, and `<note>`

DITA concept topics provide information about a particular subject.

Concept topics help someone understand an idea or the purpose of an instruction. A concept topic may give background or conceptual information about the subject in addition to essential information.

A typical concept topic will mostly contain paragraphs and lists. However, you can use images, tables, and many other elements in a concept. You can also break a concept into sections for additional clarity.

This lesson shows how to create a new concept topic and how to add commonly used elements. With the exception of the `<conbody>` element, the elements presented in this lesson and throughout the rest of the course are not specific to concepts and can also be used in the other types of DITA topics (task and reference).

Note: This lesson covers basic use of these elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Concept](#)

[DITA Style Guide, Paragraphs](#)

[DITA Style Guide, Lists](#)

Housekeeping and sample files

Download [concept_samples.zip](#) now. It contains the sample files for the entire Authoring DITA concepts course. Extract the contents and put them in a directory that you can access easily.

Inside the samples folder, you will find the following folders corresponding to lessons in this course: lesson1, lesson2, lesson3, and lesson4.

In each lesson folder, you will find four DITA files:

- `l_filename_start.dita`: The empty sample file you will add content to as you follow along with each lesson.
- `l_filename.dita`: The completed version of the sample file you can use to check your work.
- `l_filename_exercise_start.dita`: The empty file you will add content to as you complete the exercise at the end of each lesson.
- `l_filename_exercise.dita`: The completed version of the exercise file you can use to check your work.

Each lesson will instruct you on which files to use for the exercises. Save your file as you complete each step to avoid losing your work.

Create a local copy of each file to work in as you complete the lessons. That way, if you reach a point where your working file doesn't match the examples, or is broken for any reason, you can make a fresh copy and resume your work or start over.

In the instructions and examples, we show you the DITA code for each sample file. Most DITA editors have auto-complete or other similar features to guide you through the process of adding elements (for example, if you type the opening tag of an element, most DITA editors will automatically add the closing tag for you). Therefore, you will probably not need to create every piece of code from scratch as you work. Our demo videos were created in [oXygen XML Editor](#) and show the differences between working in author view, which presents the DITA content in a user-friendly visual format, and working in text view, which shows the DITA code.

Creating a new concept topic

Concept topics contain basic DITA elements, such as paragraphs, lists, images, tables, and many others. Most of these elements can be used in the other topic types (task and reference), as well. The only exception is the `<conbody>` element, which is specific to the concept topic type.

At a minimum, the concept topic must contain a `<concept>` root element (with an `id` attribute) that contains a `<title>` element.

After the `<title>` element and an optional `<shortdesc>` or `<abstract>` element, the `<conbody>` element contains the concept itself.

The `<conbody>` element has an open content model. This means that the elements contained in `<conbody>` are not required to follow a strict sequence. As long as an element is allowed inside `<conbody>`, it can appear in any order. The exceptions are the `<section>` and `<example>` elements, which can only be followed by other `<section>` or `<example>` elements or a `<conbodydiv>` element.

Video: [Creating a DITA concept](#)

Practice

1. Make a copy of the file `lesson1/l_new_concept_start.dita` and open it in your editor.

Note:

If you are using a DITA-aware text editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="my_first_concept">
<title>xyz</title>
</concept>
```

The first line (which begins with `<?xml>`) is an XML declaration, which is a standard part of all XML files.

The DOCTYPE declaration on the second line identifies this file as a concept topic.

The third line is the opening tag of the `<concept>` element, which uses the unique ID “my_first_concept”.

The `<title>` element on the fourth line contains the title of the topic.

The fifth line uses the closing tag `</concept>` to show where the `<concept>` element ends.

2. Inside the `<title>` element, change the text of the title as shown in the following example.

Note: When working inside an element, insert the content between the opening and closing tags.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="my_first_concept">
  <title>Wild duck species</title>
</concept>
```

3. After the `<title>` element, add a `<conbody>` element.

Note: When adding an element after another element, insert the new element after the closing tag of the first element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="my_first_concept">
  <title>Wild duck species</title>
  <conbody>
</conbody>
</concept>
```

The `<conbody>` element will contain all the actual content in the concept.

Adding basic elements to a concept topic

Some of the most commonly used elements in a concept topic include:

<p>	paragraph
	unordered list
	ordered list
<note>	admonition

These elements were covered in the [Introduction to DITA course](#). This lesson gives a brief review of these elements and shows how to add them to a concept topic.

Practice

1. Continue using the file lesson1/l_new_concept_start.dita.
2. Inside the <conbody> element, add a <p> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="my_first_concept">
  <title>Wild duck species</title>
  <conbody>
    <p>North American wild ducks belong to one of the following categories:</p>
  </conbody>
</concept>
```

The <p> element is for body text. It can be used inside the <conbody> element as well as inside many of the elements that <conbody> contains.

Note:

Use the <p> element for any body text that does not need a more specific element tag. As you will see in examples throughout this course, we recommend using the <p> element to tag text inside list items, notes, and table entries. <p> elements are needed for list items that are several paragraphs long, and wrapping single items in the <p> element keeps them all consistent.

The <p> element you just created introduces the next element: an unordered list.

3. After the <p> element, add a element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="my_first_concept">
  <title>Wild duck species</title>
  <conbody>
    ...
    <ul>
      <li><p>Dabbling ducks</p></li>
```

```

<li><p>Diving ducks</p></li>
<li><p>Sea ducks</p></li>
<li><p>Whistling ducks</p></li>
<li><p>Swans and geese</p></li>
</ul>
  </conbody>
</concept>

```

Note:

If you are using a DITA editor, the editor may insert an ID attribute on the element for you automatically.

The element is for unordered or bulleted lists.

The element can only contain the element, and it must contain at least one.

Each element contains the text for a bulleted entry in the list. In this example, the elements follow the best practice of wrapping a <p> element around the text.

4. After the element, add an element with an introductory <p> element as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="my_first_concept">
  <title>Wild duck species</title>
  <conbody>
    ...
    <p>The longest species of dabbling ducks in North America are:</p>
    <ol>
      <li><p>Northern pintail</p></li>
      <li><p>Mallard</p></li>
      <li><p>American black duck</p></li>
    </ol>
  </conbody>
</concept>

```

Note:

If you are using a DITA editor, the editor may insert an ID attribute on the element for you automatically.

The element is for ordered or numbered lists.

Like the element, the element can only contain elements, and it must contain at least one.

In this example, each of the elements follows the best practice of wrapping a <p> element around the text.

Note:

Use the element only to show that the list items should appear in a certain order, not to create step-by-step instructions. For step-by-step instructions, use the task topic instead.

5. After the element, add a <note> element and add content to it as shown in the following example:

Video: [Creating a note in DITA](#)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="my_first_concept">
  <title>Wild duck species</title>
  <conbody>
    ...
    <note><p>Although the northern pintail is the longest dabbling duck, the mallard
    is generally considered the largest because of its heavier body weight.</p></
    note>
  </conbody>
</concept>
```

The <note> element is used to add admonitions, such as notes, warnings, or cautions, to the concept topic. The <note> element you just created follows the best practice of wrapping a <p> element around the text.

6. Check your file lesson1/l_new_concept_start.dita against the sample file lesson1/l_new_concept.dita.

Practice

1. Open the file lesson1/l_new_concept_exercise_start.dita and use it to convert the following content from [Content Strategy 101](#) into DITA:

Content strategy and business goals

The issues to consider are:

- The real cost of low-cost documentation
- How to create an efficient content development process
- Whether high-quality documentation can lower the cost of technical support
- The most cost-effective way to share technical content across the enterprise

Note: Ignoring content can have cost implications across the organization.

To implement your project and improve your chances of success, we recommend following this sequence:

1. Identify and interview stakeholders.
 2. Establish implementation goals and metrics.
 3. Define roles and responsibilities.
 4. Establish timelines and milestones.
 5. Build the content creation system.
 6. Convert legacy content.
 7. Deliver content.
 8. Capture project knowledge.
 9. Ensure long-term success.
2. Check your file `lesson1/l_new_concept_exercise_start.dita` against the sample file `lesson1/l_new_concept_exercise.dita`.

Lesson 2: Images and tables

Objectives

- Identify the best practices for adding images to a concept topic and the elements involved.
- Add the `<fig>` and `<image>` elements to the concept.
- Identify the basic elements involved in adding tables to a concept topic.
- Add the `<simpletable>` and `<table>` elements to the concept.

Images and tables give you a visual way to present or clarify information in your concept topic.

Using the `<image>` element on its own or inside a `<fig>` element, you can include an image in your topic through a link to that image's location on the web or a local server.

With the `<simpletable>` and `<table>` elements, you can create tables to convey factual information in a clear visual format. The `<simpletable>` element is ideal for a small, basic table that accompanies other text, while the `<table>` element is ideal for complex table designs.

This lesson shows how to add images and tables to a concept topic. Like the other elements introduced in this course, images and tables are not exclusive to the concept topic type and can also be used in the other DITA topic types (task and reference).

Note:

This lesson covers basic use of these elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Graphics and figures](#)

[DITA Style Guide, Tables](#)

Adding images to a concept topic

You add images to a concept topic using the `<image>` element. The `<image>` element uses the `href` attribute to create a link to the image's location on the web or on a local server.

Much like a `<p>` element, you can use the `<image>` element anywhere in the `<conbody>` element. It is also fairly common to use images inline inside a `<p>` element.

If your image needs a title or caption, use the `<fig>` (or figure) element. The `<fig>` element can contain a `<title>` element and one or more `<image>` elements.

Video: [Creating a figure with an image in DITA](#)

Practice

1. Make a copy of the file `lesson2/l_concept_images_tables_start.dita` and open it in your editor.

You should see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
<conbody>
  </conbody>
</concept>
```

2. Inside the `<conbody>` element, add an `<image>` element as shown in the following example, using the file `lesson2/images/ducklings_swimming.jpg` for the sample image:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
    <image href="images/ducklings_swimming.jpg"/>
  </conbody>
</concept>
```

(Sample image source: [Flickr, Micolò J](#))

The `<image>` element you added contains a link (`href`) to the location of the image file. A link inside an `<image>` element can point to the filepath for an image on a server or to a web link for an image.

3. After the `<image>` element, add a `<fig>` element as shown in the following example, the file `lesson2/images/ducklings_running.jpg` for the sample image:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
    ...
  <fig>
    <title>Ducklings running</title>
    <image href="images/ducklings_running.jpg"/>
  </fig>
</conbody>
</concept>
```

(Sample image source: [Flickr, Airwolfhound](#))

The `<fig>` element is another way to add an image to a concept topic.

The `<fig>` element you added contains a `<title>` element and an `<image>` element. The `<title>` element allows you to associate a caption with your image.

The `<image>` element is structured the same way inside the `<fig>` element as it is inside the `<conbody>` element.

The `<fig>` element can contain one or more `<image>` elements (but only one `<title>` element).

Note:

In a `<fig>` element, the `<title>` element always goes before the `<image>` element in DITA, but the caption could appear above or below the image (or elsewhere) in the final output.

Adding simple tables to a concept topic

There are two elements available for creating tables in a concept topic: `<simpletable>` and `<table>`.

Use the `<simpletable>` element whenever you can, especially for short tables. Use the `<table>` element for larger tables or tables that need more complex formatting.

The elements contained in the `<simpletable>` element include:

<code><sthead></code>	The container element for the header row of a <code><simpletable></code> element. The <code><sthead></code> element contains the <code><stentry></code> element and is optional. There can only be one <code><sthead></code> element in a <code><simpletable></code> element.
<code><strow></code>	The container element for the body rows of a <code><simpletable></code> element. The <code><strow></code> element contains the <code><stentry></code> element. A <code><simpletable></code> element can have one or more <code><strow></code> elements.
<code><stentry></code>	The container element for a single cell in a <code><simpletable></code> element. The <code><stentry></code> element contains the text of a cell in the <code><simpletable></code> element, which should be

This lesson will cover the basics of using the <simpletable> element.

Figure 1. Example <simpletable> in a visual format

Age	Milestone
7 weeks	Attempt flight for the first time
12-14 weeks	Reach adult body weight
1 year	Capable of reproduction

Continue using the file lesson2/1_concept_images_tables_start.dita to add the <simpletable> element.

Note: If you are using a DITA editor, some child elements of the <simpletable> element will automatically be inserted as you work through the examples.

Practice

1. Inside the <conbody> element after the <fig> element, add a <simpletable> element as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
    ...
  </conbody>
</concept>
```

2. Inside the <simpletable> element, add the <sthead> element as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
    ...
    <simpletable>
      <sthead>
      </sthead>
    </simpletable>
  </conbody>
</concept>
```

The <sthead> element sets up the header row of the <simpletable> element.

3. Inside the <sthead> element, add two <stentry> elements and add content to them as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
    ...
  <sthead>
    <stentry><p>Age</p></stentry>
    <stentry><p>Milestone</p></stentry>
  </sthead>
  ...
  </conbody>
</concept>
```

The <stentry> element represents a single cell. The number of <stentry> elements inside the <sthead> element will determine how many columns will be in your <simpletable> element.

Each <stentry> element you added follows the best practice of wrapping a <p> element around the text.

4. After the <sthead> element, add the <strow> element as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
    ...
  </sthead>
  <strow>
  </strow>
  ...
  </conbody>
</concept>
```

The <strow> element sets up each body row following the header row of the <simpletable> element.

5. Inside the <strow> element, add two <stentry> elements and add content to them as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
```

```

    <tbody>
    ...
    <tbody>
    <tbody><tr>7 weeks</tr></tbody>
    <tbody><tr>Attempt flight for the first time</tr></tbody>
    </tbody>
    ...
    </tbody>
</table>

```

Every `<tbody>` element and the `<thead>` element should contain the same number of `<tbody>` elements.

6. After the first `<tbody>` element, add two more rows to the `<table>` element and add content to them as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
    <title>Duckling growth and development</title>
    <tbody>
    ...
    </tbody>
    <tbody>
    <tbody><tr>12-14 weeks</tr></tbody>
    <tbody><tr>Reach adult body weight</tr></tbody>
    </tbody>
    <tbody>
    <tbody><tr>1 year</tr></tbody>
    <tbody><tr>Capable of reproduction</tr></tbody>
    </tbody>
    ...
    </tbody>
</concept>

```

You can add one or more `<tbody>` elements to a `<table>` element.

Adding tables to a concept topic

Use the `<table>` element when you need to create a larger or more complex table than you can with the `<table>` element. If you need a title or caption for your table, or more than one header row, use the `<table>` element.

The elements contained in the `<table>` element include:

<code><title></code>	The container element for the text of a <code><table></code> element's title or caption. A <code><table></code> element can only contain one <code><title></code> element.
<code><tbody></code>	The container element for the main content of a

	<code><table></code> element. The <code><tgroup></code> element contains the <code><colspec></code> , <code><thead></code> , and <code><tbody></code> elements. A <code><table></code> element can contain one or more <code><tgroup></code> elements.
<code><colspec></code>	The element that provides information about the columns in a <code><table></code> element. The column specifications are defined using the <code>colname</code> (the name of the column), <code>colnum</code> (the order in which the column appears), and <code>colwidth</code> (the relative width of the column) attributes.
<code><thead></code>	The container element for the header in a <code><table></code> element. The <code><thead></code> element can contain one or more <code><row></code> elements and is optional.
<code><tbody></code>	The container element for the body rows in a <code><table></code> element. The <code><tbody></code> element can contain one or more <code><row></code> elements.
<code><row></code>	The container element for a row of cells in a <code><table></code> element. The <code><row></code> element contains one or more <code><entry></code> elements.
<code><entry></code>	The container element for a single cell in a <code><table></code> element. The <code><entry></code> element contains the text of a cell in the <code><table></code> element, which should be wrapped in a <code><p></code> element according to best practice.

This lesson will cover the basics of using the `<table>` element. To highlight the differences between the `<table>` and `<simptable>` elements, you will be using the same sample content for your `<table>` exercise as you used for your `<simptable>` exercise.

Figure 2. Example `<table>` in a visual format

Age	Milestone
7 weeks	Attempt flight for the first time
12-14 weeks	Reach adult body weight
1 year	Capable of reproduction

Continue using the file `lesson2/1_concept_images_tables_start.dita` to add the `<table>` element.

Note:

If you are using a DITA editor, some child elements of the <table> element will automatically be inserted as you work through the examples.

Video: [Creating a table in DITA](#)

Note:

This video shows an alternative way to create a table in DITA using the [oXygen XML Editor](#) table wizard. The exercises show you how to create a table using DITA code.

Practice

1. After the <simpletable> element, add a <table> element as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
    ...
  </simpletable>
  <table>
  </table>
</conbody>
</concept>
```

2. Inside the <table> element, add a <title> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
    ...
  <table>
    <title>Typical development of mallard ducks</title>
  </table>
</conbody>
</concept>
```

The <title> element is optional, but it allows you to add context to your table, just as it does for images when used inside the <fig> element.

3. After the <title> element, add the <tgroup> element as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
```

```

    <title>Duckling growth and development</title>
    <tbody>
    ...
    <title>Typical development of mallard ducks</title>
    <tgroup cols="2">
    </tgroup>
    ...
    </tbody>
</concept>

```

The `<tgroup>` element contains the body of the table. The number of table columns is set inside the `<tgroup>` element using the `cols` attribute. In the example you added, `cols="2"` shows that this table will contain two columns.

A single `<table>` element can contain more than one `<tgroup>` element, which is useful if you need to show more than one table with different headers or numbers of columns under the same title.

4. Inside the `<tgroup>` element, add two `<colspec>` elements and add content to them as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
    <title>Duckling growth and development</title>
    <tbody>
    ...
    <tgroup cols="2">
    <colspec colname="c1" colnum="1" colwidth="1.0*"/>
    <colspec colname="c2" colnum="2" colwidth="1.0*"/>
    ...
    </tbody>
</concept>

```

The `<colspec>` element sets up your table columns and specifies information about them (such as their names, sequence, and widths) using attributes. In the example you added, the `colname` attribute was used to name your columns “c1” and “c2” and the `colnum` attribute was used to designate the order in which these columns appear.

The optional `colwidth` attribute can be used to control the proportions of the column widths. In the example you added, the `colwidth` attribute is the same for each column, meaning that they will display at equal widths.

5. After the last `<colspec>` element, add the `<thead>` element as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
    <title>Duckling growth and development</title>

```



```

    <tbody>
    ...
    <colspec colname="c2" colnum="2" colwidth="1.0*"/>
    <thead>
    </thead>
    ...
  </tbody>
</concept>

```

The `<thead>` element contains the elements associated with the table header. Unlike the `<thead>` element in a `<table>` element, the `<thead>` element in a `<table>` element can contain more than one header row.

6. Inside the `<thead>` element, add the `<row>` element as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <tbody>
  ...
  <thead>
    <row>
  </row>
  ...
</concept>

```

The `<row>` element contains the elements associated with a table row.

7. Inside the `<row>` element, add two `<entry>` elements and add content to them as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <tbody>
  ...
  <row>
    <entry><p>Age</p></entry>
    <entry><p>Milestone</p></entry>
  ...
</concept>

```

Each `<entry>` element corresponds to a single table cell in a row. In the example you added, each `<entry>` element follows the best practice of wrapping a `<p>` element surrounding the text.

8. After the `<thead>` element, add the `<tbody>` element as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
  ...
</thead>
<tbody>
</tbody>
...
</conbody>
</concept>

```

The <tbody> element is for the main body of the table. Following the same structure as the <thead> element, the <tbody> element can contain <row> elements with <entry> elements inside them.

9. Inside the <tbody> element, add a <row> element and add content to it with <entry> elements and their containing text as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
  ...
  <tbody>
  <row>
  <entry><p>7 weeks</p></entry>
  <entry><p>Attempt flight for the first time</p></entry>
  </row>
  ...
</conbody>
</concept>

```

10. After the <row> element you just added, add two more rows and add content to them as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_images_tables">
  <title>Duckling growth and development</title>
  <conbody>
  ...
  </row>
  <row>
  <entry><p>12-14 weeks</p></entry>
  <entry><p>Reach adult body weight</p></entry>
  </row>
  <row>
  <entry><p>1 year</p></entry>

```

```

<entry><p>Capable of reproduction</p></entry>
</row>
...
</concept>

```

The <tbody> element can contain one or more <row> elements.

Video: [Creating a table in DITA](#)

11. Check your file lesson2/l_concept_images_tables_start.dita against the sample file lesson2/l_concept_images_tables.dita.

Practice

1. Open the file lesson2/l_concept_images_tables_exercise_start.dita and use it to convert the following content from [Content Strategy 101](#) into DITA:

Use the sample image lesson2/images/configurebetter1.png for this exercise.

Supporting marketing with technical content

Tech comm and marcom have long occupied opposite ends of the content spectrum. The stereotype is that tech comm is text-heavy, dense, and badly formatted whereas marcom is shiny, beautiful, and content-free. From there, the debate just intensifies: *Marcom versus tech comm: the stereotypes*

	Marcom	Tech comm
Design or automation?	Design	Automation
How much detail?	As little as possible	As much as possible
Assumed impact on revenue	A lot	None
Primary purpose	Persuade people to buy	Inform people
Give people a friendly interface that lets them quickly narrow down their options and choose the one they want. You don't have to expose all of the fields in the database—just the ones that help people narrow down their choices.		

The product list on the right updates as you make selections on the left

Product	Color	Price
TurboWidget	orang	\$34
SlimeWidget	green	\$12
HipsterWidget	violet	\$132
ArrestMeWidge	red	\$59

2. Check your file `lesson2/l_concept_images_tables_start.dita` against the sample file `lesson2/l_concept_images_tables_exercise.dita`.

Lesson 3: More Elements

Objectives

- Learn about more elements and identify best practices for adding them to a concept topic.
- Add the ``, `<i>`, `<u>`, `<term>`, `<cite>`, `<varname>`, `<sub>`, `<sup>`, `<dl>`, `<menucascade>`, and `<fn>` elements to the concept.

You may need more specific elements in your concept topic than basic body elements, images, and tables. Maybe you need to indicate that certain words or phrases are important, or you need a more sophisticated way to define terms or provide references.

Some frequently used elements in DITA for inline styling include the ``, `<i>`, `<u>`, `<sub>`, and `<sup>` elements. DITA also offers the `<term>`, `<cite>`, and `<varname>` elements as alternatives for inline styling that have more semantic value, or provide more information about the content.

Other useful elements include `<fn>` for inserting footnotes, `<menucascade>` for showing the order of software menu choices, and `<dl>` for creating lists of terms and their definitions.

This lesson introduces these elements and shows how to add them to a concept topic. Like the other elements introduced in this course, these elements are not exclusive to the concept topic type and can also be used in the other DITA topic types (task and reference).

Note:

This lesson covers basic use of these elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Phrases](#)

Adding inline styling

DITA uses several elements for inline styling, or styling of pieces of text inside a paragraph:

<code></code>	bold text
<code><i></code>	italic text
<code><u></code>	underlined text
<code><term></code>	a word or phrase that needs a definition
<code><cite></code>	a word or phrase that needs a citation
<code><varname></code>	a word or phrase that may change based on the user's circumstances
<code><sub></code>	subscript text
<code><sup></code>	superscript text

The inline elements ``, `<i>`, and `<u>` are familiar and easy to use. However, because they are closely linked with the appearance of text, it is easy to misuse them and defeat the purpose of separating content from formatting. Use these elements sparingly; in most cases, it is better to use more meaningful elements instead, such as the `<term>`, `<cite>`, or `<varname>` elements.

In the following examples, you will learn how to insert the simple ``, `<i>`, and `<u>` elements first, then how to replace them with the semantically rich alternatives.

Practice

1. Make a copy of the file `lesson3/l_concept_elements_start.dita` and open it in your editor.

Note:

If you are using a DITA-aware text editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
<title>Feeding ducklings</title>
<conbody>
</conbody>
</concept>

```

2. Add a `<p>` element containing ``, `<i>`, and `<u>` elements as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
<title>Feeding ducklings</title>
<conbody>
<p>Ducklings need <b>round-the-clock access</b> to both <i>food</i> and <i>water</i>
during the first <u>two weeks</u> of their lives.</p>
</conbody>
</concept>

```

By adding the `` element to this paragraph, you have indicated that the words “round-the-clock access” should be rendered in boldface. Similarly, by adding the `<i>` element, you have indicated that the words “food” and “water” should be rendered in italic, and by adding the `<u>` element, you have indicated that the words “two weeks” should be rendered with an underline.

Although these inline elements add emphasis to important words, they have no other semantic value. Suppose you wanted to define the words “round-the-clock access” later, or to style them the same way as other words describing feeding frequency. Rather than using the `` element to tag the words simply as bold, you could use the `<term>` element to tag them as an important term.

There are similar, more semantically rich alternatives for the `<i>` and `<u>` elements, as you will see in the next step.

3. After the `<p>` element, add another `<p>` element containing the same text as the previous paragraph, with the `` element changed to `<term>`, the `<i>` element changed to `<cite>`, and the `<u>` element changed to `<varname>`, as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
<title>Feeding ducklings</title>
<conbody>
...
<p>Ducklings need <term>round-the-clock access</term> to both <cite>food</cite> a
nd
<cite>water</cite> during the first <varname>two weeks</varname> of their lives.</p>

```

```
</conbody>
</concept>
```

In this example, you have tagged “round-the-clock access” with the `<term>` element to show that it is a term that should be defined later. You have tagged “food” and “water” with the `<cite>` element to show that they are words that could be given bibliographic citations should you need them. Finally, you have tagged “two weeks” with the `<varname>` element to show that it may need to be substituted with other information based on the user’s circumstances (for example, certain breeds of ducks may need round-the-clock access to food and water longer than two weeks).

4. After the last `<p>` element, add another `<p>` element containing a `<sub>` element as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
<title>Feeding ducklings</title>
<conbody>
...
<p>Because duck feed has a dry, gritty texture, ducklings need plenty of
H<sub>2</sub>O to keep their bills clean.</p>
</conbody>
</concept>
```

The `<sub>` element is used to tag subscript text, or text that appears smaller and slightly below the main line. In the example you added, the `<sub>` element is used to ensure that the “2” in H₂O is displayed as subscript.

5. After the last `<p>` element, add another `<p>` element containing a `<sup>` element as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
<title>Feeding ducklings</title>
<conbody>
...
<p>You can make your own low, no-spill food and water dishes out of recycled
plastic containers, or buy ready-made dishes from E-Z-Feed<sup>2</sup>.</p>
</conbody>
</concept>
```

The `<sup>` element is used to tag superscript text, or text that appears smaller and slightly above the main line. In the example you added, the `<sup>` element is used to indicate that the “2” (symbol for “squared”) in the name of the duck food company (“E-Z-Feed²”) should be displayed as superscript.

Note: Do not use `<sup>` elements to create raised numbers for footnotes. Instead, use the DITA `<fn>` (footnote) element, which we will cover later in this lesson.

Adding other elements

DITA uses many other important elements. Here are three more:

<code><fn></code>	The element that inserts a footnote.
<code><menucascade></code>	The element that indicates the order of a menu path, such as File > Save As. The <code><menucascade></code> element must contain one or more <code><uicontrol></code> elements. Each <code><uicontrol></code> element contains the text for a menu item.
<code><dl></code>	The element that contains a definition list. A definition list is a list of terms and their corresponding definitions, presented in a format similar to a two-column table by default. The definition list contains one or more entries, tagged with the <code><dlentry></code> element, and each entry contains a term (tagged with the <code><dt></code> element) and one or more definitions (tagged with the <code><dd></code> element).

Practice

1. Continue using the file `lesson3/1_concept_elements_start.dita` to add each of these elements to your example file.
2. After the last `<p>` element, add a new `<p>` element with the `<fn>` element inside it as shown in the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
  <title>Feeding ducklings</title>
  <conbody>
    ...
    <p>Ducklings need higher levels of protein in their diets than adult ducks.<fn>A
protein level of 18-20% is recommended for newborn ducklings.</fn></p>
  </conbody>
</concept>
```

The `<fn>` element indicates where the footnote reference number will appear inside the paragraph. By default, the text inside the `<fn>` element is displayed at the bottom of the page (for PDF) or end of the topic (for HTML) when you generate output from the file.

3. After the last `<p>` element, add a new `<p>` element with a `<menucascade>` element inside it, as shown in the following example:

Video: [Creating a menucascade in DITA](#)


```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
  <title>Feeding ducklings</title>
  <conbody>
    ...
    <p>To create a spreadsheet to keep track of when you need to replenish your
    ducklings' food and water, go to
    <menucascade>
    </menucascade>.
  </p>
</conbody>
</concept>

```

In the example you added, the `<menucascade>` element will be used to show the menu options involved in creating a spreadsheet.

4. Inside the `<menucascade>` element, add a `<uicontrol>` element and add content to it as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
  <title>Feeding ducklings</title>
  <conbody>
    ...
    <menucascade>
    <uicontrol>File</uicontrol>
    </menucascade>
    ...
  </conbody>
</concept>

```

In the example you added, the `<uicontrol>` element contains the name of the first menu option involved in creating a spreadsheet: “File.”

5. After the `<uicontrol>` element, add two more `<uicontrol>` elements and add content to them as shown in the following example.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
  <title>Feeding ducklings</title>
  <conbody>
    ...
    <menucascade>
    <uicontrol>File</uicontrol>
    <uicontrol>New</uicontrol>
    <uicontrol>Spreadsheet</uicontrol>

```

```

</menucascade>
...
</conbody>
</concept>

```

By default, an arrow symbol is inserted between <uicontrol> elements in the output to indicate the hierarchy of the menu options.

With the <menucascade> and <uicontrol> elements in the example you added, the user now knows to click on “File,” then “New,” then “Spreadsheet.”

You can use the <uicontrol> element without the <menucascade> element. For example, you can tag a word with the <uicontrol> element to indicate a button the user should click. Because the <uicontrol> element indicates that the text it surrounds will have special styling, avoid using the element or other inline styling elements inside the <uicontrol> element.

6. After the <menucascade> element, add a new introductory <p> element followed by a <dl> element as shown in the following example:

Video: [Creating a definition list in DITA](#)

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
  <title>Feeding ducklings</title>
  <conbody>
    ...
  </menucascade>
  <p>Domestic ducks fall into categories based on weight. These weight classes may help you choose a species based on how much you will need to feed your ducks.</p>
  <dl>
  </dl>
</conbody>
</concept>

```

The <dl> element sets up the framework for a definition list.

7. Inside the <dl> element, add a <dlentry> element as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
  <title>Feeding ducklings</title>
  <conbody>
    ...
  <dl>
    <dlentry>
  </dlentry>
  ...

```

```

</conbody>
</concept>

```

Each <dlentry> element contains a term and its definition. The <dlentry> element can contain one or more <dd> elements.

8. Inside the <dlentry> element, add the <dt> and <dd> elements and add content to them as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
  <title>Feeding ducklings</title>
  <conbody>
    ...
    <dlentry>
      <dt>Bantam</dt>
      <dd>The lightest-weight ducks and best fliers, such as the Mallard duck.<
/ dd>
    ...
  </conbody>
</concept>

```

The <dt> element contains the term, and the <dd> element contains its definition.

9. After the <dlentry> element, add three more <dlentry> elements and add content to them with <dt> and <dd> elements as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_elements">
  <title>Feeding ducklings</title>
  <conbody>
    ...
    </dlentry>
    <dlentry>
      <dt>Light</dt>
      <dd>The second lightest-weight ducks and best egg-layers, such as the Kha
ki
        Campbell duck.</dd>
    </dlentry>
    <dlentry>
      <dt>Medium</dt>
      <dd>The mid-weight and generally calmest ducks, such as the Swedish duck.
</dd>
    </dlentry>
    <dlentry>
      <dt>Heavy</dt>
      <dd>The heaviest and generally friendliest ducks, such as the Pekin duck.

```

```
</dd>
  </dentry>
...
</conbody>
</concept>
```

10. Check your file `lesson3/l_concept_elements_start.dita` against the sample file `lesson3/l_concept_elements.dita`.

Practice

1. Open the file `lesson3/l_concept_elements_exercise_start.dita` and use it to convert the following content from [Content Strategy 101](#) into DITA:

Increasing product visibility

Technical content can help organizations increase the visibility of their products in the marketplace. Officially, technical content is intended for **product customers**—people who buy a product and then look at the documentation.

But one opinion poll indicates that about one-third of buyers¹ look at the documentation before buying a product, and the quality of the documentation will affect their purchasing decision.

To reel in new prospects, your content must perform in three different ways: be searchable, findable, and discoverable.

Searchable

Information must be available via an internet search.

Findable

Information must perform well for relevant keywords.

Discoverable

Information must increase the likelihood that people will link to your information.

To read more, go to **Contents > Business goals > Marketing and product visibility > Increasing product visibility**.

¹ “*Consumer Feelings about Product Documentation*,” an opinion poll conducted online by Sharon Burton

2. Check your file `lesson3/l_concept_elements_exercise_start.dita` against the sample file `lesson3/l_concept_elements_exercise.dita`.

Lesson 4: Advanced elements

Objectives

- Learn about advanced elements and identify best practices for adding them to a concept topic.

- Add the <codeblock>, <codeph>, <lq>, <section>, <draft-comment>, and <required-cleanup> elements to the concept.

Now that you’ve mastered using the basic DITA elements and other commonly used ones, you’re ready to start using some more advanced elements in your concept topic.

You can use the <codeblock> and <codeph> elements to insert snippets of code into the body of your topic. With the <lq> element, you can insert a quote and create a link to its original source.

If you need to break your topic into subsections, you can add one or more <section> elements. The <section> element comes with two constraints: you can only use the <section> element inside of the <conbody> element, not inside of other <section> elements, and you can only follow a <section> element with another <section> element, an <example> element, or a <conbodydiv> element.

You can collaborate with other authors using the <draft-comment> element to make notes on pieces of information, or you can use the <required-cleanup> element to show that an element needs to be tagged correctly.

This lesson introduces these advanced elements and shows how to add them to a concept topic. Like the other elements introduced in this course, these elements are not exclusive to the concept topic type and can also be used in the other DITA topic types (task and reference).

Note:

This lesson covers basic use of these elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Element domains](#)

Adding advanced elements

Here are some common advanced elements in DITA:

<codeblock>	The container element for a piece of code. Typically, it is rendered in a monospace font in PDF or HTML output. Placing content inside the <codeblock> element gives you control over line breaks.
<codeph> (code phrase)	The container element that displays a single word or phrase inside a <p> element as code. For example, if you mention an element in a paragraph and want to display only that element’s name in a monospace font, the <codeph> element is useful.
<lq> (long quote)	The container element for including a quote from an outside source in your content rather than (or in addition to) linking to the source.

Practice

1. Make a copy of the file lesson4/l_concept_advanced_start.dita and open it in your editor.

Note:

If you are using a DITA-aware text editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_advanced">
<title>Writing about ducks</title>
<conbody>
</conbody>
</concept>
```

Video: [Creating a codeblock in DITA](#)

2. Inside the <conbody> element, add the <codeblock> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_advanced">
<title>Writing about ducks</title>
<conbody>
<codeblock>
duckdata> add entry (mallard)
;
1 entry added (0.05 sec)
duckdata> _
</codeblock>
</conbody>
</concept>
```

The <codeblock> element isolates the piece of code content so that any tags in it exist for display purposes only and do not affect the DITA tags in your concept topic file.

The <codeblock> element also allows you to control line breaks. In the example you added, the text inside the <codeblock> element shows an entry being added to a database using the command line.

Note:

If you need an opening tag in a <codeblock> element, type < in place of the opening tag character (<). If you are working in the author or visual mode of a DITA editor, you can type the opening tag character inside a <codeblock> element, and the <codeblock> element will automatically render it as < in the text.

3. After the `<codeblock>` element, add a `<p>` element with a `<codeph>` element inside it and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_advanced">
<title>Writing about ducks</title>
<conbody>
...
</codeblock>
<p>To add a duck species to your database, type <codeph>add entry</codeph> followed by
the name of the duck in parentheses on the command line and press Enter.</p>
</conbody>
</concept>
```

In the example you added, the `<codeph>` element allows you to isolate the words “add entry” and indicate that they should be displayed in a monospace font to show that they are a command.

4. After the `<p>` element, add a `<lq>` element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_advanced">
<title>Writing about ducks</title>
<conbody>
...
</p>
<lq href="http://classiclit.about.com/library/bl-etexts/hdthoreau/bl-hdtho-wald-9.htm"
format="html" scope="external">How much fairer than the pool before the farmers door,
in which his ducks swim! Hither the clean wild ducks come. Nature has no human
inhabitant who appreciates her. The birds with their plumage and their notes are
in
harmony with the flowers, but what youth or maiden conspires with the wild luxuriant
beauty of Nature?</lq>
</conbody>
</concept>
```

In the example you added, the `<lq>` element allows you to include a quote from *Walden* by Henry David Thoreau.

The `<lq>` element can contain a link to the quote’s source. In the example you added, the `<lq>` element contains an `href` attribute pointing to the source URL, a `format` attribute indicating that the source is HTML, and a `scope` attribute showing that the source is external.

Adding more advanced elements

Here are some more common advanced elements in DITA:

<code><section></code>	<p>The element that divides the body of a topic into subsections with individual titles. The <code><section></code> element can contain a <code><title></code> element to give it a heading, along with most of the same elements allowed inside the <code><conbody></code> element. However, the <code><section></code> element cannot contain other <code><section></code> elements and can only be followed by another <code><section></code> element, an <code><example></code> element, or a <code><conbodydiv></code> element.</p> <p>Note:</p> <p>If you find that you are adding many sections or sections of substantial length to a topic, make each of those sections into its own topic instead so that you can reuse the information more easily.</p>
<code><draft-comment></code>	<p>The element that lets you insert comments and questions into the content while it is being developed. By default, the <code><draft-comment></code> element is hidden in the final output, so you can render output without deleting all draft comments. (You have to decide whether you're willing to risk it, though!)</p>
<code><required-cleanup></code>	<p>The element that lets you wrap content that is tagged incorrectly and needs to be fixed. By default, any content inside the <code><required-cleanup></code> element is hidden in output.</p>

Practice

1. Continue using the file `lesson4/l_concept_advanced_start.dita` to add each of these elements to your example file.

Video: [Adding a section in DITA](#)

2. After the `<lq>` element, add a `<section>` element as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_advanced">
  <title>Writing about ducks</title>
  <conbody>
    ...
  </lq>
  <section>
  </section>
```



```
</conbody>
</concept>
```

In the example you added, you created a subsection of the “Writing about ducks” topic.

3. Inside the `<section>` element, add `<title>` and `<p>` elements and add content to them as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_advanced">
<title>Writing about ducks</title>
<conbody>
...
<section>
<title>Sharing your duck database</title>
<p>Export your duck database as HTML output to make it easy to share on your webs
ite.
You can also offer your database to users as a download.</p>
</section>
</conbody>
</concept>
```

Only one `<title>` element is allowed inside the `<concept>` element where a title already exists. However, by adding a `<section>` element, you are able to have multiple subtitles in your concept topic. Like the `<concept>` element, each `<section>` element can only contain one `<title>` element.

A `<section>` element can contain all the same body elements as a `<conbody>`, with the exception of another `<section>` (that is, you cannot nest `<section>` elements inside each other).

4. After the `<p>` element in the new section, add a `<draft-comment>` element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_advanced">
<title>Writing about ducks</title>
<conbody>
...
</p>
<draft-comment>Are you sure you want to offer a download?</draft-comment>
...
</conbody>
</concept>
```

In the example you added, you are telling other authors who may work on this topic to rethink the decision to offer a download.

5. At the end of the <section> element content (before the </section> close tag), add a <required-cleanup> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE concept PUBLIC "-//OASIS//DTD DITA Concept//EN" "concept.dtd">
<concept id="concept_advanced">
<title>Writing about ducks</title>
<conbody>
...
<required-cleanup>
<p>Consider giving users the option to send feedback or suggested entries for you
r
duck database.</p>
</required-cleanup>
</section>
</conbody>
</concept>
```

In this example, you have surrounded the <p> element with the <required-cleanup> element to indicate that someone must move it to a valid position (such as inside the <section> element) or re-tag it with a valid element (such as the <example> element). Otherwise, you will not be able to produce output because of the presence of invalid elements.

6. Check your file lesson4/l_concept_advanced_start.dita against the sample file l_concept_advanced.dita.

Practice

1. Open the file lesson4/l_concept_advanced_exercise_start.dita and use it to convert the following content from [Content Strategy 101](#) into DITA:

Developing a technical content strategy

After reviewing the existing information products, you should have a list of content challenges and ideas for improvement. Here are some common scenarios.

Reuse between technical documentation and training materials

The training department uses reference and task information created by the technical documentation team, but the instructional designers copy and paste instead of linking because the two groups use different, incompatible content creation tools.

Comment: Add: “If the two teams standardized on a single workflow, they could share content seamlessly and avoid lots of tedious reworking.”

HTML output is needed in addition to PDF

Your content may look like this in HTML:

```
<div class="p">You may produce high-value content, such as the following:
<ul class="ul">
<li class="li"><p class="p">Training materials</p></li>
<li class="li"><p class="p">White papers</p></li>
<li class="li"><p class="p">Knowledge base articles</p></li>
</ul>
</div>
```

The `<div>` contains the unordered list.

Technical content is outdated because of inefficient updating process

The book production process—generating tables of contents and indexes, checking pagination, creating change page logs, and similar tasks—takes a significant amount of time. As a result, books are only updated twice a year. But the product is changing quarterly or even monthly, so the technical documentation is almost always out of date. Readers are complaining about the lack of synchronization between the product updates and the content updates.

2. Check your file `lesson4/l_concept_advanced_exercise_start.dita` against the sample file `lesson4/l_concept_advanced_exercise.dita`.

Lesson 5: XML overview and best practices

Objectives

- Identify best practices for authoring concept topics.
- Show examples of best practices in a concept topic.

Additional reading

[DITA Style Guide, The separation of content and form](#)

[DITA Style Guide, Content re-use](#)

[DITA Style Guide, Language and punctuation](#)

[Concise Writing, contributed by Pam Noreault, Tracey Chalmers, and Julie Landman](#)

Separate content from formatting

One of the major benefits of creating structured content in DITA is the separation of content from formatting. Content that is tagged according to its structure rather than its appearance will be more flexible in its output capabilities.

For example, an unstructured piece of content that has been created with page layout in mind will work best in print or PDF output, and may work poorly or not at all in an HTML-based output. However, a

structured DITA topic with no formatting specified in the content should work equally well in multiple output types.

For authors who are accustomed to a formatting-based environment such as a desktop publishing program, it can be tempting to misuse the DITA tags to try to control the look and feel of the content. Using tags for a purpose other than they were intended decreases the value of those tags. Tag abuse can also have unintended consequences—a misused tag may achieve the desired formatting effect in one output type, but destroy the look and feel of the content in another output type.

One form of tag abuse is the forced line break. Do not use the `<p>` element to control where your lines of text should break, as this will cause problems in responsive output types such as HTML. Use the `<codeblock>` element to control line breaks for certain parts of your content (for example, code samples).

Be consistent

Because DITA content exists separately from formatting, consistent use of the DITA structure and tags is key to ensuring that all of your output types will look the way you expect.

One negative side effect of inconsistent tagging is mixed content. Suppose you have the following unordered list:

```
<ul>
<li>text</li>
<li>text</li>
<li>text
<p>text</p>
</li>
</ul>
```

In this example, the last list item contains two paragraphs. Because the `<p>` element is valid inside the `` element, it is possible to tag each paragraph separately. However, tagging this content as shown in the example means that you now have mixed content—the last `` element has one line of text tagged simply with the `` element, and another line with `<p>` tags around it. This could affect the styling of the text of this list item in the final output.

To avoid having mixed content, surround all text inside an `` element with `<p>` tags, whether you are tagging one paragraph or more:

```
<ul>
<li><p>text</p></li>
<li><p>text</p></li>
<li>
<p>text</p>
<p>text</p>
</li>
</ul>
```

In addition to the element, we recommend that text inside the <note>, <entry>, and <stentry> elements be surrounded with <p> tags in case there are multiple paragraphs. To help ensure consistency and avoid mixed content, use the <p> tag for any text that does not need to be tagged in a specific way.

Create semantically rich content

One of DITA's advantages is the ability to create semantically rich content, or content in which the element tags contain information about the content.

For example, using the <term> element may simply cause a word to display as bold text in the output, but the tag conveys much more information than just “render as boldface”—it shows that this word is a term that needs to be defined.

To take advantage of the semantically rich aspect of DITA, tag each piece of content with the element that most closely reflects what the content is and how it should be used. For example, use elements such as <term> or <cite> to highlight important words rather than simply using or <i>.

Proper use of topic types also makes your content more semantically rich. For example, write step-by-step procedures in a task topic instead of a concept topic. That way, you can use the structure that DITA provides for creating steps, which provides much better information about the steps than using the element in a concept topic.

Course III. The DITA Task Topic Type

Lesson 1: Creating a task topic

Objectives

- Create a task topic file
- Differentiate strict and general tasks
- Introduce the DOCTYPE for strict task topics
- Add the root elements for a task
- Understand the purpose of the <prereq> and <context> elements

This lesson shows how to create a new task topic and how to add the first common elements found in a task topic.

Note: This lesson covers basic use of the task elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Task](#)

[DITA Style Guide, General Task](#)

[DITA Style Guide, The prereq element](#)

[DITA Advanced Topics, contributed by Pam Noreault, Tracey Chalmers, and Julie Landman](#)

Housekeeping and sample files

Download [task_samples.zip](#) now. It contains the sample files for the entire Authoring DITA tasks course. Extract the contents and put them in a directory that you can access easily.

Inside the samples folder, you will find the file `l_task_start.dita`, which you will use to complete exercises throughout the course. You will also find the following folders corresponding to lessons in this course: `lesson1`, `lesson2`, and `lesson3`.

In each lesson folder, you will find three DITA files:

- `l_filename.dita`: The completed version of the sample file you can use to check your work during each lesson.
- `l_filename_exercise_start.dita`: The empty file you will add content to as you complete the exercise at the end of each lesson.

- `l_filename_exercise.dita`: The completed version of the exercise file you can use to check your work.

Each lesson will instruct you on which files to use for the exercises. Save your file as you complete each step to avoid losing your work.

Create a local copy of each file to work in as you complete the lessons. That way, if you reach a point where your working file doesn't match the examples, or is broken for any reason, you can make a fresh copy and resume your work or start over.

In the instructions and examples, we show you the DITA code for each sample file. Most DITA editors have auto-complete or other similar features to guide you through the process of adding elements (for example, if you type the opening tag of an element, most DITA editors will automatically add the closing tag for you). Therefore, you will probably not need to create every piece of code from scratch as you work. Our demo videos were created in [oXygen XML Editor](#) and show the differences between working in author view, which presents the DITA content in a user-friendly visual format, and working in text view, which shows the DITA code.

Strict or general tasks

The DITA specification defines two types of task topics: a strict task and a general task.

The main elements in the body of a strict task must occur in a specific order. The strict task can contain only one set of steps. Also, the steps the reader performs must use specific elements (`<steps>` and `<step>`).

The element order in the general task is not as rigid as in the strict task. The general task allows one or more sets of steps. The elements used for the steps can be almost any element that is allowed in a DITA `<section>` element.

Although it may sound as though it's easier to write a general task, there are a number of good reasons for using the strict task. These include:

- Consistency. All the tasks in your document or documents will have the same organization.
- Dependability. Because the organization is consistent, readers can count on finding the similar information in the same place.
- Ease of authoring. DITA editors can direct writers to the next element they need to add.
- Simplicity. Most tasks only require a numbered set of steps.
- Semantics. Using the `<steps>` and `<step>` elements in the strict task has high semantic value.

This course focuses on the strict task, with the expectation that once you're familiar with the elements in a strict task, it's easy to apply that knowledge to the general task.

For more information about the general task, see the [DITA Specification](#), and the [<steps-informal>](#) element.

Creating a new task topic

At a minimum, the task topic must contain a <task> root element (with an id attribute) that contains a <title> element.

Following the <title> element and an optional <shortdesc> element, a <taskbody> element contains the task content. The elements within the <taskbody> conform to a specific order. This course presents the elements of the <taskbody> in the order in which they must occur.

[Video: Creating a DITA task topic](#)

Practice

1. Make a copy of the file l_task_start.dita and open it in your editor.

Note: If you are using a DITA-aware text editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "task.dtd">
<task id="my_first_task">
  <title>xyz</title>
</task>
```

The first line (which begins with <?xml>) is an XML declaration, which is a standard part of all XML files.

The second line is the DOCTYPE declaration, which tells DITA editors or DITA output generators that this is a DITA strict task topic. The programs then use that information when validating the content of the topic. The DOCTYPE will be different for each topic type that you create (including the general task).

The third line contains the opening tag of the <task> element.

The fourth line contains the title element.

2. Inside the <title> element, update the text of the task title.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "task.dtd">
<task id="my_first_task">
  <title>Watching wild ducks</title>
</task>
```

3. After the <title> element, add a <taskbody> element.

The <taskbody> element contains all the actual content in the task.


```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "task.dtd">
<task id="my_first_task">
  <title>Watching wild ducks</title>
  <taskbody>
  </taskbody>
</task>
```

This is the essential part of a DITA strict task topic. The rest of this lesson shows you how to add the first two child elements to the <taskbody> element.

Adding introductory elements to a task topic

The <prereq> (prerequisite) and <context> elements are optional elements that introduce the instructional content in a task topic.

The <prereq> element describes what the reader needs before performing the task, which can include:

- Things the reader must have, such as binoculars or a smartphone with a bird identification app
- Information the reader must have, such as knowledge of migratory patterns
- Previous tasks the reader must have accomplished, such as completing a basic duck identification course

The <context> element provides background information about the task, such as the purpose of the task and what the reader will gain. The <context> element content should be relatively brief; if you need to provide more detailed background information, consider creating a concept topic.

Although <prereq> and <context> are optional, they occur in order—the <prereq> element must go before the <context> element.

[Video: Strict sequence of introductory elements in a DITA Task](#)

Practice

1. Continue working in the file l_task_start.dita.
2. Inside the <taskbody> element, add a <prereq> element.

```
<taskbody>
  <prereq>
    <p>Before you go watching wild ducks, make sure you know what a wild duck looks like.</p>
  </prereq>
</taskbody>
```

3. After the <prereq> element, add a <context> element.

```

<taskbody>
  ...
  </prereq>
  <context>
    <p>Watching wild ducks can be calming and recreational. </p>
    <p>Read this to learn more about what you need to have a satisfying duck-
watching
    experience.</p>
  </context>
</taskbody>

```

Your task now has a title, a set of prerequisites, and a context. The next lesson covers how to add steps to the task.

Practice

1. Open the file `lesson1/l_new_task_exercise_start.dita` and use it to convert the following content from [Content Strategy 101](#) into DITA:

Creating a business case for your content strategy

Before you begin: Thoroughly assess your content and content development process so that you can understand the gap between the strategy you need and the strategy currently in place.

About this task: These instructions will show you how to develop a business case that you can use to show that your company needs a new content strategy.

You may find that your company could save money on content production in the following areas:

- Automation
- Reuse
- Localization

2. Check your file `lesson1/l_new_task_exercise_start.dita` against the sample file `lesson1/l_new_task_exercise.dita`.

Lesson 2: Creating the steps

Objectives

- Specify the location of the `<steps>` element in the `<taskbody>`
- Identify the core elements found in a step and how they are used
- Differentiate the `<choices>` and `<choicetable>` elements
- Understand how to create substeps and how they differ from steps

The <steps> element (note plural) contains <step> elements (singular). The individual step instructions are inside the <step> elements.

When output is generated from the task topic, the steps usually take the form of a numbered list. The labels that identify each step (such as a number and some punctuation or the word “Step” followed by a number) are determined by the output generator. When you’re authoring a step, most DITA editors will display step numbers beside each step.

Note:

This lesson covers basic use of these elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Semantics in steps](#)

[DITA Style Guide, The command element](#)

[DITA Style Guide, Required and optional steps](#)

[DITA Style Guide, Choice tables](#)

[DITA Style Guide, Substeps within steps](#)

[DITA Style Guide, Notes and extra information in a step](#)

First steps

The basic elements involved in creating a set of steps include the <steps>, <step>, <cmd>, and <info> elements.

<steps>	A series of step-by-step instructions. The <steps> element can contain one or more <step> elements.
<step>	A single step. The step must contain one <cmd> element. The <step> element can also contain the <info> element, and additional elements that provide information about the step (these are described later in this course).
<cmd>	A command. The text in the <cmd> element should be in the active voice and should be a single sentence.
<info>	Additional information about the step.

[Video: Adding steps to a DITA task](#)

Practice

1. Continue working in the file l_task_start.dita.

2. After the closing tag of the `<context>` element, add a `<steps>` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "task.dtd">
<task id="my_first_task">
  ...
  </context>
  <steps>
  </steps>
</taskbody>
</task>
```

As with the `<prereq>` and `<context>` elements, the `<taskbody>` can contain only one `<steps>` element. Restricting you to a single `<steps>` element helps ensure that the topic only describes how to perform one task.

3. Inside the `<steps>` element, add a `<step>` element.

```
...
  <steps>
    <step>
    </step>
  </steps>
...
```

The `<steps>` element must contain at least one `<step>` element. Most DITA editors automatically add the first `<step>` element inside the `<steps>` element (and usually a `<cmd>` element nested in the `<step>`).

In DITA, the content of each individual step is also highly controlled; each step has a specific set of elements, which must occur in a specific order. The action the reader is to perform is contained in a `<cmd>` element (read as “command”), which is a required element in `<step>`. In most cases, `<cmd>` contains a single sentence, which is a direct instruction to the reader to perform an action.

4. Inside the `<step>` element, add a `<cmd>` element with the text shown here.

```
...
  <steps>
    <step>
      <cmd>Choose a location to watch ducks.</cmd>
    </step>
  </steps>
...
```

As a best practice, the `<cmd>` element should be limited to a single sentence, in active voice, that directs the reader to perform an action. To provide additional information about the `<cmd>` element, use the `<info>` element.

To provide additional information about the step direction contained in the `<cmd>` element, use the `<info>` element. You can use any number of `<info>` elements after the `<cmd>` element, and before the `<stepresult>` element.

5. After the `<cmd>` element, add an `<info>` element, with the text shown here.

```
...
    <steps>
      <step>
        <cmd>Choose a location to watch ducks.</cmd>
        <info><p>Generally, ducks are found in wild locations, near water.</p></info>
      </step>
    </steps>
...
```

The `<info>` element can contain text, notes, images, tables, and so on. As with other elements that can contain mixed content, best practices dictate that you should use a `<p>` element to wrap text, even if the `<info>` element only contains one paragraph.

The remaining topics in this lesson introduce additional elements you can use to document the step.

Giving the reader choices

As part of giving a reader an instruction, there may be more than one option for the reader. For example, when using an auto stereo system, the reader has the choice of which mode to use (AM, FM, CD, line-in, and so on).

To present this information, and to give it a semantically appropriate label, you can use the `<choices>` or `<choicetable>` elements.

- Use `<choices>` when the choices only require a brief description.
- Use `<choicetable>` (detailed later in this course) when you need to provide specific keywords, along with an extended description of each.

The `<choices>` element is usually presented as a bulleted list. The structure of the `<choices>` element is like a `` element:

<code></code> element	<code><choices></code> element
<pre> one thing another thing </pre>	<pre><choices> <choice>one thing</choice> <choice>another thing</choice> </choices></pre>

The <choices> element contains one or more <choice> elements. Each <choice> element presents one of the options for the reader. The content of the <choice> element is any element that can be used inside an element.

Practice

1. Continue working in the file l_task_start.dita.
2. After the closing tag of the first <step> element, add a new <step> element. Inside the <step>, create a <cmd> element (if one isn't created for you). Add the content shown here to the <cmd> element:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "task.dtd">
<task id="my_first_task">
  ...
  </step>
  <step>
    <cmd>Pick a field guide to identify the ducks.</cmd>
  </step>
</steps>
</taskbody>
</task>
```

3. After the new <cmd> element, add a <choices> element.

```
<step>
  <cmd>Pick a field guide to identify the ducks.</cmd>
  <choices>
  </choices>
</step>
```

4. If your editor does not automatically add a first <choice> element inside the <choices> element, add one now.

```
<step>
  <cmd>Pick a field guide to identify the ducks.</cmd>
  <choices>
    <choice>Many modern duck watchers rely on the Sibley guide</choice>
  </choices>
</step>
```

By default, most DITA output generators render a <choices> element as a bulleted list.

5. Add an additional <choice> element.

```
<step>
  <cmd>Pick a field guide to identify the ducks.</cmd>
  <choices>
    <choice>Many modern duck watchers rely on the Sibley guide</choice>
```

```

        <choice>A traditional favorite is the Peterson guide</choice>
    </choices>
</step>

```

A table of choices

The choice table is a two-column table. In each row, the first column contains an option (perhaps a keyword, a command line parameter, or an argument value). The second column in the row contains a detailed description of the option. The detailed description can include multiple paragraphs, lists, or even images.

The elements contained in the `<choicetable>` element include:

<code><chhead></code>	The header row of a <code><choicetable></code> element. The <code><chhead></code> element contains the <code><choptionhd></code> and <code><chdeschd></code> elements and is optional. There can only be one <code><chhead></code> element in a <code><choicetable></code> element.
<code><choptionhd></code>	Header text for the option column in a choice table. The text should be wrapped in a <code><p></code> element according to best practice.
<code><chdeschd></code>	Header text for the description column in a choice table. The text should be wrapped in a <code><p></code> element according to best practice.
<code><chrow></code>	A body row in a choice table. The <code><chrow></code> element contains the <code><choption></code> and <code><chdesc></code> elements. There can be one or more <code><chrow></code> elements in a <code><choicetable></code> element.
<code><choption></code>	Option text for a row in a choice table. The text should be wrapped in a <code><p></code> element according to best practice.
<code><chdesc></code>	Description text for a row in a choice table. The text should be wrapped in a <code><p></code> element according to best practice. There can be multiple elements in the <code><chdesc></code> element.

[Video: Creating a choicetable in a DITA task](#) shows how to use [oXygen XML Editor](#)'s choicetable wizard.

Practice

1. Continue working in the file `l_task_start.dita`.
2. Following the `<step>` element containing the `<choices>` element, create a new step with a `<cmd>` element and text as shown here:

```

<step>
  <cmd>Find some good optics</cmd>
</step>

```

3. After the <cmd> element, add a <choicetable> element and all its child elements:

```

<step>
  <cmd>Find some good optics</cmd>
  <choicetable>
    <chrow>
      <choption></choption>
      <chdesc></chdesc>
    </chrow>
    <chrow>
      <choption></choption>
      <chdesc></chdesc>
    </chrow>
  </choicetable>
</step>

```

For each row in the <choicetable>, you need a <chrow> element. The row contains two elements, which are both required: <choption> which contains the option text and <chdesc> which contains the description text. In standard usage, the <choption> contains just a few words (usually one), while the description can contain multiple sentences, or even paragraphs (<p> element).

The <choicetable> has an optional head (<chhead>), which allows you to label the two columns in the table.

4. Before the first row of the choice table, add a <chhead> element and its children, as shown here.

```

<step>
  <cmd>Find some good optics</cmd>
  <choicetable>
    <chhead>
      <choptionhd></choptionhd>
      <chdeschd></chdeschd>
    </chhead>
    <chrow>
      <choption></choption>
      <chdesc></chdesc>
    </chrow>
    ...
  </choicetable>
</step>

```

The <choptionhd> provides a heading for the option column; the <chdeschd> provides a heading for the description column.

5. Add content to the <chhead> element and its child elements, as shown here.


```

...
<chhead>
  <choptionhd>
    <p>Type</p>
  </choptionhd>
  <chdeschd>
    <p>Advantages</p>
  </chdeschd>
</chhead>
...

```

6. Add content to the <chrow> element and its child elements, as shown here.

```

...
<chrow>
  <choption>
    <p>Binoculars</p>
  </choption>
  <chdesc>
    <p>Wide field of view, allows quick observations</p>
  </chdesc>
</chrow>
...

```

7. After the closing tag of the <chrow> element, add another <chrow> element and add content to it as shown here.

```

<choicetable>
...
</chrow>
<chrow>
  <choption>
    <p>Spotting scope</p>
  </choption>
  <chdesc>
    <p>Higher magnification, better light-gathering</p>
  </chdesc>
</chrow>
</choicetable>

```

Getting more detailed

When creating a set of steps, you might need to provide a bit more detail about how to perform an action. In this case, you can add a <substeps> element to a <step>.

The <substeps> element contains one or more <substep> elements, which are similar in content to the <step> element. The <substep> element must contain a <cmd> element.

DITA allows only two levels of steps. The first level uses the <steps> element; the second level uses the <substeps> element. You cannot add a third level. If you find you need a third level of steps, reconsider the level of information in your procedure. The information might be better presented as a series of separate task topics. The same is true if you find you are providing a great deal of detail at the substep level.

[Video: Adding substeps in a DITA task](#)

Practice

1. Continue working in the file l_task_start.dita.
2. After the last <step> element, create a new <step> element, containing a <cmd> element and the text shown here.

```
<step><cmd>Go watch some birds.</cmd>
</step>
```

3. After the closing tag of the <cmd> element, add a <substeps> element.

```
<step>
  <cmd>Go watch some birds.</cmd>
  <substeps>
  </substeps>
</step>
```

You can have one or more <substeps> elements inside the <step> element. However, these <substeps> elements cannot be nested inside each other—they can only be added in sequence at the same level.

4. If your editor doesn't automatically add the <substep> element and nested <cmd> elements inside the <substeps> element, add them.

```
<step>
  <cmd>Go watch some birds.</cmd>
  <substeps>
    <substep>
      <cmd></cmd>
    </substep>
  </substeps>
</step>
```

5. Add content to the <cmd> element inside the <substep> element as shown here.

```
<step>
  <cmd>Go watch some birds.</cmd>
  <substeps>
    <substep>
```

```

        <cmd>Walk around.</cmd>
      </substep>
    </substeps>
  </step>

```

You can add the same optional elements inside a <substep> element that you can add inside a <step> element, such as the <info> element. However, you cannot add a <substeps> element inside a <substep> element.

6. After the closing tag of the <substep> element, add two more <substep> elements and add content to them as shown here.

```

<step>
  <cmd>Go watch some birds.</cmd>
  <substeps>
    <substep>
      <cmd>Walk around.</cmd>
    </substep>
    <substep>
      <cmd>When you see one, stop.</cmd>
    </substep>
    <substep>
      <cmd>Identify it.</cmd>
    </substep>
  </substeps>
</step>

```

Showing examples and results

A step can include an example of how the step is performed and some text about the result of having successfully completed the steps.

- Use <stepxmp> (step example) to show an example of how to perform the step. A step can contain any number of <stepxmp> elements.
- Use <stepresult> to describe the result of the step.

Note: The <stepxmp> element shows an example for the current step; to show an example of the entire task, use the <example> element, which is discussed in the next lesson.

[Video: Adding a step example in a DITA task](#)

Practice

1. Continue working in the file l_task_start.dita.
2. After the closing tag of the last <step> element, create a new <step> element, containing a <cmd> element and the text shown here.

```
...<step><cmd>When you have definitively identified a duck, make a note in your guide.</cmd>
</step>...
```

3. After the `<cmd>` element, add a `<stepxmp>` element.

```
...<step><cmd>When you have definitively identified a duck, make a note in your guide.</cmd>
  <stepxmp>
</step>...
```

The `<stepxmp>` element contains an example that illustrates or clarifies the step.

The `<stepxmp>` element can contain any of the elements that you can use inside an `` element. To show examples where line breaks and spaces are important, you can use the `<pre>` element or the `<codeblock>` element.

4. Inside the `<stepxmp>` element, add a `<p>` element, then insert an `<image>` element inside the `<p>` element.

```
...<step><cmd>When you have definitively identified a duck, make a note in your guide.</cmd>
  <stepxmp>
    <p><img href="checklist.png"/></p>
  </stepxmp>
</step>...
```

Note: The file `checklist.png` is in the sample files, in both the `lesson2` folder.

The `<stepresult>` element describes the result of the successful completion of the step.

The `<stepresult>` element must be the last element in a `<step>` element; no other elements can come after it.

5. After the closing tag of the `<stepxmp>` element, add a `<stepresult>` element.

```
...
  </stepxmp>
  <stepresult>
</step>
...
```

6. Inside the `<stepresult>` element, add a `<p>` element and add content to it as shown here.

```
...
  </stepxmp>
  <stepresult>
```

```

<p>You'll now have a record you can keep for the rest of your life.</p>
  </stepresult>
</step>
...

```

Interrupting the flow

There are two elements you can use to add additional information to your steps:

- Use the `<stepsection>` element to add comments between two steps.

This can be useful if you need to make a comment to the reader about the steps themselves, such as “The remaining steps focus on the the x component” or “The next two steps must be performed in a y environment.”

- Use a `<note>` or `<hazardstatement>` element to add an admonition before a `<cmd>` element in a `<step>` or `<substep>`.

Typically, you use the `<note>` or `<hazardstatement>` elements when you need to alert readers about a risk or danger before they read or perform the `<cmd>` action.

Practice

1. Continue working in the file `l_task_start.dita`.
2. After the closing tag of the last `<step>` element, add a `<stepsection>` element.

```

...
</step>
<stepsection>
</stepsection>

```

3. Add content to the `<stepsection>` element as shown here.

The text should be wrapped in a `<p>` element according to best practice.

```

...
<stepsection>
  <p>Once you've identified the duck, you can spend some time noting its behavior.</p>
</stepsection>

```

4. After the closing tag of the `<stepsection>` element, add a `<step>` element with a `<note>` element inside it containing the text shown here.

```

...
</stepsection>
<step>
  <note>Don't disturb the ducks while observing their behavior; they behave differently.

```

```
ntly when humans are present.</note>
</step>
```

5. After the closing tag of the <note> element, add the <cmd> element and add content to it as shown here.

```
...
</note>
<cmd>Watch how the ducks interact with one another, what they're eating, or how a
nd when they preen.</cmd>
</step>
```

Practice

1. Open the file lesson2/l_task_steps_exercise_start.dita and use it to convert the following content from [Content Strategy 101](#) into DITA:

Developing and implementing a content strategy

Create a content strategy.

1. Identify and interview stakeholders.
2. Establish implementation goals and metrics.
3. Define roles and responsibilities.

Example:

- Education
 - Development
 - Review
 - Approval
4. Establish timeline and milestones.

Implement your content strategy.

5. Build the content creation system.
6. Convert legacy content.

Choose from:

- Convert everything into the new system
- Identify high-priority content and convert it
- Just-in-time conversion

- Assess for conversion
- Convert nothing

7. Deliver content

Delivery method	Benefits
<i>PDF</i>	Visual design, option to print
<i>HTML</i>	Accessibility, interactivity, responsive design
<i>EPUB</i>	Electronic content when Internet access is unavailable

8. Capture project knowledge.

- Document your content model, specifications, and best practices.
- Provide training to authors on writing in the new environment.
- Provide training to staff on maintaining the new system.

9. Ensure long-term success.

Result: You can tick off the items you listed at the beginning of the project as accomplishments now.

2. Check your file `lesson2/l_task_steps_exercise_start.dita` against the sample file `lesson2/l_task_steps_exercise.dita`.

Lesson 3: Finishing up the task

Objectives

- Understand the use of the `<result>`, `<example>`, and `<postreq>` elements.

After you've added the steps, you can use three additional elements to complete the description of the task:

- `<result>` describes the result of having successfully performed the task
- `<example>` shows an example of the entire task
- `<postreq>` (postrequisites) tells what to do next

Note:

This lesson covers basic use of these elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Transitional information in semantic blocks](#)

Task results

Following the `<steps>` element, you can use the `<result>` element to describe the end result of a successful completion of the task.

This element is similar to the `<stepresult>` element, but while the `<stepresult>` element describes the result of completing a step, the `<result>` element describes the result of completing the entire task.

You can use `<stepresult>` to describe a confirmation message, provide a simple test the reader might perform to verify the end result, or merely include a congratulatory message to the reader.

Practice

1. Continue working in the file `l_task_start.dita`.
2. After the closing tag of the `<steps>` element, add a `<result>` element.

```
...
    </steps>
    <result>
    </result>
  </taskbody>
</task>
```

3. Add the text shown here to the result element.

```
<result>
<p>The fresh air, exercise, and thrill of seeing wild ducks should
give you an enhanced feeling of well-being.</p>
</result>
```

Wrap the text in a `<p>` element according to best practice.

Making an example

The `<example>` element allows you to show an example of the entire task. The `<stepxmp>` element is similar, but it shows an example relevant to a single step instead of an example relevant to the entire task.

[Video: Adding an example in a DITA task](#)

Practice

1. Continue working in the file `l_task_start.dita`.
2. After the closing tag of the `<result>` element, add an `<example>` element.


```

...
    </result>
    <example>
    </example>
  </taskbody>
</task>

```

3. Add the following content to the <example> element:

```

...
    </result>
    <example>
    <p>Don't expect this kind of behavior: <xref href="http://dai.ly/x2j6pt1" format=
    "html"
                                scope="external">Duck Amuck</xref>
    </p>
    </example>
  </taskbody>
</task>

```

What's next?

The final element in a <taskbody> element is the postrequisites (<postreq>) element.

Use the <postreq> element to document the things that the reader must do once the task is complete. Examples include:

- Putting equipment away or closing files
- Documenting or logging the task completion
- Performing subsequent tasks

[Video: Strict sequence of finishing elements in a DITA task](#)

Practice

1. Continue working in the file l_task_start.dita.
2. After the closing tag of the <example> element, add a <postreq> element.

```

...
    </example>
    <postreq>
    </postreq>
  </taskbody>
</task>

```

3. Add the following content to the <postreq> element:

```

...
    </example>
    <postreq>
      <ul>
        <li>Watch the ducks again with a camera</li>
        <li>Go look for other birds</li>
      </ul>
    </postreq>
  </taskbody>
</task>

```

Practice

1. Open the file lesson3/l_task_finishing_exercise_start.dita and use it to convert the following content from [Content Strategy 101](#) into DITA:

Managing change

1. Demonstrate value to upper management and those in the trenches.
2. Offer training and knowledge transfer.

Example:

- Classroom training
 - Live web-based training
 - Train the trainer
3. Differentiate between legitimate issues with the new workflow and reflexive recalcitrance.
 4. Enlist participants in a pilot project to explain process change.

Results: Good management is critical when a company changes workflow; without it, the implementation of new processes will likely fail. Bad management kills implementations, and things can get ugly for everyone involved.

Example: Without good change management, a percentage of staff who will be dead set against changes on general principle could gain the support of other team members, until almost everyone refuses to use the new system.

What to do next: When you have managed change successfully, you can start creating useful content in the new system.

2. Check your file lesson3/l_task_finishing_exercise_start.dita against the sample file lesson3/l_task_finishing_exercise.dita.

Lesson 4: Best practices for tasks

Objectives

- Identify best practices for authoring task topics
- Show examples of best practices in a task topic

This lesson covers best practices for authoring task topics. You will learn about planning tasks, providing appropriate context for a task, using a reasonable number of steps, using substeps appropriately, and keeping an eye on opportunities for reuse.

Additional reading

[DITA Style Guide, Restricting tasks to one procedure only](#)

[DITA Style Guide, Complex nested tasks](#)

[DITA Style Guide, Finding elements to re-use](#)

[Task Oriented Writing, contributed by Pam Noreault, Tracey Chalmers, and Julie Landman](#)

[DITA Topic Based Writing, contributed by Pam Noreault, Tracey Chalmers, and Julie Landman](#)

Plan your tasks

It's tempting to jump into creating tasks, writing until you think you're done. However, this approach risks missing important tasks, resulting in incomplete coverage.

Rather than create tasks as they occur to you, a far better strategy is to plan out the set of tasks you need to document.

There are many resources for and approaches to task analysis. The essential point to all of them is to:

1. Gather information about what the reader should accomplish.
2. Identify the tasks you need to document.
3. Understand the scope of those tasks.
4. Consider your readers' abilities and needs.
5. Plan your task topics accordingly.

Once you have planned the tasks, it's a good idea to have all stakeholders review the entire list of tasks. These should include writers, editors, subject matter experts and other content contributors, project managers, and anyone else who has a hand in your content development process. This step is important, because the more contributors you have, the less chance you have of missing an important task.

The list of tasks can help you plan your concepts, reference, and glossary entry topics. For each task in the list, consider what concepts and reference information is needed to support that task. You can also identify the terms in the task that will make good candidates for glossary entry topics.

Provide appropriate context

The `<context>` element should provide enough context for readers to understand why they need to perform a task or what the task accomplishes. The `<context>` element should not be used for extensive conceptual or background information.

If you need to provide additional conceptual information, background information, or operating principles, create a concept topic and provide a link to the concept from your task.

Note that there can be some similarity between the `<context>` element and the `<shortdesc>` (short description) element. Both provide information about the purpose of a task. However, the `<shortdesc>` element typically provides succinct information used in link previews or search results, whereas the `<context>` element can be slightly more detailed and can provide additional information about what the task can accomplish.

Use a reasonable number of steps

There is no technical limit to the number of steps you can have in a DITA task; however, the task should not be too long.

Although the length of a task can be subjective, longer tasks with a large number of steps can seem overwhelming to readers (particularly novice readers).

It is difficult to give an absolute rule for the maximum number of steps that you should have in a task. Some writers use a ten-step maximum, but allow for limited exceptions. Nearly everyone agrees that a procedure with more than twenty steps is highly problematic.

An additional effect of longer tasks is that the reader may have difficulty keeping track of where they are when performing the task. This is even more true when there is a long list of very similar steps.

Addressing shorter tasks: you may encounter some tasks that only require one or two steps. These might be unavoidable if the goal is complete coverage of a set of tasks. Although most of your tasks will probably be longer, there is absolutely nothing wrong with having a single-step or two-step task if necessary.

Use substeps appropriately

The DITA `<substep>` element can be useful, but it should be applied sparingly.

If you find that you are using substeps extensively or that you have some very long, involved substeps, you may want to revisit your task analysis. Consider carefully how you divided your major tasks into task topics. It may be that you did not divide the major tasks far enough.

Another clear warning sign that your task analysis is not complete: you are using many substeps, but frequently find the need for sub-substeps.

Substeps should only be used in a handful of situations, such as:

- You encounter a complex step, where readers could benefit from additional details.
- Your audience includes some novice readers who might benefit from some additional guidance. Most readers will understand all that the `<cmd>` element directs them to do, but some less-experienced readers may need to have things spelled out in a little more detail with substeps.
- Some output types might allow you to hide substeps that provide extra detail until the reader asks for more information.
- Procedures in which the steps are not homogeneous. Occasionally you'll encounter a task where most of the steps require a consistent level of explanation, but one or two steps—although at the same level as the other steps—require much more detail.

Be careful using substeps where the `<choice>` or `<choicetable>` element are more appropriate.

Keep an eye toward reuse

Although the LearningDITA courses have not yet covered reuse (which is one of the advantages of DITA), task topics contain many potential areas for reuse.

As you create the steps in your tasks, watch out for steps that are the same from one task to another, such as:`<step><cmd>Click <uicontrol>OK</uicontrol>.</cmd></step>`

You might create a series of tasks that all begin with the same set of step to start the process, such as shutting down the equipment, or removing access panels. Similarly, a series of tasks might use a common set of steps to finish the procedure. These are excellent candidates for reuse.

Another common area for reuse in task topics is standard text, such as cautions or warnings that accompany steps in many procedures.

In addition to reusing the `<step>` element, the `<prereq>`, `<example>`, `<choices>`, and `<choicetable>` elements in a task are all good candidates for reuse.

Reuse is another reason to plan your tasks, as described in the first topic of this Best Practices lesson. When doing your task planning, it is useful to identify steps that are common to many tasks. An important aspect of reuse is to identify the elements that you need to reuse and then maintain those elements in a separate file (or files). Once approved, these files can be secured so that they are modified only when necessary.

Course IV. The DITA Reference and Glossary Topic Types

Lesson 1: Creating a reference topic

Objectives

- Understand the reasons for using the reference topic type
- Know the basic structure of a reference topic
- Understand the purpose of the <refsyn> and <properties> elements

This lesson shows how to create a new reference topic and how to add the first common elements found in a reference topic.

Note:

This lesson covers basic use of the reference elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Reference](#)

Housekeeping and sample files

Download [reference samples.zip](#) now. It contains the sample files for the entire DITA reference and glossary topic type course. Extract the contents and put them in a directory that you can access easily.

Inside the samples folder, you will find four files: l_glossentry_start.dita, l_glossgroup_start.dita, l_glossorg_start.ditamap, and l_reference_start.dita, which you will use to complete exercises throughout the course. You will also find the following folders corresponding to lessons in this course: lesson1 and lesson3.

In each lesson folder, you will find several DITA files:

- l_filename.dita: The completed version of the sample files you can use to check your work during each lesson.
- l_filename_exercise_start.dita: The empty file you will add content to as you complete the exercise at the end of each lesson.
- l_filename_exercise.dita: The completed version of the exercise file you can use to check your work.

- `l_filename.ditamap`: The completed version of the ditamap you can use to check your work in Lesson 3.

Each lesson will instruct you on which files to use for the exercises. Save your file as you complete each step to avoid losing your work.

Create a local copy of each file to work in as you complete the lessons. That way, if you reach a point where your working file doesn't match the examples, or is broken for any reason, you can make a fresh copy and resume your work or start over.

In the instructions and examples, we show you the DITA code for each sample file. Most DITA editors have auto-complete or other similar features to guide you through the process of adding elements (for example, if you type the opening tag of an element, most DITA editors will automatically add the closing tag for you). Therefore, you will probably not need to create every piece of code from scratch as you work. Our demo videos were created in [oXygen XML Editor](#) and show the differences between working in author view, which presents the DITA content in a user-friendly visual format, and working in text view, which shows the DITA code.

About reference topics

Reference topics answer the question “What?”, as in:

- What value do I enter here?
- What does it mean when the LED flashes three times?
- What command parameters should I use?
- What status values can I expect from this function call?
- What does this error message mean?

Reference topics provide information about a specific item. While a concept topic may provide general background information and answer the question “Why?”, the reference topic provides specific information. Typical readers of reference topics find the information they need, then continue with what they were doing; they typically do not read or browse any additional information.

The format of content within a reference is determined by the authoring organization, and ideally codified within a style guide. The examples in this lesson show only one way that you can organize information.

While reference topics can contain many of the elements that we've already covered in previous courses, this course will focus on new ones.

Reference topics fall into two general categories:

- Single, stand-alone topics containing a small number of tables or figures. Examples include:
 - Matrix tables of devices and specific values or tolerances

- Illustrations of device ports or pinouts
- Multiple, related topics that provide information about related items, one topic per item. These topics usually have a parallel structure so that the reader can consistently locate information. Examples include:
 - Command line reference
 - API function reference
 - Error code reference
 - Device data sheet

Creating a new reference topic

At a minimum, the reference topic must contain a <reference> root element (with an id attribute) that contains a <title> element.

Following the <title> element and an optional <shortdesc> element, a <refbody> element contains the reference content. By design, the content in the <refbody> element is limited to a handful of elements. In other words, a reference topic is not as flexible in its content as a concept topic.

The body of the reference topic can contain elements that have been introduced in previous courses: <table>, <fig> (figure) or <image>, <example>, and <section>. You usually use the <section> element within a <refbody> element to organize the common sections of a reference topic, such as command descriptions, usage guidelines, error codes, and the like. In addition, the <refbody> element can contain two elements that are specific to reference topic types: <refsyn> (reference syntax) and <properties>.

The body of the reference can contain any number of these elements, in any order.

The <example>, <section>, and <refsyn> elements can contain the <title> element. This allows you to create subdivisions within a reference topic. If necessary, an output transform can insert its own titles for <properties> elements.

[Video: Creating a DITA reference topic](#)

Practice

1. Make a copy of the file l_reference_start.dita and open it in your editor.

Note: If you are using a DITA editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE reference PUBLIC "-//OASIS//DTD DITA Reference//EN" "reference.dtd">
<reference id="my_first_ref">
```



```
<title></title>
</reference>
```

The first line (which begins with <?xml) is an XML declaration, which is a standard part of all XML files.

The second line is the DOCTYPE declaration, which tells DITA editors or DITA output generators that this is a DITA reference topic. The programs then use that information when validating the content of the topic. The DOCTYPE will be specific to each topic type that you create.

The third line contains the opening tag of the <reference> element.

The fourth line contains the <title> element.

2. Inside the <title> element, update the text of the reference title.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE reference PUBLIC "-//OASIS//DTD DITA Reference//EN" "reference.dtd">
<reference id="my_first_ref">
  <title>tNav</title>
</reference>
```

3. After the <title> element, add a <refbody> element.

The <refbody> element contains all the actual content in the reference.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE reference PUBLIC "-//OASIS//DTD DITA Reference//EN" "reference.dtd">
<reference id="my_first_ref">
  <title>tNav</title>
  <refbody>
  </refbody>
</reference>
```

4. Within the <refbody> element, add a <section> element and add content to it as shown.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE reference PUBLIC "-//OASIS//DTD DITA Reference//EN" "reference.dtd">
<reference id="my_first_ref">
  <title>tNav</title>
  <refbody>
    <section><p>The <cmdname>tNav</cmdname> command is used in the
      Duck Database to navigate to a different table or view.</p></section>
  </refbody>
</reference>
```

Note:

The `<cmdname>` (command name) element denotes the name of a command when referenced as a part of software, and is a part of the programming domain. For more information on the programming domain, see the [OASIS DITA Version 1.2 Standard](#).

Describing the syntax

A crucial part of an API reference or command line reference is the syntax description, where the possible options or forms of each command or function call are indicated.

A syntax diagram uses a textual or graphic shorthand to depict the keywords, placeholders (variables), and other characters that a command or function call requires. Textual syntax diagrams typically use brackets [], braces {}, and bars | to indicate optional or required portions of syntax. Graphical syntax diagrams (often called “railroad diagrams”) use lines and arrows to depict optional, repeated, and required portions of syntax. For an illustration of graphical syntax diagrams, see this [Wikipedia](#) page.

Because the syntax description is frequently required in reference topics, the reference specialization provides the `<refsyn>` element to contain the command syntax.

The `<refsyn>` element is specialized from the `<section>` element and can contain any elements allowed in the `<section>` element, including the `<title>` element.

There are a number of ways to add the actual syntax diagram to the `<refsyn>` element, such as:

- Using `<image>` or `<fig>` elements to include an image of the syntax diagram (generally not recommended)
- Using `<codeblock>` or `<pre>` (preformatted) elements to create a textual syntax diagram (acceptable, but allows very little additional markup)
- Using `<synph>` (syntax phrase) to create a textual syntax diagram (illustrated in this lesson)
- Using the `<syntaxdiagram>` element, which uses additional elements to describe the parts and relationships in a syntax diagram. These elements can be rendered in a number of ways, depending on your output transform. However, there is no visual editor available for the `<syntaxdiagram>` elements, so using them can be challenging.

For further discussion of the pros and cons of using the `<synph>` and `<syntaxdiagram>` elements, see [Simon Bate’s blog post on understanding diagrams in DITA](#).

Practice

1. Continue using `l_reference_start.dita`.
2. After the `<section>` element, add a `<refsyn>` element.

```
<title>tNav</title>
<refbody>
<section><p>The <cmdname>tNav</cmdname> command is used in the
Duck Database to navigate to a different table or view.</p></section>
```

```

<refsyn>
</refsyn>
</refbody>
</reference>

```

3. Inside the `<refsyn>` element, add a `<synph>` (syntax phrase) element to contain the command example and add content to it as shown.

```

    <title>tNav</title>
<refbody>
<section><p>The <cmdname>tNav</cmdname> command is used in the
Duck Database to navigate to a different table or view.</p></section>
<refsyn>
<synph>
-tNav tName [tView]
</synph>
</refsyn>
</refbody>
</reference>

```

4. Tag the string “tName” inside the `synph` with the `<var>` (variable) element.

The `<var>` element identifies the string as a variable or placeholder in the syntax. In most output transforms, this will be rendered in italics.

```

    <title>tNav</title>
<refbody>
<section><p>The <cmdname>tNav</cmdname> command is used in the
Duck Database to navigate to a different table or view.</p></section>
<refsyn>
<synph>
-tNav <var>tName</var> [tView]
</synph>
</refsyn>
</refbody>
</reference>

```

Adding a properties table to a reference

In most API reference or command line reference topics, the syntax diagram is followed immediately by a list or table that describes the various keywords, placeholders, and other options in detail. The syntax diagram shows the reader where these objects are used; the properties table provides information about why the reader would use the objects.

For example, in a graphics export command the filetype parameter might allow the values “gif”, “png”, “bmp”, and “svg”; you might want to give specific information about the implications of using each of the filetypes.

In the DITA reference topic type, the `<properties>` element provides one way to structure this information. The `<properties>` element is one of the elements that you can use within a `<refbody>` element.

The `<properties>` element contains one or more `<property>` elements, each of which describes a single property. The `<property>` element can contain elements for type (`<proptype>`), value (`<propvalue>`), and description (`<propdesc>`). In typical use, these might contain:

Document type	<code><proptype></code>	<code><propvalue></code>	<code><propdesc></code>
Command line reference	Command parameter	Value or keyword allowed for the parameter	Description of the parameter (what it means, what its use implies)
API reference	Function argument	Argument data type or object type	Description of the argument and its effect on the API
Database reference	Database field or column name	Data type	Description of the content

In the case of a command parameter (`<proptype>`) that has multiple keywords, you can create a `<property>` element with `<proptype>`, `<propvalue>`, and `<propdesc>` elements for the first keyword. Then for subsequent keywords, create additional `<property>` elements that only contain `<propvalue>` and `<propdesc>` elements.

The `<properties>` element is a specialization of the `<simpletable>` element and is often rendered in output as a table. If necessary, you can create column heads for the type, value, and description columns using a `<prophead>` element containing `<proptypehd>`, `<propvaluehd>`, and `<propdeschd>` elements.

[Video: Creating a DITA properties table](#)

Practice

1. Continue using `l_reference_start.dita`.
2. After the `<refsyn>` element, add a `<properties>` element and add a `<prophead>` element within it.

```

...
</refsyn>
<properties>
  <prophead>

    </prophead>
  </properties>
</refbody>
</reference>

```

3. Inside the `<prophead>` element, add a `<proptypehd>`, `<propvaluehd>`, and `<propdeschd>` element and add content to them as shown.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Reference//EN" "reference.dtd">
<reference id="my_first_ref">
  <title>tNav</title>
  <refbody>
    ...
  </refsyn>
  <properties>
    <prophead>
      <proptypehd>Type</proptypehd>
      <propvaluehd>Name</propvaluehd>
      <propdeschd>Description</propdeschd>
    </prophead>
  </properties>
</refbody>
</reference>

```

Note:

While not all required, the elements added here must be in this specific order. If they are not, your document will not validate.

4. After the <prophead> element, add a <property> element.

```

...
</refsyn>
<properties>
  <prophead>
    <proptypehd>Type</proptypehd>
    <propvaluehd>Name</propvaluehd>
    <propdeschd>Description</propdeschd>
  </prophead>
  <property>
  </property>
</properties>
</refbody>
</reference>

```

5. Inside the <property> element, add a <proptype>, <propvalue>, and <propdesc> element and add content to them as shown.

```

...
</refsyn>
<properties>
  <prophead>
    <proptypehd>Type</proptypehd>
    <propvaluehd>Name</propvaluehd>
    <propdeschd>Description</propdeschd>
  </prophead>

```

```

    <property>
      <proptype>Table</proptype>
      <propvalue>dbo.DBBL</propvalue>
      <propdesc>Dabbling ducks</propdesc>
    </property>
  </properties>
</refbody>
</reference>

```

Note: If you omit any of the headers, you should also omit the associated property element. For example, if your <prophead> element does not contain a <propdeschd>, you should omit the <propdesc> element within your <property> elements.

6. Add additional <property> elements and add content to them as shown.

```

...
</refsyn>
<properties>
  <prophead>
    <proptypehd>Type</proptypehd>
    <propvaluehd>Name</propvaluehd>
    <propdeschd>Description</propdeschd>
  </prophead>
  <property>
    <proptype>Table</proptype>
    <propvalue>dbo.DBBL</propvalue>
    <propdesc>Dabbling ducks</propdesc>
  </property>
  <property>
    <proptype></proptype>
    <propvalue>dbo.DVNG</propvalue>
    <propdesc>Diving ducks</propdesc>
  </property>
  <property>
    <proptype></proptype>
    <propvalue>dbo.WHST</propvalue>
    <propdesc>Whistling ducks</propdesc>
  </property>
  <property>
    <proptype>View</proptype>
    <propvalue>regn.ne</propvalue>
    <propdesc>Ducks located primarily in the northeast</propdesc>
  </property>
  <property>
    <proptype></proptype>
    <propvalue>regn.se</propvalue>
    <propdesc>Ducks located primarily in the southeast</propdesc>
  </property>
  <property>

```

```

        <proptype></proptype>
        <propvalue>pttrn.migr</propvalue>
        <propdesc>Ducks organized by migratory pattern</propdesc>
    </property>
</properties>
</refbody>
</reference>

```

Practice

Open the file lesson1/l_reference_exercise_start.dita and use it to convert the following content into DITA:

addEntry

The addEntry command is used to enter a new documentation release into the content analysis database.

```
-addEntry {nLanguage | nVersion | nOutput}
```

Required?	Parameter	Description
Yes	nLanguage	The language displayed by default
	nVersion	The version of the content displayed by default
	nOutput	The required delivery format

Lesson 2: Best practices for reference topics

Objectives

- Accelerate the authoring process by creating reference topic templates
- Improve maintainability of links by using relationship tables
- Understand the use of the DITA domain elements

This lesson covers best practices for authoring reference topics. You will learn how to create templates for reference topics, use relationship tables to manage links, and use DITA domain elements for improved semantics.

Additional reading

[DITA Style Guide, Relationship tables](#)

[DITA Style Guide, Element domains](#)

[DITA Style Guide, Finding elements to re-use](#)

Create templates

When a reference work consists of multiple topics—one for each command, function call, or other serial component—each of the topics should follow the same organization. For example, each topic in a command line reference could contain a brief description, the command syntax, a description of the parameters, an in-depth discussion, examples, and potential error messages. This allows readers to consistently find the same type of information in the same area of each topic.

Each of these items should be contained in a <section> element within the <refbody> element, along with a <title> element that labels the section.

If you need to create many of these topics, consider creating a template. A template is a standalone DITA topic that does not contain much content, but has the necessary repeatable structure in place. For example, you could create a reference topic that contains each of the sections and the <title> element. Add text in each section indicating that it is a placeholder section.

Note: The template will need to include an id attribute on the <reference> element; make sure content creators know to avoid creating duplicate IDs. One way to do this is to use an explicit placeholder value for the id attribute, like id="DUMMY", so that content creators are know to replace it with a valid value.

Use relationship tables for links

A very useful feature of command line or API reference documentation is when each topic contains links to related commands or functions. For example, a command for mounting a device links to the command for unmounting a device; a command for configuring ports links to other commands that also pertain to ports; a function for initiating a network connection links to the next function most network programmers will need to call.

Before DITA, these lists of related links could be difficult to maintain, particularly for products that are still under development. If a component changes (or is added or deleted), you must find all the relevant links and modify them.

However, if you use a DITA relationship table to keep track of related reference topics, the DITA Open Toolkit can create these links for you when your document is generated. If a component changes, you make one or two changes in the relationship table, and the next time you generate your document, the change is reflected in all the links.

Note also that if you are converting legacy content to DITA, your conversion process can eliminate (or at least comment out) these lists of links.

Use DITA domain elements

This applies to other topic types, but is worth mentioning for reference topics, because of their frequent association with programming and user interface elements.

DITA defines a number of elements that are useful for topics that describe programming, computer software, and computer user interfaces. These are called domain specializations, because—although they’re specializations of existing elements—they can be used across all topic specializations, such as concept, task, and reference topic types.

It’s advisable to use these semantic elements because they:

- Provide information about their content to other authors
- Aid in faceted searches (“Find all filenames that use product name X”)
- Can direct the output generator to format content to make it more easily visible to readers

Some of the elements which can be useful are:

Programming domain

<apiname>, <codeblock>, <codeph> (code phrase), <option>, <parml> (parameter list), <synph> (syntax phrase)

Computer software domain

<cmdname> (command name), <filepath>, <varname> (variable name), <userinput>, <systemoutput>

Computer user interfaces

<uicontrol> (user interface control), <menucascade>, <wintitle> (window title), <screen>

Two of the computer user interface elements deserve additional attention: <uicontrol> and <menucascade>. You use <uicontrol> when describing a user interface button or a command on a menu. If you want to describe the sequence of menu items to get to a command, use a series of <uicontrol> elements inside a <menucascade> element:

```
...by choosing <menucascade><uicontrol>File</uicontrol><uicontrol>Open</uicontrol></menucascade>.
```

An output transform inserts whatever character is appropriate between commands. This helps ensure that all menu cascades are formatted in the same way, using the same separator character:

```
...by choosing File » Open
```

For more information on all domain elements, see the [DITA Specification](#).

We also strongly advise that you create a style guide in which you document the specific uses of each of these element within your product documentation.

Lesson 3: Creating a glossary entry

Objectives

- Differentiate a glossary entry topic and a glossary group topic
- Understand the basic structure of a glossary entry topic

- Add glossary entries to a DITA map file

This lesson shows how to create a new glossary entry and glossary group topic and add their basic elements.

Note: This lesson covers basic use of the glossary entry elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

Additional reading

[DITA Style Guide, Glossaries](#)

About glossary entry and glossary group topics

Use a glossary entry topic to define a specific term or concept contained within your content.

Each glossary entry contains a single term and its definition. This means that each glossary entry is a separate file.

In some cases maintaining one file for each glossary entry can be an inconvenience, so where it would be beneficial to keep a number of glossary entries together in a single file, DITA provides the glossary group topic type. The glossary group topic type contains one or more glossary entries, each of which is structured just like the stand-alone glossary entry topic.

Initially, the idea of having each glossary term in a separate DITA topic (in other words, a separate file) seems like a lot of trouble. But there are a number of good reasons for maintaining glossary entries as separate topics:

- **Reuse.** Once you've created your glossary entries, you—or anyone else on your team—can use those same glossary entries in multiple work products. If the glossary entries are buried in a glossary group, they are much harder to reuse.
- **Ownership.** Typically the glossary entries in a glossary group are specific to a product or team. Trying to integrate new glossary entries for your product into an existing glossary group can be difficult.
- **Localization.** If your glossary will be translated, it will probably need to be sorted according to the translated glossary terms. Maintaining the entries in a glossary group topic may create an impediment to the sorting process.
- **References.** There are several DITA elements that you can use to create a reference from text to a glossary entry. These references are easy to build when each glossary entry is in a separate file. It requires much more work when the glossary entries are gathered in a glossary group file.

Despite these arguments, a reasonable case can be made for using glossary groups. As a result, this lesson describes both how to create glossary entry topics and how to create glossary group topics.

Creating a new glossary entry

The glossary entry topic must contain at least a <glossentry> (glossary entry) root element (with an id attribute) that contains a <glossterm> (glossary term) element.

The <glossterm> element is a specialization of the <title> element. The <glossterm> element can contain any element that a <title> element can contain.

The <glossdef> element must follow the <glossterm> element, and contains the definition. The <glossdef> element can contain any element that is allowed in the <section> element, except a <title> element.

The <glossbody> (glossary body) element can optionally follow the glossdef element, and can contain a series of specialized elements that can be used to provide additional details about the glossary term. The <glossbody> element is beyond the scope of this exercise, and will be covered in a later lesson.

[Video: Creating a DITA glossary entry](#)

Note:

Generally, it's a best practice to prepend "g_" to the filenames of your glossary entries in order to make it easier to find them in a file structure. However, in keeping with our other learning materials, we're using "l_".

Practice

1. Make a copy of the file l_glossentry_start.dita and open it in your editor.

Note: If you are using a DITA editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE glossentry PUBLIC "-//OASIS//DTD DITA Glossary//EN" "glossary.dtd">
<glossentry id="duck">
  <glossterm></glossterm>
</glossentry>
```

The first line (which begins with <?xml>) is an XML declaration, which is a standard part of all XML files.

The second line is the DOCTYPE declaration, which tells DITA editors or DITA output generators that this is a DITA glossentry topic. The programs then use that information when validating the content of the topic. The DOCTYPE will be specific to each topic type that you create.

The third line contains the opening tag of the <glossentry> element. The id attribute associated with the <glossentry> element is required.

The fourth line contains the <glossterm> element.

2. Inside the <glossterm> element, insert the glossary term to be defined.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glossentry PUBLIC "-//OASIS//DTD DITA Glossary//EN" "glossary.dtd">
<glossentry id="duck">
  <glossterm>duck</glossterm>
</glossentry>
```

3. After the <glossterm> element, add a <glossdef> element.

Whenever possible, the id attribute associated with the <glossentry> attribute should be human readable for ease of referencing later, preferably as close as possible to the glossary term.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Glossary//EN" "glossary.dtd">
<glossentry id="duck">
  <glossterm>duck</glossterm>
  <glossdef><p>A type of swimming bird characterized by webbed feet, a broad, flat
bill, and quacking</p></glossdef>
</glossentry>
```

Creating a new glossary group topic

Similar to the glossary entry topic, the glossary group topic must contain a single <glossgroup> root element with an id attribute that contains one <title> element and one or more <glossterm> or <glossgroup> elements, each with an id attribute.

Note: This lesson covers basic use of the glossary group elements. For the full specifications for each element, see the [OASIS DITA Version 1.2 Standard](#).

[Video: Creating a DITA glossary group](#)

Practice

1. Make a copy of the file l_glossgroup_start.dita and open it in your editor.

Note: If you are using a DITA editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE glossgroup PUBLIC "-//OASIS//DTD DITA Glossary Group//EN" "glossgroup.d
td">
<glossgroup id="duck_equipment">
  <title></title>
</glossgroup>
```

The first line (which begins with <?xml) is an XML declaration, which is a standard part of all XML files.

The second line is the DOCTYPE declaration, which tells DITA editors or DITA output generators that this is a DITA glossgroup topic. The programs then use that information when validating the content of the topic. The DOCTYPE will be specific to each topic type that you create.

The third line contains the opening tag of the <glossgroup> element. The id attribute associated with the <glossgroup> element is required.

The fourth line contains the <title> element.

2. Inside the <title> element, add a title to help identify the glossary group.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glossgroup PUBLIC "-//OASIS//DTD DITA Glossary Group//EN" "glossgroup.dtd">
<glossgroup id="duck_equipment">
  <title>Duck watching equipment</title>
</glossgroup>
```

While the <title> element is required as a part of the content model, you don't necessarily have to add content.

3. After the <title> element, add four <glossentry> elements with id attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glossgroup PUBLIC "-//OASIS//DTD DITA Glossary Group//EN" "glossgroup.dtd">
<glossgroup id="duck_equipment">
  <title>Duck watching equipment</title>
  <glossentry id=""></glossentry>
  <glossentry id=""></glossentry>
  <glossentry id=""></glossentry>
  <glossentry id=""></glossentry>
</glossgroup>
```

4. Within each <glossentry> element, add a <glossterm> element, and fill in the id attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glossgroup PUBLIC "-//OASIS//DTD DITA Glossary Group//EN" "glossgroup.dtd">
<glossgroup id="duck_equipment">
  <title>Duck watching equipment</title>
  <glossentry id="binoculars">
    <glossterm>binoculars</glossterm>
  </glossentry>
  <glossentry id="duck_bait">
```

```

        <glossterm>duck bait</glossterm>
    </glossentry>
    <glossentry id="duck_call">
        <glossterm>duck call</glossterm>
    </glossentry>
    <glossentry id="spotting_scope">
        <glossterm>spotting scope</glossterm>
    </glossentry>
</glossgroup>

```

5. After each <glossterm> element, add a <glossdef> element with definitions.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glossgroup PUBLIC "-//OASIS//DTD DITA Glossary Group//EN" "glossgroup.d
td">
<glossgroup id="duck_equipment">
    <title>Duck watching equipment</title>
    <glossentry id="binoculars">
        <glossterm>binoculars</glossterm>
        <glossdef>An optical device for use with both eyes used to view distant o
bjects</glossdef>
    </glossentry>
    <glossentry id="duck_bait">
        <glossterm>duck bait</glossterm>
        <glossdef>Bait used to attract ducks. Can range from specialty pellets to
cheap white bread</glossdef>
    </glossentry>
    <glossentry id="duck_call">
        <glossterm>duck call</glossterm>
        <glossdef>A device consisting of a tube with a reed contained inside it,
used to imitate and attract ducks</glossdef>
    </glossentry>
    <glossentry id="spotting_scope">
        <glossterm>spotting scope</glossterm>
        <glossdef>An optical device for use with a single eye, which tends toward
s greater detail view than binoculars</glossdef>
    </glossentry>
</glossgroup>

```

Organizing glossary entries in a map

Usually the glossary entry topics are gathered together in one or more DITA map files. The content of the map files and their organization depend greatly on the needs of your organization and the capabilities of your DITA output transforms. If you have a robust output transform, you will have more options in how you can structure your glossary map files.

The most straightforward strategy for glossary organization is to create a DITA map file that contains `<topicref>` elements for each of your glossary entries.

```
<map>
  <title>Duck glossary</title>
  <topicref href="g_acorns.dita"/>
  <topicref href="g_aythyinae.dita"/>
  <topicref href="g_canvasback.dita"/>
  ...
</map>
```

If the contents of this single glossary map file are arranged alphabetically, you can generate a usable glossary with the standard DITA output types without having to sort entries in the output transform.

Note: The contents of the `<title>` in this map file are mostly there to assist the writers. If this glossary map will be used in other maps, the contents of the `<title>` is ignored when the maps are merged.

Once you have created this glossary map, you can include it in any of your other maps (with a `<topicref>` or `<mapref>` element).

However, the world of glossary creation is often much messier and complex than simply organizing all entries in a single map. When your organization actually begins to implement DITA as part of your content strategy, it's a good idea to work with a DITA stylesheet developer to handle glossary information that's organized in a way that works for you and to generate the glossary output you need.

Some things to keep in mind about glossary organization and output:

- If you have several different contributing organizations, each of them may maintain a separate set of glossary entries. These will need to be merged (and sorted) as part of the output process.
- Similarly, different product lines may have different glossaries that may need to be merged, depending on some conditions or requirements.
- There may be other reasons for maintaining a number of separate glossary maps and merging them later.
- An output transform can also incorporate glossary entries from glossary group topics into an output glossary.
- If your content will be translated, it's highly likely that the glossary will need to be sorted for your target language. If your output transform handles the sorting tasks, your localization agency does not have to be responsible for sorting the translated glossary entries.

All of these situations can be handled by a well-crafted output transform.

Practice

1. Make a copy of the file `l_glossorg_start.ditamap` and open it in your editor.

Note: Opening a map file in an XML editor will often open the file in a different interface than when opening a DITA file. You may need to go through an extra step to open it within the editing interface.

You should see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map>
  <title></title>
</map>
```

2. Inside the <title> element, type a title for the map

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map>
  <title>Duck glossary</title>
</map>
```

3. After the <title> element, add a <topicref> element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map>
  <title>Duck glossary</title>
  <topicref href="l_glossentry.dita"/>
</map>
```

If you used a different name when you saved your file from the glossary entry exercise, use that filename instead of l_glossentry.dita.

4. If you created other glossary entry files, add topicref elements for each of those files.

Practice

Open the file lesson3/l_glossary_exercise_start.dita and use it to convert the following content into DITA:

Note:

While this exercise uses a single DITA file for conversion, it is also acceptable to convert it into a map with glossary entries.

Content strategy terms

structured authoring

An environment for creating content where the required structure is enforced by the authoring software and following the template is not optional

structured content

Information that is organized in a predictable way

searchable content

Information that is available via an Internet search

findable content

Information that performs well for relevant keywords

discoverable content

Information that has in-bound links, especially on social media

Lesson 4: Best practices for glossary entries and groups

Naming files and ids

If you are using a file system to manage your DITA content—or if your content management system allows you to specify filenames and id attributes—we advocate these conventions:

- When possible, start all filenames for your glossary entry topics with “g_”. The rest of the filename should reflect the content of the <glossterm> element in the glossary entry topic. This helps you find all your glossary entries within your filesystem. However, use the underscore (“_”) character to separate words, rather than a space (spaces in DITA filenames can cause problems).
- The id attribute for the glossary entry topic should be the same as the filename (minus the .dita extension). You may opt to omit the “g_” from the id attribute as well, so long as the id attribute value matches the same term used in the filename. So, if a topic is named “g_mallard.dita”, the id of the glossary entry topic should be “g_mallard” or “mallard”.
- Although this topic does not address the concept of keys in DITA, we also advise using the value of the id attribute as the key name.

Building a useful glossary

Some of the characteristics that make a good glossary definition include:

- Make your definitions short and to the point. Link to concept topics if additional explanation is needed.
- Do not use the glossary term (or a form of the term) as part of the definition.
- Avoid using terms that need further definition or require a search for another glossary term.
- Provide links to other glossary entries, where needed.
- Spell out abbreviations.

- Do not create circular references.

Useful glossaries have these qualities:

- If a term has multiple meanings, create a separate glossary entry for each meaning.
- Provide glossary entry topics for synonyms, particularly when creating glossary entries for concepts.

Adding a glossary to a bookmap

If you're organizing your content using a bookmap, use the `<booklists>` element (in either the `<frontmatter>` or `<backmatter>` elements) and add a `<glossarylist>` element to indicate where the glossary should be generated in the output.

```
<booklists>
...
  <glossarylist>
    <topicref href="glossary.ditamap" format="ditamap"/>
  </glossarylist>
...
</booklists>
```

If you need to point to the glossary DITA map, we advise using a nested `<topicref>` element, as shown here.

Manage glossary creation

In the early days of a project, it's fine if the glossary grows organically, with as many people contributing glossary entries as possible.

But as a project matures, more and more content creators will depend on the contents of the glossary. At this point, it's a good idea to manage the glossary contents. That is, have a process in place to manage:

- How new terms are added to the glossary.
- When and how changes are made to glossary entries.
- When an obsolete or deprecated term should be removed from the glossary.

Course V. Using DITA Maps and Bookmaps

Lesson 1: Creating a map

Objectives

- Explain the structure and organization of a DITA map.
- Create a DITA map and add <topicref> and <mapref> elements to it.
- Use a DITA map to create a hierarchy of referenced topics and maps.

This lesson shows how to create a new DITA map and how to add references to other topics and maps.

Additional reading

[DITA Style Guide, Purpose of ditamap files](#)

[DITA Style Guide, DITA map vocabulary](#)

[DITA Style Guide, The topicref element](#)

[Ditamaps, contributed by Pam Noreault, Tracey Chalmers, and Julie Landman](#)

Housekeeping and sample files

Download [maps bookmaps samples.zip](#) now. It contains the sample files for the entire DITA map and bookmap course. Extract the contents and put them in a directory that you can access easily.

Inside the samples folder, you will find the following sub-folders:

- exercises
- samples

Each lesson will instruct you on which folders and files to use for the samples and exercises. Save your file as you complete each step to avoid losing your work.

Create a local copy of each file to work in as you complete the lessons. That way, if you reach a point where your working file doesn't match the examples, or is broken for any reason, you can make a fresh copy and resume your work or start over.

In the instructions and examples, we show you the DITA code for each sample file. Most DITA editors have auto-complete or other similar features to guide you through the process of adding elements (for example, if you type the opening tag of an element, most DITA editors will automatically add the closing tag for you). Therefore, you will probably not need to create every piece of code from scratch as you work. Our demo videos were created in [oXygen XML Editor](#) and show the differences between

working in author view, which presents the DITA content in a user-friendly visual format, and working in text view, which shows the DITA code.

Creating a new DITA map

Maps are used to compile, organize, and define the relationships among DITA topics. A map might contain the following:

- all of the concept and reference topics with information about a single product
- all of the task topics that give instructions for using that product
- a glossary of related terms

Although a single DITA topic can be published on its own, many authors collect topics in maps to prepare them for publication.

Using maps can add flexibility and facilitate reuse in your publishing workflow. If you have a topic with content that applies to multiple products (such as common safety information), you can write that topic once and reference it in multiple maps.

You can also reference other maps from within a map. This allows you to group related sets of topics together. For example, you might need to compile all the maps for individual products into a single map for a product family.

Maps can contain metadata, such as product information, copyright, and publication date. Metadata in maps can be used to help your company manage and distribute your content more effectively—or help your customers find the content they need more easily.

At a minimum, a map must contain a root `<map>` element. The `<map>` element may contain the following elements:

- The `<title>` element, which allows you to name the map (such as “Product X User Guide”). The text in the `<title>` element can also be used as the title of your output document.
- Any number of `<topicref>` elements, which allow you to reference topics within the map. A `<topicref>` element can contain other `<topicref>` elements.
- Any number of `<mapref>` elements, which allow you to reference other maps within the main map.
- Any number of `<reltable>` elements, which allow you to define relationships in your content.

Note:

For more information about the `<map>` element, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Make a copy of the file `maps_bookmaps_samples/samples/_m_ducks_start.ditamap` and open it in your editor.

Note:

If you are using a DITA-aware text editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
  <title>xyz</title>
</map>
```

The first line (which begins with `<?xml>`) is an XML declaration, which is a standard part of all XML files.

The DOCTYPE declaration on the second line identifies this file as a DITA map.

The third line is the opening tag of the `<map>` element, which uses the unique ID “ducks”.

The `<title>` element on the fourth line contains the title of the map.

The fifth line uses the closing tag `</map>` to show where the `<map>` element ends.

2. Inside the `<title>` element, change the text of the title as shown in the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
  <title>Ducks</title>
</map>
```

Now that you’ve set up the base structure for your map, you can begin adding topic references.

Adding topic references to a map

The `<topicref>` element is structured as follows:

```
<topicref href="filepath/filename.dita">
```

The `href` attribute contains a link to a topic that is included in the map. In DITA, a `<topicref>` element in a map should not point to a topic that lives above the map in your folder structure, as this can cause problems with some output types. Therefore, it is best to store your maps at the top level.

The `<topicref>` element may contain the following elements:

- The <topicmeta> element, which defines metadata about the referenced topic.
- Any number of <topicref> elements.
- Any number of <mapref> elements.

By nesting <topicref> or <mapref> elements inside of a <topicref> element, you can create a hierarchy in your map. This makes it easier for you to organize your topics. You can reflect the hierarchy of topics in a table of contents when you publish the map.

When nesting <topicref> elements, it's important to keep your final output in mind. In DITA, there is no structural limit to the number of <topicref> elements that can be nested—any <topicref> element in a map can contain any number of <topicref> elements. However, each nested <topicref> element will typically equate to a new heading level in your published output. Therefore, it is generally considered best practice to avoid nesting <topicref> elements more than five levels deep (and, ideally, no more than two or three).

[Video: Creating a DITA map](#)

Note: For more information about the <topicref> element, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Continue using the file maps_bookmaps_samples/samples/_m_ducks_start.ditamap.
2. After the <title> element, add a <topicref> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
  <title>Ducks</title>
  <topicref href="c_wild_ducks.dita">
</topicref>
</map>
```

This <topicref> element creates a link to the concept topic c_wild_ducks.dita.

3. Inside the <topicref> element you just added, add another <topicref> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
  <title>Ducks</title>
  <topicref href="c_wild_ducks.dita">
    <topicref href="c_wild_duck_types.dita"/>
  </topicref>
</map>
```

By placing this `<topicref>` element inside the first one, you have nested the topic `c_wild_duck_types.dita` at the next level beneath `c_wild_ducks.dita` in the hierarchy of your map.

4. After the `<topicref>` element you just added, add two more `<topicref>` elements and add content to them as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
<title>Ducks</title>
<topicref href="c_wild_ducks.dita">
    <topicref href="c_wild_duck_types.dita"/>

    <topicref href="c_wild_duck_species.dita"/>
    <topicref href="t_watching_wild_ducks.dita"/>
</topicref>
</map>
```

These new `<topicref>` elements make the hierarchy in your DITA map even clearer. The topics `c_wild_duck_types.dita`, `c_wild_duck_species.dita`, and `t_watching_wild_ducks.dita` all pertain to wild ducks, and as such, they are nested under the topic `c_wild_ducks.dita`.

5. After the closing tag of the `<topicref>` element for `c_wild_ducks.dita`, add more `<topicref>` elements and add content to them as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
<title>Ducks</title>
<topicref href="c_wild_ducks.dita">
    <topicref href="c_wild_duck_types.dita"/>
    <topicref href="c_wild_duck_species.dita"/>
    <topicref href="t_watching_wild_ducks.dita"/>
</topicref>
<topicref href="c_domestic_ducks.dita">
    <topicref href="c_duckling_growth.dita"/>
    <topicref href="c_feeding_ducklings.dita">
        <topicref href="c_duck_weight.dita"/>
    </topicref>
</topicref>
<topicref href="c_duckdb.dita">
    <topicref href="c_writing_about_ducks.dita" locktitle="yes"/>
    <topicref href="r_tnav.dita"/>
</topicref>
</map>
```

The map hierarchy now contains three first-level `<topicref>` elements: `c_wild_ducks.dita`, `c_domestic_ducks.dita`, and `c_duckdb.dita`. Each of these `<topicref>` elements contains several

nested second-level <topicref> elements. One of these second-level <topicref> elements, c_feeding_ducklings.dita, contains a nested third-level <topicref> element, c_duck_weight.dita.

Adding map references to a map

The <mapref> element is structured as follows:

```
<mapref href="filepath/filename.ditamap" format="ditamap">
```

The href attribute contains a link to another map that is referenced in your main map. The format attribute specifies that you are linking to a DITA map.

The <mapref> element may only contain the <topicmeta>, <data>, and <data-about> metadata elements. You cannot nest <topicref> or <mapref> elements inside a <mapref> element in DITA.

Just as the <topicref> element allows you to reuse a topic in more than one map, the <mapref> element allows you to reuse a collection of topics. By building a map once and referencing it in other maps, you can save the time and effort it would have taken to re-create the same series or hierarchy of <topicref> elements in multiple maps.

In addition to reuse, the <mapref> element gives you added flexibility when you publish your content. You can group related maps together in a larger map so that you can quickly deliver all of the content for an entire product family. You can also use the <mapref> element to add a glossary to a map, as you'll learn to do in the following example.

Note:

For more information about the <mapref> element, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Continue using the file maps_bookmaps_samples/samples/_m_ducks_start.ditamap.
2. After the last <topicref> element, add a <mapref> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
<title>Ducks</title>
...
</topicref>
<mapref href="g_duck_glossorg.ditamap" format="ditamap"/>
</map>
```

This <mapref> element creates a link to the glossary g_duck_glossorg.ditamap. By using the <mapref> element to link to the glossary, you can avoid the hassle of using <topicref> elements to

build the glossary directly into your map. This also make your glossary more easily available for reuse in other maps.

3. Check your file `maps_bookmaps_samples/samples/_m_ducks_start.ditamap` against the sample file `maps_bookmaps_samples/samples/_m_ducks.ditamap`.

Practice

1. Open the file `maps_bookmaps_samples/exercises/_m_cs101_start.ditamap` and use it to assemble topics from [Content Strategy 101](#) into a DITA map, using the following hierarchical structure as a guide:
 - Controlling technical communication costs
 - The fallacy of low-cost documentation
 - Efficient technical content development
 - Reducing the cost of technical support
 - Better technical information, fewer support calls
 - More efficient support operations
 - Content collaboration across the organization
 - “Collaboration” ≠ “content free-for-all”
 - Marketing and product visibility
 - Supporting marketing with technical content
 - Reinforcing the marketing message
 - When technical content contradicts marketing
 - Increasing product visibility
 - Third-party books
 - Building user community and loyalty
 - Extending the game experience into content
 - Legal and regulatory issues
 - Avoiding legal exposure
 - Meeting regulatory requirements
 - Information delivery
 - Technical standards

2. Check your file `maps_bookmaps_samples/exercises/_m_cs101_start.ditamap` against the sample `filemaps_bookmaps_samples/exercises/_m_cs101.ditamap`.

Lesson 2: Creating a bookmap

Objectives

- Explain the structure and organization of a DITA bookmap.
- Create a bookmap and add `<chapter>` elements to it.
- Add `<topicref>` and `<mapref>` elements to the chapters in a DITA bookmap.
- Use a DITA bookmap to create a hierarchy of referenced topics and maps.

This lesson shows how to create a new DITA bookmap and how to develop a book-like structure.

Additional reading

[DITA Style Guide, The bookmap feature](#)

[DITA Style Guide, Nesting bookmaps within ditamaps](#)

[DITA Style Guide, Sample bookmap file](#)

Overview: maps vs. bookmaps

Bookmaps extend the functionality of maps by providing elements that allow you to create book-like structures. These include chapters, appendices, parts (which can be used to group chapters or appendices), and optional front matter and back matter. Although maps and bookmaps share a similar structure, bookmaps contain a more expansive set of elements, which provide many more options for preparing your content for publication—particularly if you have large amounts of content. This makes bookmaps better suited to publishing content in a book format, and more flexible than maps overall.

When you're deciding whether to compile your topics into a map or a bookmap, your publication and delivery process should be a major factor. If you need to deliver your content in print or PDF form, it is generally recommended that you use bookmaps instead of maps. Maps are sufficient for online-only distribution (such as HTML or online help), but if you plan to publish the same set of topics in both print-based and online formats, it's better to use a bookmap.

One of the main differences between bookmaps and maps is the ability to have chapters in a bookmap. In a map, you can nest second-level topics under a first-level topic to create a hierarchy. In a bookmap, these first-level topics are designated as chapters. You can also designate a first-level topic as an appendix in a bookmap. Although the hierarchical structure of topics is the same, there is added semantic value in calling the first-level topics chapters. When you publish the same set of topics in a bookmap instead of a map, your output transforms can format your chapter topics differently than the other topics.

Bookmaps also give you an additional way to define metadata with the <bookmeta> element. The <bookmeta> element, which contains publication-specific information, can only be applied to the <bookmap> element—it is not available in maps. The ability to use the <bookmeta> element is another reason that bookmaps are better suited than maps for publishing book-like structures.

[Video: DITA maps and bookmaps](#)

Creating a new bookmap

A bookmap is a specialization of a DITA map file, which uses the <bookmap> element rather than the <map> element as a container for its topic and map references. At a minimum, a bookmap must contain a root <bookmap> element. The <bookmap> element may contain the following elements:

- The <booktitle> element, which allows you to name the bookmap (such as “Product X Manual”).
- The <bookmeta> element, which defines metadata about the bookmap.
- The <frontmatter> element, which contains information typically found at the beginning of a book, such as a table of contents or a preface.
- Any number of <chapter> elements. Each <chapter> element contains a link to a topic that is included in the bookmap at the first level. A <chapter> element can also contain any number of subordinate <topicref> elements.
- The <backmatter> element, which contains information typically found at the end of a book, such as an index or a glossary.

Note: For more information about the <bookmap> element, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Make a copy of the file `maps_bookmaps_samples/samples/_b_ducks_start.ditamap` and open it in your editor.

Note:

If you are using a DITA-aware text editor, make sure you are in text mode, rather than author or visual mode.

You should see this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
</bookmap>
```

The first line (which begins with <?xml>) is an XML declaration, which is a standard part of all XML files.

The DOCTYPE declaration on the second line identifies this file as a bookmap.

The third line is the opening tag of the <bookmap> element, which uses the unique ID “ducks”.

The fourth line uses the closing tag </bookmap> to show where the <bookmap> element ends.

2. Inside the <bookmap> element, add the <booktitle> element as shown in the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
  <booktitle>
</booktitle>
</bookmap>
```

3. Inside the <booktitle> element, add the <mainbooktitle> element and add content to it as shown in the following example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
  <booktitle>
    <mainbooktitle>Main book title</mainbooktitle>
  </booktitle>
</bookmap>
```

Now that you have set up the basic framework of your bookmap, you can begin adding chapters.

Adding chapters to a bookmap

The <chapter> element is structured as follows:

```
<chapter href="filepath/filename.dita">
```

Except for the element name, the structure of a <chapter> element matches the structure of the <topicref> element. In a <chapter> element, the href attribute contains a link to a topic that is included in the bookmap at the first level, or chapter level.

The <chapter> element may contain the following elements:

- The <topicmeta> element, which defines metadata about the referenced chapter-level topic.
- Any number of <topicref> elements.
- Any number of <mapref> elements.

If you’re using a bookmap, you’re almost certainly going to be distributing your content in print or PDF form, so it’s important to think about the physical space limitations of this type of output when you’re building your hierarchy of topics. If you nest too many levels of topics within a chapter, you risk

running out of ways to differentiate your heading levels, whether it's by decreasing font sizes or increasing indents. This could cause confusion for people who are trying to read your content.

Note:

For more information about the <chapter> element, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Continue using the file maps_bookmaps_samples/samples/_b_ducks_start.ditamap.
2. After the closing tag of the <booktitle> element, add a <chapter> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
  ...
</booktitle>
<chapter href="c_wild_ducks.dita">
</chapter>
</bookmap>
```

This <chapter> element creates a link to the concept topic c_wild_ducks.dita.

3. Inside the <chapter> element you just added, add a <topicref> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
  ...
<chapter href="c_wild_ducks.dita">
  <topicref href="c_wild_duck_types.dita"/>
</chapter>
</bookmap>
```

By placing this <topicref> element inside <chapter> element, you have nested the topic c_wild_duck_types.dita within the chapter c_wild_ducks.dita in the hierarchy of your bookmap.

4. After the <topicref> element you just added, add two more <topicref> elements and add content to them as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
  ...
<chapter href="c_wild_ducks.dita">
  <topicref href="c_wild_duck_types.dita"/>
```

```

        <topicref href="c_wild_duck_species.dita"/>
        <topicref href="t_watching_wild_ducks.dita"/>
</chapter>
</bookmap>

```

The chapter `c_wild_ducks.dita` now contains three nested topics: `c_wild_duck_types.dita`, `c_wild_duck_species.dita`, and `t_watching_wild_ducks.dita`.

5. After the closing tag of the `<chapter>` element, add more `<chapter>` elements and add content to them as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
...
</chapter>
<chapter href="c_domestic_ducks.dita">
    <topicref href="c_duckling_growth.dita"/>
    <topicref href="c_feeding_ducklings.dita">
        <topicref href="c_duck_weight.dita"/>
    </topicref>
</chapter>
<chapter href="c_duckdb.dita">
    <topicref href="c_writing_about_ducks.dita" locktitle="yes">
    </topicref>
    <topicref href="r_tnav.dita"/>
</chapter>
</map>

```

The bookmap hierarchy now contains three chapters: `c_wild_ducks.dita`, `c_domestic_ducks.dita`, and `c_duckdb.dita`. Each of these chapters contains several nested `<topicref>` elements. One of these second-level `<topicref>` elements, `c_feeding_ducklings.dita`, contains a nested third-level `<topicref>` element, `c_duck_weight.dita`.

6. Check your file `maps_bookmaps_samples/samples/_b_ducks_start.ditamap` against the sample file `maps_bookmaps_samples/samples/_b_ducks.ditamap`.

Practice

1. Open the file `maps_bookmaps_samples/exercises/_b_cs101_start.ditamap` and use it to assemble topics from [Content Strategy 101](#) into a DITA map, using the following hierarchical structure as a guide:
 - **Chapter 1: Controlling technical communication costs**
 - The fallacy of low-cost documentation
 - Efficient technical content development

- Reducing the cost of technical support
 - Better technical information, fewer support calls
 - More efficient support operations
 - Content collaboration across the organization
 - “Collaboration” ≠ “content free-for-all”
 - **Chapter 2: Marketing and product visibility**
 - Supporting marketing with technical content
 - Reinforcing the marketing message
 - When technical content contradicts marketing
 - Increasing product visibility
 - Third-party books
 - Building user community and loyalty
 - Extending the game experience into content
 - **Chapter 3: Legal and regulatory issues**
 - Avoiding legal exposure
 - Meeting regulatory requirements
 - Information delivery
 - Technical standards
2. Check your file `maps_bookmaps_samples/exercises/_b_cs101_start.ditamap` against the sample file `maps_bookmaps_samples/exercises/_b_cs101.ditamap`.

Lesson 3: Advanced DITA map and bookmap concepts

Objectives

- Describe the difference between map and bookmap metadata
- Demonstrate where `<frontmatter>` and `<backmatter>` elements would be used
- Create and populate `<booklists>` elements for a table of contents and an index

This lesson presents advanced concepts about maps and bookmaps, including metadata, content grouping, and using the `<booklists>` element to create items such as a table of contents or an index.

Additional reading

[DITA Style Guide, Metadata](#)

Adding metadata to a map

Just as topic metadata allows you define information about topics, map metadata allows you to define information about a DITA map and the topic references contained in the map.

You add metadata to a DITA map using the `<topicmeta>` element. The `<topicmeta>` element can exist at the main map level (inside the `<map>` element) or inside any `<topicref>` or `<mapref>` element within the map.

When used at the map level, map metadata adds information to a map and facilitates searching and filtering. In most cases, map metadata is not shown in the final published output, but it can influence the way that your content is published—for example, by specifying how an output transform should generate cover information.

When applied to `<topicref>` or `<mapref>` elements, map metadata provides additional information about the referenced topics or maps and can even override metadata contained in those topics and maps. For example, if a referenced topic contains a `<prolog>` or `<shortdesc>` element, metadata in a `<topicref>` element that contains `<prolog>` or `<shortdesc>` elements takes precedence.

A number of metadata elements are allowed inside the `<topicmeta>` element. Some of the most useful ones include:

- `<navtitle>`
- `<author>`
- `<copyright>`
- `<category>`

The `<metadata>` element, which can also contain most of these same metadata-related elements, is allowed inside the `<topicmeta>` element, as well.

The structure of a `<topicmeta>` element might look something like this:

```
<topicmeta>
<navtitle>Navigation title</navtitle>
<author>Author name</author>
<copyright>
  <copyryear year="20XX"/>
  <copyrholder>Company name</copyrholder>
</copyright>
<category>Category name</category>
</topicmeta>
```

Note:

For more information about the <topicmeta> element and the elements it contains, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Make a copy of the file `maps_bookmaps_samples/samples/_m_ducks_advanced_start.ditamap` and open it in your editor.

You should see content that begins like this (the file is much longer than what is shown here):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
  <title>Ducks</title>
  <topicref href="c_wild_ducks.dita">
    <topicref href="c_wild_duck_types.dita"/>
    <topicref href="c_wild_duck_species.dita"/>
    <topicref href="c_habitats.dita"/>
    <topicref href="t_watching_wild_ducks.dita"/>
  </topicref>
  <topicref href="c_domestic_ducks.dita">
    <topicref href="c_duckling_growth.dita"/>
    <topicref href="c_housing.dita"/>
  </topicref>
  ...
</map>
```

2. Inside the <map> element, after the <title> element, add a <topicmeta> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
  <title>Ducks</title>
  <topicmeta>
    <critdates>
      <created date="2016-03-25"/>
      <category>Sample</category>
    </critdates>
  </topicmeta>
  ...
</map>
```

The <topicmeta> element you just added establishes the following map-level metadata:

- The map was created on March 25, 2016
 - This is a sample map
3. Inside the <topicref> element that links to `c_duckdb.dita`, add a <topicmeta> element and add content to it as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
<title>Ducks</title>
...
<topicref href="c_duckdb.dita">
<topicmeta>
<audience type="user" experiencelevel="expert"/>
</topicmeta>
<topicref href="c_writing_about_ducks.dita"/>
...
</topicref>
...
</map>

```

The <topicmeta> element you just added establishes the following topic-level metadata:

- This section is intended for users of the product, as opposed to administrators or casual readers.
- This section is intended for expert users

4. Inside the <mapref> element that links to g_duck_glossorg.ditamap, add a <topicmeta> element and add content to it as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="ducks">
<title>Ducks</title>
...
<mapref href="g_duck_glossorg.ditamap" format="ditamap">
<topicmeta>
<searchtitle>Duck Glossary</searchtitle>
</topicmeta>
</mapref>
</map>

```

The <topicmeta> element you just added establishes the following metadata:

- When someone uses a search tool and locates this topic, the title should be “Duck Glossary” instead of the title in the topic.

5. Check your file maps_bookmaps_samples/samples/_m_ducks_advanced_start.ditamap against the sample file maps_bookmaps_samples/samples/_m_ducks_advanced.ditamap.

Adding metadata to a bookmark

One of the major advantages of using bookmarks rather than maps is their expanded metadata capabilities. Bookmarks can contain extensive metadata related to ownership, identification, copyright, workflow, and publication of a book. There are many more elements available for capturing metadata in more specific ways in bookmarks than in maps. This means that a set of topics collected in a

bookmap will be more semantically rich, more easily searchable, and better suited to publication than the same set of topics in a map.

You can add metadata to a bookmap in several places:

- Inside the <bookmap> element using <bookmeta> element
- Inside elements contained in the book's frontmatter and backmatter using the <topicmeta> element
- Inside the book's chapters, topic references, and map references using the <topicmeta> element

The <bookmeta> element in a bookmap can contain the same metadata elements that the <topicmeta> element in a map can contain, plus many more. Some of the most useful ones include:

- <authorinformation>
- <publisherinformation>
- <critdates>
- <bookid>

The <bookmeta> element is optional in the bookmap structure. If you use it, it must be placed after the <booktitle> element and before the <frontmatter> element. The structure of the <bookmeta> element might look something like this:

```
<bookmeta>
  <authorinformation>
    <personinfo>
      <namedetails>
        <personname><firstname>Author first name</firstname>
          <lastname>Author last name</lastname></personname>
      </namedetails>
    </personinfo>
    <organizationinfo>
      <namedetails>
        <organizationnamedetails>
          <organizationname>Company name</organizationname>
        </organizationnamedetails>
      </namedetails>
    </organizationinfo>
  </authorinformation>
  <critdates>
    <created date="1/1/2001"/>
    <revised modified="1/1/2016"/>
  </critdates>
  <bookid>
    <edition>4</edition>
    <booknumber>ID-9999</booknumber>
    <volume>2</volume>
```

```
</bookid>
</bookmeta>
```

This example also shows how the author information is much more sophisticated in bookmaps than it is in maps. Although the `<author>` element is available in both bookmaps and maps, the `<authorinformation>` element is only available in bookmaps. The `<authorinformation>` element can contain far more information tagged in a way that provides more semantic value.

The `<topicmeta>` element is available in both maps and bookmaps, but the bookmap structure allows it to be used in more places. For example, you can apply the `<topicmeta>` element to a preface in the book's frontmatter, or to a glossary in the book's backmatter. You can also apply the `<topicmeta>` element to a chapter, topic reference, or map reference in a bookmap. When used in a bookmap, the `<topicmeta>` element can contain all the same `<metadata>` elements that it can contain in a map, plus the `<authorinformation>` element.

Note: For more information about the `<bookmeta>` element and the elements it contains, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Make a copy of the file `maps_bookmaps_samples/samples/_b_ducks_advanced_start.ditamap` and open it in your editor.
2. Inside the `<bookmap>` element, after the closing tag of the `<booktitle>` element, add a `<bookmeta>` element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
...
</booktitle>
  <bookmeta>
    <author>Your Name Here</author>
    <bookrights>
      <copyrfirst>
        <year>2015</year>
      </copyrfirst>
      <copyrlast>
        <year>2016</year>
      </copyrlast>
    <bookowner>
      <organization>BookCo</organization>
    </bookowner>
  </bookrights>
</bookmeta>
</bookmap>
```

The `<bookmeta>` element you just added establishes the following bookmap-level metadata:

- You authored this book
- The initial copyright was in 2015, and was renewed in 2016
- The book was produced by BookCo

3. Inside the `<chapter>` element that links to `c_wild_ducks.dita`, add a `<topicmeta>` element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
  ...
</bookmeta>
<chapter href="c_wild_ducks.dita">
  <topicmeta>
    <category>Wild ducks</category>
  </topicmeta>
  ...
</chapter>
</bookmap>
```

The `<topicmeta>` element you just added establishes the following topic-level metadata:

- This chapter pertains to wild ducks

Frontmatter, backmatter, and booklists

The `<frontmatter>` and `<backmatter>` elements are optional in a bookmap. They contain information about the document that appears before or after the main body content, respectively. The `<frontmatter>` element must appear before the first `<chapter>` element but after the `<bookmeta>` element, if present. The `<backmatter>` element must appear before the `<reltable>` element but after the last `<chapter>` or `<appendix>` element, if present.

The `<frontmatter>` and `<backmatter>` elements are most commonly used to contain the `<booklists>` element. Booklists indicate to a DITA processor where to place automatically generated output, such as a table of contents or an index. Common elements inside the `<booklists>` element include:

- `<toc>`
- `<indexlist>`
- `<glossarylist>`

The `<toc>` element directs the DITA processor to generate a table of contents at that location in the map. The table of contents is generated from the hierarchical structure of the map. No additional information is required.

The <indexlist> element directs the DITA processor to generate an index at that location in the map. The index is generated from <indexterm> elements within the map and topics. If there are no <indexterm> elements within the map or topic files, no index entries will be created.

The <glossarylist> element can contain either one or more <topicref> elements linking to glossary entries or one or more <mapref> elements linking to glossary maps. Glossary entries and glossary maps may be used outside of a <glossarylist>, but grouping them within a <glossarylist> allows you to filter and format them more uniformly.

Note: For more information about the <frontmatter> and <backmatter> elements, see [frontmatter](#) and [backmatter](#) in the OASIS DITA Version 1.2 Standard.

[Video: The DITA booklists element](#)

Practice

1. Continue using the file maps_bookmaps_samples/samples/_b_ducks_advanced_start.ditamap.
2. After the <bookmeta> element, add a <frontmatter> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
  ...
</bookmeta>
<frontmatter>
  <booklists>
    <toc/>
  </booklists>
</frontmatter>
```

3. After the final <chapter> element, add a <backmatter> element and add content to it as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
  ...
</chapter>

<backmatter>
  <booklists>
    <glossarylist>
      <topicref href="g_duck_glossgroup.dita"/>
    </glossarylist>
  </booklists>
  <indexlist/>
```

```
</booklists>
</backmatter>
```

The `<glossarylist>` element indicates that the glossary comes from a specific file. The `<indexlist>` element indicates that the DITA processor should generate an index from the topics in the map.

Adding a navtitle to a bookmap

Occasionally, you may find that the title of a topic—particularly a reused topic—is not consistent with the titles of the other topics in a map. For example, all your task topics might use an infinitive form (“To print the...”), but the reused topic might use a present participle form (“Printing the...”). To make the title of a topic consistent in your text and table of contents, use the `<navtitle>` element to specify a more consistent or appropriate title.

In some older DITA maps and bookmaps, you might encounter `navtitle` as an attribute for `<topicref>` and other elements. This is an older form and its use is discouraged. Always use the `<navtitle>` element when creating new content.

Note:

For more information about the `<navtitle>` element, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Continue using the file `maps_bookmaps_samples/samples/_b_ducks_advanced_start.ditamap`.
2. Find the `<topicref>` element that links to `r_tnav.dita` and add a `locktitle` attribute and set it to “yes” as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
...
<topicref href="r_tnav.dita" locktitle="yes"/>
...
</bookmap>
```

Note: If you want the `navtitle` to be used, you must set the `locktitle` attribute to “yes.” Setting it to “no” or omitting it will result in the `navtitle` not being used.

3. Change the `<topicref>` element from an empty tag (“`<topicref ... />`”) to an open and close tag (“`<topicref...></topicref>`”)

```
<topicref href="r_tnav.dita" locktitle="yes">
</topicref>
```

4. Inside the `<topicref>` element add a `<topicmeta>` element and add content to it as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
...
<topicref href="r_tnav.dita" locktitle="yes">
    <topicmeta>
        <navtitle>The tNav command</navtitle>
    </topicmeta>
</topicref>
...
</bookmap>

```

Now that you’ve specified a navtitle, the phrase “The tNav command” will appear in the table of contents instead of the title in that topic, which is simply “tNav.”

5. Check your file `maps_bookmaps_samples/samples/_b_ducks_advanced_start.ditamap` against the sample file `maps_bookmaps_samples/samples/_b_ducks_advanced.ditamap`.

Exercise

1. Open the file `maps_bookmaps_samples/exercises/_b_cs101_start.ditamap` and use it add metadata, frontmatter, and backmatter to the bookmap for [Content Strategy 101](#), using the following bookmap structure as a guide:

Author: Sarah O’Keefe

Author: Alan S. Pringle

Publisher: Scriptorium Publishing Services, Inc.

Created: 2012

Revised: 2014

Table of Contents:

- **Chapter 1: Controlling technical communication costs**
 - The fallacy of low-cost documentation
 - Efficient technical content development
 - Reducing the cost of technical support
 - Better technical information, fewer support calls
 - More efficient support operations
 - Content collaboration across the organization
 - “Collaboration” ≠ “content free-for-all”
- **Chapter 2: Marketing and product visibility**

- Supporting marketing with technical content
 - Reinforcing the marketing message
 - When technical content contradicts marketing
- Increasing product visibility
 - Third-party books
- Building user community and loyalty
 - Extending the game experience into content
- **Chapter 3: Legal and regulatory issues**
 - Avoiding legal exposure
 - Meeting regulatory requirements
 - Information delivery
 - Technical standards

Index:

2. Check your file `maps_bookmaps_samples/exercises/_b_cs101_start.ditamap` against the sample `filemaps_bookmaps_samples/exercises/_b_cs101_advanced.ditamap`.

Adding a topichead to a bookmark

The `<topichead>` element allows you to specify a title-only entry in a map that applies to one or more topics. The specified title will appear in both the table of contents, if present, and as a header in the main body of the text. The `<topichead>` element must contain a `<navtitle>` element.

The `<topichead>` element is useful because it allows you to group topics together that are contextually related without implying a hierarchical relationship, and prevents you from having to create a container topic to create a title.

Note:

For more information about the `<topichead>` element, see the [OASIS DITA Version 1.2 Standard](#).

Practice

1. Continue using the file `maps_bookmaps_samples/samples/_b_ducks_advanced_start.ditamap`.
2. Add a `<topichead>` tag before the `<topicref>` referencing `t_create_table.dita` and add a closing `</topichead>` tag after the `<topicref>` referencing `t_modify_table.dita` as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmark PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmark.dtd">
```

```

<bookmap id="ducks">
...
<chapter href="c_duckdb.dita">
<topichead>
<topicref href="t_create_table.dita">
<topicref href="t_adding_entry.dita">
<topicref href="t_modify_entry.dita">
<topicref href="t_deleting_entry.dita">
<topicref href="t_queries.dita">
<topicref href="t_modify_table.dita">
</topichead>

```

3. Add a `<topicmeta>` element after the opening `<topichead>` element you just added and add a `<navtitle>` element to it as shown:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE bookmap PUBLIC "-//OASIS//DTD DITA BookMap//EN" "bookmap.dtd">
<bookmap id="ducks">
...
<chapter href="c_duckdb.dita">
<topichead>
<topicmeta>
<navtitle>Common tasks</navtitle>
</topicmeta>
<topicref href="t_create_table.dita">
<topicref href="t_adding_entry.dita">
<topicref href="t_modify_entry.dita">
<topicref href="t_deleting_entry.dita">
<topicref href="t_queries.dita">
<topicref href="t_modify_table.dita">
</topichead>

```

Lesson 4: Relationship tables

Objectives

- State the purpose of a relationship table.
- Create a relationship table and populate it with `<topicref>` elements.
- Identify the meaning of rows, columns, and cells in a relationship table
- Use the collection-type and linking attributes to modify relationships.

This lesson introduces relationship tables, then shows how to create a relationship table, how to populate it, and how to use the collection-type and linking attributes.

Additional reading

[DITA Style Guide, Relationship tables](#)

About relationship tables

Relationship tables specify how DITA topics in a map relate to one another. They can also define associations among DITA topics and non-DITA resources.

The information in the relationship table is most commonly used by a DITA processor to generate lists of related links. However, the information can be used in many other ways, depending on the capabilities of the output processor and the target output format.

Note:

Although the information is maintained in a tabular format and the containers are called “relationship tables,” the table itself is never displayed in output.

- Maintaining topic associations in a relationship table (rather than using embedded cross-references) ensures that the topics are reusable. If a topic contains embedded cross-references, the target topic for each cross-reference must also be available in the map, whether or not the topic is relevant to the map. (Then, of course, if that topic has cross-reference dependencies, they must also be present, and so on.) A relationship table eliminates these dependencies.
- A network of links among related topics can be maintained in a single location (the relationship table), rather than being distributed across all topics. This makes the maintenance of networks or webs of links much easier.

Relationship tables are contained in DITA maps and bookmaps. A map or bookmap can contain any number of relationship tables.









Each row in a relationship table contains a related set of content. For example, a row might contain references to all topics that relate to a specific system component, or all topics that are related to a particular operation.

Each column in the relationship table contains a similar type of information. The most common use of columns is to separate concept, task, and reference topic types. If you use additional topic types, you could define separate columns for those. In some cases, another column can contain links to external information, such as websites.

Note: Column order or relative position has no meaning in relationship tables.

Each cell in the relationship table contains one or more <topicref> elements that indicate linked topics in the map.

Here is a brief example of a relationship table.

concept	task	reference
 c_nourishment.dita	 t_feeding.dita	 r_food.dita  r_feeders.dita
 c_shelters.dita	 t_building.dita	 r_houses.dita  r_feeders.dita

The column heads indicate the types of information that are in each column. The first body row contains references to topics that are related to feeding; the second body row refers to topics related to housing. Two additional things to note about this example:

- A `<topicref>` element can appear in more than one row. For example, the `topicref` to `r_feeders.dita` appears in two different rows because it pertains to both feeding and housing.
- The cells in the relationship table can contain any number of `<topicref>` elements (including none).

When an output generator creates output from your DITA topics, it uses the relationship table to create links to related information.

- For all topics referenced in a row, the output generator creates links among those topics. Usually this takes the form of a list of links at the end of an output topic.
- If required (and if supported by the output generator), the list of links can be grouped and labeled by the columns defined in the relationship table.

Note:

For more information about the `<reltable>` element and the elements it contains, see the [OASIS DITA Version 1.2 Standard](#).

Creating a relationship table

A relationship table is contained in the `<reltable>` element. The `<reltable>` element is always a child of the `<map>` or `<bookmap>` element. A map or bookmap can contain multiple `<reltable>` elements.

The location of the `<reltable>` is different for maps and bookmaps:

- In a map, the `<reltable>` elements can occur anywhere in the map. However, it is usually placed at the end of the `<map>` (after the last of the `<topicref>` elements).
- In a bookmap, the `<reltable>` elements must occur as the last elements in the bookmap (that is, before the `</bookmap>` tag).

Each column in a relationship table contains a similar type of information. The types of information in each of the columns is defined by `<relcolspec>` elements in the relationship table header row (`<relheader>`).

In the relationship table's most common form, the `<relcolspec>` elements define columns for concept, task, and reference topics. However, a relationship table is not limited to those types.

Practice

1. Open the file `maps_bookmaps_samples/samples/_reltable_start.ditamap`.

This map is already populated with `<topicref>` elements.

2. At the end of the file, before the `</map>` tag, add a `<reltable>` element.

```
...
    </topichead>
    </topicref>

    <mapref href="g_duck_glossorg.ditamap" format="ditamap"/>

    <reltable title="Ducks reltable">
    </reltable>

</map>
```

Note that you can use the title attribute in the `<reltable>` element to help identify the purpose of the relationship table, but the title is never displayed on output.

3. Inside the relationship table, add a `<relheader>` element that contains three `<relcolspec>` elements.

```
<reltable title="Ducks reltable">
  <relheader>
    <relcolspec/>
    <relcolspec/>
    <relcolspec/>
  </relheader>
</reltable>
```

4. Add type attributes to each of the three `<relcolspec>` elements for concept, task, and reference.

```
<relheader>
  <relcolspec type="concept"/>
  <relcolspec type="task"/>
  <relcolspec type="reference"/>
</relheader>
```

Continue populating the `<reltable>` element in the next topic.

Adding rows to the relationship table

The rows in a relationship table contain `<topicref>` elements for related information. In the exercise, you will add references to topics that mention the “add” command in the fictitious Duck database.

[Video: DITA relationship tables](#)

Practice

1. Continue using the file `maps_bookmaps_samples/samples/_reltable_start.ditamap`.
2. After the `<relheader>` element, add a `<relrow>` element.

```
<reltable title="Ducks reltable">
  <relheader>
    <relcolspec type="concept"/>
    <relcolspec type="task"/>
    <relcolspec type="reference"/>
  </relheader>

  <relrow>
</relrow>
</reltable>
```

3. Inside the `<relrow>` element, add three `<relcell>` elements.

```
<relrow>
  <relcell>
</relcell>
  <relcell>
</relcell>
  <relcell>
</relcell>
</relrow>
```

Because we defined three `relcolspec` elements, we know that there are three columns in the relationship table.

4. In the first `<relcell>` element, add a `<topicref>` element; include an `href` attribute that contains the value `c_writing_about_ducks.dita`.

```
<relrow>
  <relcell>
    <topicref href="c_writing_about_ducks.dita"/>
  </relcell>
  <relcell>
</relcell>
  <relcell>
</relcell>
</relrow>
```

5. Add `<topicref>` elements with href attributes to the other two `<relcell>` elements, as shown here.

```
<relrow>
  <relcell>
    <topicref href="c_writing_about_ducks.dita"/>
  </relcell>
  <relcell>
    <topicref href="t_adding_entry.dita"/>
  </relcell>
  <relcell>
    <topicref href="r_add.dita"/>
  </relcell>
</relrow>
```

The relationship table now indicates that there is a relationship among the three topics indicated by the href attribute values.

Advanced relationship tables

The collection-type attribute

The DITA specification states that when a cell in the relationship table contains multiple `<topicref>` elements, the generated topics indicated in that cell do not link to one another. However, if the collection-type attribute on the `<relcell>` element has the value “family”, all of the topics referenced in in that `<relcell>` element will link to one another.

In the following example, topics A and B will link to each other:

```
<relcell collection-type="family">
  <topicref href="a.dita"/>
  <topicref href="b.dita"/>
</relcell>
```

The linking attribute

Typically, when two topics (topic A and topic B) are found in the same row of a relationship table (but in different columns), the output topic A will contain a link to topic B, and topic B will contain a link to topic A.

```
<relrow>
  <relcell>
    <topicref href="a.dita"/>
  </relcell>
  <relcell>
    <topicref href="b.dita"/>
  </relcell>
</relrow>
```

However, in some cases this behavior is not necessary or is not wanted.

Topic B might be a common task or a glossary term that is associated with a many concepts. We might want topic A (and other topics) to link to topic B, but we don't want topic B to link back to topic A (or any of the other topics). In this case, we modify the <topicref> element for topic B by adding the linking attribute and setting it to the value "targetonly".

```
<relrow>
  <relcell>
    <topicref href="a.dita"/>
  </relcell>
  <relcell>
    <topicref href="b.dita" linking="targetonly"/>
  </relcell>
</relrow>
```

Conversely, topic A might be a "landing page" for a help system; we may want to use it as an entry point that links to a number of different topics, but we don't want the topics to refer back to the page (for whatever reason). In this case, we add the linking attribute to the <topicref> for topic A and use the value "sourceonly".

```
<relrow>
  <relcell>
    <topicref href="a.dita" linking="sourceonly"/>
  </relcell>
  <relcell>
    <topicref href="b.dita"/>
  </relcell>
</relrow>
```

Labeling columns

When generating output, the DITA Open Toolkit can use the topic types in the <relcolspec> to group sets of related links (typically into sections labeled "Concept", "Task", and "Reference"). Other output generators can follow this same practice. You can override this labeling by using a <title> element in the <relcolspec> element







```
<relcolspec type="task">
  <title>Referenced in these tasks</title>
</relcolspec>
```

The <relcolspec> element can also contain a <topicref> to a topic that provides more information about the column itself. For a full description of the contents of the <relcolspec> element, see <http://docs.oasis-open.org/dita/v1.2/os/spec/langref/relcolspec.html>

Linking to external resources

A relationship table can contain links to external resources. To do this, the `<relcolspec>` element and the contents of the `<relcell>` element are slightly different than what has been shown before.

In this screen shot of a relationship table, the fourth column contains links to an external resource.

concept	task	reference	Web links
 c_nourishment.dita	 t_feeding.dita	 r_food.dita  r_feeders.dita	 http://www.example.com <i>scope="external" format="html"</i> <div> Navigation Title: A duck food source.</div>

Because the fourth column is not pointing to DITA resources, the `<relcolspec>` element has no type attribute. However, it includes a `<title>` element to define the purpose of the column.

```
<relcolspec>
  <title>Web links</title>
</relcolspec>
```

In the `<topicref>` element, the `href` attribute indicates the URL, the `scope` attribute is “external”, and the `format` attribute is “html”. In addition, the `<topicref>` includes a `<navtitle>` element so that the link is more reader-friendly:

```
<relcell>
  <topicref href="http://www.example.com" scope="external" format="html">
    <topicmeta>
      <navtitle>A duck food source.</navtitle>
    </topicmeta>
  </topicref>
</relcell>
```

Best practices for relationship tables

Consider the number of columns

Although the relationship table can contain any number of columns, too many columns can make the relationship table difficult to edit.

If multiple columns cannot be avoided, one alternative is to create multiple relationship tables with different columns for different sets of relationships. As stated earlier: column order or relative position has no meaning in relationship tables.

Consider the number of `topicref` elements in a cell

The `<relcell>` element can contain any number of `<topicref>` elements, however a large number of `<topicref>` elements in a `<relcell>` gets to be very difficult to read (whether viewed in text or author mode of your DITA editor).

If you find a `<relrow>` is getting crowded, create one or more new `<relrow>` elements and distribute the `<topicref>` elements among the rows.

Reuse and relationship tables

A map can contain any number of relationship tables. When the relationship tables are processed, the contents of the relationship tables are usually seen as a single collection (depending on the output processor and the intent of the relationship table). Thus, if you have a block of topics that are reused across a number of different maps, you can create a submap of those topics and create a relationship table that is specific to those topics. When the submap is included with a `<mapref>`, the relationship table will be included, also.

You can apply filtering to relationship tables, using DITA filtering attributes. Individual rows in the relationship table can be filtered, or you can divide rows among multiple relationship tables and then filter on the tables.

Course VI. Introduction to Reuse in DITA

Lesson 1: Introduction to reuse

Objectives

- Demonstrate the advantages of reusing content
- Analyze content for reusability
- Explain the types of reuse available in DITA

This lesson shows how reuse in DITA can improve the content development process and reduce costs.

Additional reading

[DITA Style Guide, Content re-use definition](#)

[DITA Style Guide, Re-use and the DITA Maturity Model](#)

[Content Reuse, contributed by Pam Noreault, Tracey Chalmers, and Julie Landman](#)

Housekeeping and sample files

Download [reuse_basic_samples.zip](#) now. It contains the sample files for the entire Introduction to reuse in DITA course. Extract the contents and put them in a directory that you can access easily.

Inside the samples folder, you will find the following sub-folders:

- conrefs
- reusing_topics_and_maps
- writing_for_reuse

Each lesson will instruct you on which folders and files to use for the exercises. Save your file as you complete each step to avoid losing your work.

Create a local copy of each file to work in as you complete the lessons. That way, if you reach a point where your working file doesn't match the examples, or is broken for any reason, you can make a fresh copy and resume your work or start over.

In the instructions and examples, we show you the DITA code for the sample files. Most DITA editors have auto-complete or other similar features to guide you through the process of adding elements (for example, if you type the opening tag of an element, most DITA editors will automatically add the closing tag for you). Therefore, you will probably not need to create every piece of code from scratch as you work.

Reasons for reuse

Reuse—storing content in a single source and using it in multiple places where relevant—is one of the benefits of maintaining content in a structured form such as DITA. Reusing content offers the following advantages:

- Reduces replication. Reusing content eliminates the need to copy and paste information, or to write nearly identical content multiple times in multiple locations. This saves time and increases content accuracy.
- Reduces costs. By reusing content, you reduce the cost of time that would have been spent duplicating information manually. This makes content creation faster and eliminates multiple reviews. Reuse reduces localization costs. Replicated content is translated separately every time it appears, but a single source of content that is reused is only translated once.
- Increases consistency. Every time you manually duplicate content, you introduce the risk of human error. Having multiple copies of the same information also increases the risk that the content will be updated in some places but not in others, or that the various copies will become slightly different from each other over time. Maintaining a single source means that the reused content will always be up-to-date and consistent wherever it is used.
- Allows product- or customer-specific output. You can combine different reusable topics in different maps for different products or customers. For example, you might have a series of product manuals that contain some information that's identical and other information that's more product-specific. Rather than writing a separate manual for each product, you can combine product-specific topics with common topics in a map for each product. This allows you to write each of the common topics only once.

Implementing reuse is a long-term investment. You will usually see the benefits of reuse in update or maintenance costs. Most of these benefits will occur after, not during, the initial release.

Analyzing content for reuse

Most content contains some information that can be reused. An analysis of your content can help you find what is reusable. You might find entire topics that are reusable, or you might find individual elements (such as paragraphs, notes, or list items) that are reusable.

When analyzing your content you might find:

- No match. If chunks of content do not match each other, you don't have a case for reuse.
- Exact match. If you find chunks of content (or even entire topics) that are identical, you should reuse them.
- Fuzzy (inexact or partial) match. If you find content that almost matches or is similar to other content, you can probably benefit from reusing it, with some minor changes. Here are some examples of content that would be considered a fuzzy match:

- **Naming:** Content can be almost the same, except for the occurrence of product or device names, model numbers, or company names.
- **Different processes, definitions, or details:** You might have several product manuals that are mostly the same except for certain pieces of information—for example, a different set of installation instructions for one version of a product, or some additional safety information for another.
- **Locale:** Different units of measure might be used in versions of content intended for different locales.
- **Different order:** The same content might appear in a different sequence from one deliverable to another.
- **Subset of content:** A shorter version of your content—for example, a quick guide version of a user manual—constitutes a partial match. Common content between the shorter and longer versions should be reused.
- **Specifics:** Your content might be identical except for a few pieces of information (such as the size or number of screws). In many cases, simply eliminating the specifics makes your content reusable.

Sometimes it makes more sense to rewrite content than to try to reuse it as-is. This is particularly true when two pieces of content are almost the same, except for some differences in usage or phrasing (or both). If two pieces of content convey basically the same information, but are written differently, consider merging them into a single reusable chunk (or choose the version that works best in all locations).

Types of reuse

DITA offers the following reuse facilities: topics and maps, fragments, variables, and filtering.

Topics and maps: Topic- and map-level reuse is one of the most straightforward ways to reuse content. You can achieve this type of reuse by referencing the same topic in more than one map. Such topics might include boilerplate safety information or common installation instructions that apply to multiple products. Similarly, you can reuse one map in multiple places. For example, the map for a product datasheet might be published individually, and also as part of a larger bookmap for that product's user manual.

Fragments (conrefs): Fragment-level reuse involves pulling shared pieces of content (usually elements) from a common source into a topic. Examples of the shared content include paragraphs, lists or list items, tables, or sections. In DITA, the content reference (or conref) mechanism implements fragment-level reuse. Fragment-level reuse can also be used in DITA maps.

Variables (keys): In variable-level reuse, you add a placeholder to your content that is replaced by a piece of text when you generate your output. The replacement text varies depending on the

circumstances. Variable-level reuse is used for inline content, such as company names, product names, URLs, or filenames.

Filtering (conditions): Filtering allows you to selectively remove certain pieces of content, based on different conditions, to create product- or customer-specific deliverables. For example, when documenting two similar products, a basic version of the product does not have features that are available in a more advanced version. Filtering allows you to create a single piece of content containing all features, then filter out the advanced features when using the content for the basic version.

This course covers topic- and map-level reuse and fragment-level reuse. Variables and filtering will be addressed more thoroughly in future courses.

Lesson 2: Creating reusable topics

Objectives

- Write a reusable topic
- Demonstrate why inline replacements can cause problems in translation
- Write paragraphs, steps, and list items that are reusable

This lesson shows how to write topics and elements so that they are reusable.

Additional reading

[DITA Style Guide, Stem sentences, glue text, and other transitional information](#)

[DITA Style Guide, Writing for re-use](#)

[DITA Style Guide, Re-use guidelines](#)

Writing for reuse

When writing for reuse your content must be:

- Consistent
- Context-free
- Generalized

By separating specific information from common information wherever possible, you can make sure that your common chunks of content are generic enough to be reused wherever needed.

To make your content consistent:

- Keep topics short and granular; this maximizes their reuse potential. A short topic with only one heading level can be reused in more places than a long topic that covers large amounts of information and contains multiple sections.

- Establish a style guide for tagging and usage (or, at a minimum, agree on usage and set aside some time to document that agreement as the basis for a future formal style guide).
- Use a consistent voice and specific vocabulary and terminology to make your topics more reusable. Limit your use of pronouns (particularly gender-specific ones) and avoid idioms and colloquialisms. A consistent voice will not only help with reuse, but also with localization.

To make your content context-free:

- Do not assume anything about information that comes before or after the topic. Avoid using words such as “previous,” “next,” “earlier,” and “later.”
- Do not assume the type of the document in which the content is used. Avoid using phrases such as “in this chapter” or “in this section.”
- Do not use inline cross-references. In your version of the content, the cross-reference target exists, but it might not be in the next map where your content is reused. Instead, use DITA relationship tables to automatically generate links to related topics.

To make your writing more generalized:

- Avoid using keywords or product dependencies (wherever possible). For example, if the same set of instructions can be used with multiple products, eliminate product-specific references.
- Don’t be too specific unless it’s crucial. Avoid including gratuitous modifiers, such as number, size, or color, in your content, since they might not always be relevant as products change over time.

To strike a balance between making content generic for reuse and including necessary specifics, consider adding a table or list to the beginning of each deliverable that identifies certain product information. By providing that information up-front, you can then remove keywords from the content and rely on generic terms.

Generic content will not only help facilitate reuse, but will also better prepare you for localization. Reusing content also means that you will have fewer topics overall to translate, which will help reduce localization costs.

Reuse and translation

When you first learn about the reuse mechanisms in DITA, it might be tempting to use them in any number of situations. However, it’s wise to proceed with some caution, especially if your content will be translated. And even if your content is not being translated today, assuming that it will be translated someday will avoid problems when “someday” comes.

Every language has its own rules of grammar and syntax. Just because replacing or inserting a piece of text works in English does not mean that it will work in another language.

Some languages require words to take a different form depending on whether they are the subject or object of the sentence. Some languages have grammatical gender, and so words with gender can affect words that are related to it in the sentence. The plural forms in some languages are quite different from the English constructs for zero, one, and more than one object.

It would be difficult to describe the rules for every language, but we can use some English examples to illustrate why substituting words isn't a good idea when your content will be translated into other languages with other rules.

Take, for example, this simple sentence:

Casey ate an orange.

What if we were to substitute another word for orange?

Casey ate an apple.

In this case, the substitution works because both words begin with a vowel. But what if we substituted a word that begins with a consonant?

Casey ate an tomato.

Now we have a problem. In English, we use the article a when the word starts with a consonant sound and the article an when the word starts with a vowel sound. Other languages have rules similar to this; some are even more complex.

“Knowledge is knowing that a tomato is a fruit. Wisdom is not putting it in a fruit salad.” – Miles Kington

Other substitutions can also cause problems in our sample sentence. So far, our example has used a singular noun; for example:

Casey ate an apple.

What if we substituted the plural of “apple”?

Casey ate an apples.

You're probably getting the idea. English has rules that are broken easily by substituting words within sentences. Other languages have other rules that are also broken easily by substituting words within sentences. And just because you're following the rules for English doesn't mean you're not breaking the rules for another language.

You probably don't know all the rules for all the languages your English content will be translated into. So, you should not use replaceable text within sentences to substitute parts of the sentence. If you use replaceable text on parts of a sentence, you will have unpredictable results when the content is translated.

Reusing paragraphs

Whole, grammatically independent paragraphs can be reused safely within the same context. This includes the <p> element and any element that grammatically is a paragraph, even if it has only one sentence. Let's take a closer look.

Grammatically independent

A paragraph that is grammatically independent doesn't rely on anything outside the paragraph to complete its meaning. For example, if pronouns within the paragraph refer to something outside the paragraph then it would not be grammatically independent. Substituting pronouns for nouns to make a paragraph less specific does not make it a good candidate for reuse. Substituting less specific nouns, however, is a good strategy. For example:

Do not use this

The Compu-Master 5000 is equipped with an attractive protective case. To avoid losing your computer, please do not paint or wallpaper the case to match your décor.

Use this

Your computer is equipped with an attractive protective case. To avoid losing your computer, please do not paint or wallpaper the case to match your décor.

Here is another example:

Do not use this

This may not appear because it requires an additional license. Please call our office for more information.

Use this

The database access module may not appear because it requires an additional license. Please call our office for more information.

Within the same context

Some words have multiple meanings that still make sense in English even if they're not being used in their original context. Take the word "key" as an example. It might refer to a small piece of cut metal that's used to open a lock, or it might refer to something on a rotating shaft that keeps another machine component such as a gear rotating along with the shaft. The word "key" also has other meanings related to maps or cryptography.

If we were told to "examine the key for defects", it might make sense to us in English in all of those contexts. In our minds we substitute a different concept for the word "key" depending on the context. But we can't count on those different concepts sharing the same word in other languages.

This can be a tricky one to spot, because we usually substitute the correct concepts in our minds without even being conscious of it. So here's another example. In English, the word "battery" could refer to something in an automobile or to something in a watch. Both supply electrical current, so we might think that it's the same concept and context. But when we stop and think about it, a watch is not at all the same context as an automobile and their batteries are very different in their construction. So

it's not safe to reuse between those two contexts, even if there are similarities in the concepts. And in fact, different words are used for those two kinds of batteries in some languages.

The main point to keep in mind is that in other languages an entirely different word might be used for different concepts or contexts, so reusing in English across different contexts is not helpful for translated content.

Reusing steps

As with paragraphs, when you reuse `<step>` elements, the contents of the `<cmd>` element must be grammatically independent and should be reused in the same context. For example:

Do not use this:

```
<step><cmd>Close it.</cmd></step>
```

Use this:

```
<step><cmd>Close the window.</cmd></step>
```

Structured parts of steps

Similarly, you can reuse other parts of the `<step>` element, such as `<stepxmp>` (step example), `<stepinfo>`, `<choices>`, `<tutorialinfo>`, and others, as long as they are grammatically independent and reused in the same context.

Reusing list items

As with paragraphs and `<step>` element contents, you can reuse `` (list item) elements, as long as they're used in the same context and are grammatically independent (that is, they do not use pronouns that refer to something outside the list item). But there is an additional caution when a list is introduced by an incomplete sentence (a “stem sentence”).

When an incomplete sentence introduces a list, the list items must fit with the grammar of the stem sentence. List items that are written to fit with a stem sentence might not work well with other stem sentences, making them poor candidates for reuse.

When creating reusable items for a list, introduce them with a complete sentence or independent clause (or write the list so that the introduction is not necessary), and make sure that the contents of the list item are also independent clauses. (An independent clause stands on its own and expresses a complete thought.)

Not reusable:

```
<p>Before using your computer, you should...</p>
<ol>
  <li>Connect the monitor.</li>
  <li>Connect the mouse.</li>
```

```
<li>Connect the keyboard.</li>
</ol>
```

Reusable:

```
<p>Before using your computer, make sure the following items are connected:</p>
<ol>
  <li>Monitor</li>
  <li>Mouse</li>
  <li>Keyboard</li>
</ol>
```

Storing your reusable content

It's good practice to maintain all your reusable information (topics, elements, and inline content) in a set of common, shared folders and topics. These are called “warehoused” folders and topics. All content creators should be aware of the warehoused topics and should understand the rules for modifying the topics.

If you reuse content from an arbitrary topic, the author of that topic might not know that you are reusing their content. If they do not know, they could possibly change or even delete their content, which could make your content incorrect. At worst, their changes could make your information dangerously wrong.

Keeping all reused content in warehoused folders and topics gives all content creators tighter control on the information:

- You can manage dependencies from one topic to another.
- You can manage changes and deletions to warehoused content.
- You can communicate those changes once they have been made.
- You can ensure that warehoused information does not contain links to other topics (which would create additional dependencies when reusing them)

Some component content management systems (CCMS) may have specific requirements for where reused content must be stored. For example, some systems require that all content included in a map, including content included solely for reuse, be stored in a folder beneath the folder that holds the map. Check the requirements of your CCMS if you're in doubt.

Other CCMS options provide access control for warehoused content.

Practice

1. Open the file `reuse_basics_samples/writing_for_reuse/c_documenting_ducks_start.dita`.
2. Rewrite the following content in this topic to make it reusable:

In the previous chapter, you learned about ownership and care of domestic ducks. In this chapter, you'll learn how to document valuable information about your ducks using the Magic Mallard duck database.

Every duck owner should keep track of her ducks' growth. With the Magic Mallard database, you can create an entry for each duck. Each entry contains a chart where you can type in the duck's height, weight, and wingspan week by week.

To add an entry for a duck to your database, type add entry followed by the name of the duck in parentheses into the command line and press Enter.

3. Check your rewritten content against the file reuse_basic_samples/writing_for_reuse/c_documenting_ducks.dita. (Results will vary slightly based on rewriting choices.)

Lesson 3: Reusing topics and maps

Objectives

- Demonstrate reuse at the map and topic levels
- Add a reusable topic to a map
- Create a reusable map
- Add a reusable map to a map

This lesson shows how to reuse topics and maps in DITA.

Additional reading

[DITA Style Guide, Embedded topics and ditamaps](#)

[DITA Reuse Strategies](#)

[Reuse strategies and the mechanisms that support them](#)

Reusable topics and maps

In the most basic form of reuse, you can reuse a DITA topic in any number of maps.

It is important that each reusable topic follows the guidelines presented in the previous lesson. That is, each topic should be:

- Consistent
- Context-free
- Generalized

As described in the LearningDITA course on *Using DITA maps and bookmaps*, you use the DITA `<topicref>` element in a map or bookmap to include a topic in a map.

```
<topicref href="filepath/filename.dita"/>
```

If your topic meets all of the reusable guidelines, you should be able to re-use the topic in just about any type of map, at just about any level in the hierarchy.

Reusing a topic in the same map

The phrase “reusing a topic” usually means using one topic in many different maps. However, there are times when you might want to reuse the same topic several times in the same map.

As an example, consider a DITA map for some course material where the content introducing the tests is always the same. Rather than create separate topics for each introduction, you could create a shared, reusable introduction, then reuse it each time before each test.

However, when you reuse the same topic multiple times in one map, you need to use the `copy-to` attribute in any `<topicref>` element that references the common topic file.

```
<map>
  ...
  <topicref href="my_topic.dita" copy-to="first_copy.dita"/>
  ...
  <topicref href="my_topic.dita" copy-to="second_copy.dita"/>
  ...
</map>
```

The main reason for using `copy-to` is to ensure that when you create a cross-reference or some other link to the shared topic, that the cross-reference points to the correct instance of the file.

It is a best practice to use the `copy-to` attribute on all `<topicref>` elements that reference the shared topic, rather than skipping the first `<topicref>` to the shared topic. If you reorganize your content, you might lose track of the fact that the topic is shared.

If your main map or bookmap references several submaps and each of those submaps reference the same topic, you must still use the `copy-to` attribute in those `<topicref>` elements. When you generate output, preprocessing pulls all submaps into the main map to create a complete, merged map. The merged map is then used to generate output.

You can use the `<navtitle>`, `<linktext>`, and `<shortdesc>` elements to provide unique title and description information for the copied topic:

```
<map>
  ...
  <topicref href="my_topic.dita" copy-to="first_copy.dita">
    <topicmeta>
```

```

        <navtitle>Name for first copy</navtitle>
        <linktext>Name for first copy</linktext>
        <shortdesc>Modified information about the first copy of the topic.</
shortdesc>
    </topicmeta>
</topicref>
...
<topicref href="my_topic.dita" copy-to="second_copy.dita">
    <topicmeta>
        <navtitle>Name for second copy</navtitle>
        <linktext>Name for second copy</linktext>
        <shortdesc>Modified information about the second copy of the topic.</
shortdesc>
    </topicmeta>
</topicref>
...
</map>

```

Reusing maps

You can create reusable maps for content that occurs in multiple published documents. For instance, many components of a preface might be used in all prefaces for a set of documents. A reference section might be necessary in multiple books; glossaries and appendixes might also appear in many related publications.

For these repeating sets of information, you can create a map (often called a “submap”), then reuse the submap in other maps or bookmaps.

Practice

1. Open the file `reuse_basic_samples/reusing_topics_and_maps/_m_breeds.ditamap`. This is an example of a submap (named `_m_breeds.ditamap`) for reference information on duck breeds. Using the “_m_” helps to group all the maps together in the file system and displays them at the top of the file list so that you can find them quickly.

```

<map>
    <title>Duck breed reference</title>
    <topicmeta>
        <category>reference map</category>
    </topicmeta>
    ...
    <topicref href="r_mallard.dita"/>
    <topicref href="r_muscovy.dita"/>
    ...
</map>

```

2. Open the file `_m_keeping_ducks_start.ditamap`. To reuse the submap `_m_breeds.ditamap` in this map, add it using the `<topicref>` element with `format="ditamap"`.

```

<map>
  <title>Keeping ducks</title>
  ...
  <topicref href="_m_breeds.ditamap" format="ditamap"/>
  ...
</map>

```

Alternatively, you can use the <mapref> element to reference the submap:

```

<map>
  <title>Keeping ducks</title>
  ...
  <mapref href="_m_breeds.ditamap"/>
  ...
</map>

```

A reusable map itself can contain references to other reusable maps (using <mapref> or <topicref format="ditamap">).

A note about bookmaps and reusable submaps: When using a bookmap, you cannot add a <mapref> (or <topicref format="ditamap">) element at the top level (<frontmatter>, <chapter>, <part>, <appendixes>, <appendix>, and <backmatter>). You can reuse maps from within bookmaps, but only from inside the top-level containers. Therefore, you cannot create a submap that contains one of these top-level containers.

3. Check your work against the file reuse_basic_samples/reusing_topics_and_maps/_m_keeping_ducks.ditamap.

Merging maps

In the previous examples, the submap (_m_breeds.ditamap) contains <title> and <topicmeta> elements. It's a good idea to use these elements to provide information to other writers to describe the purpose of the submap and to provide metadata for searches. But note that these are not retained in final processing.

When you generate output from a map or bookmap, all submaps are merged into a final map, which is used to generate your output.

This table lists some of the more common elements you might use in a submap and what happens to them when the submap is merged in to a final map.

Element	Merged map result
<title>	Deleted
<topicmeta>	Deleted
<topicref> to DITA topic	Copied exactly
<topicref format="ditamap">	Replaced with content from href attribute

Element	Merged map result
<mapref>	Replaced with content from href attribute
<topichead>	Copied exactly
<topicgroup>	Copied exactly
<reltable>	Copied exactly

Practice

1. Open the following submap files:
 - reuse_basic_samples/reusing_topics_and_maps/_m_wild_ducks_start.ditamap
 - reuse_basic_samples/reusing_topics_and_maps/_m_domestic_ducks_start.ditamap
2. Reuse the topic file c_duck_safety.dita by adding it as the last topic in each of the submap files you just opened.
3. Save and close both of the submap files.
4. Open the map file reuse_basic_samples/reusing_topics_and_maps/_m_ducks_start.ditamap. The submap files you just edited are reused in this map. Therefore, you should be able to see the topic file you added (c_duck_safety.dita) in each submap within the main map.
5. Check your work in your main map file against the map file reuse_basic_samples/reusing_topics_and_maps/_m_ducks.ditamap.

Lesson 4: Using content references

Objectives

- Define the limits on element reuse
- Demonstrate element reuse with content references
- Explain element requirements for content references

This lesson shows how to create and use content references (conrefs) to achieve fragment-level reuse.

Additional reading

[OASIS DITA Version 1.2 Standard, The conref attribute](#)

[DITA Style Guide, The content reference \(conref\) attribute](#)

[Using content references in DITA](#)

What is a content reference?

A content reference (conref) enables you to reuse an element from one topic in one or more other topics. In DITA, the conref provides fragment-level reuse.

In typical use, you create a shared piece of content (such as an element), and then reuse that content in other documents where it is required. One of the best examples of a conref is a DITA <note> element. You can create the <note> element once, make sure it is approved by all whom it affects (including—possibly—your legal department), and make it available through a warehouse topic. Then any writer who needs to use the <note> element can use a conref to add it to their content where it is appropriate.

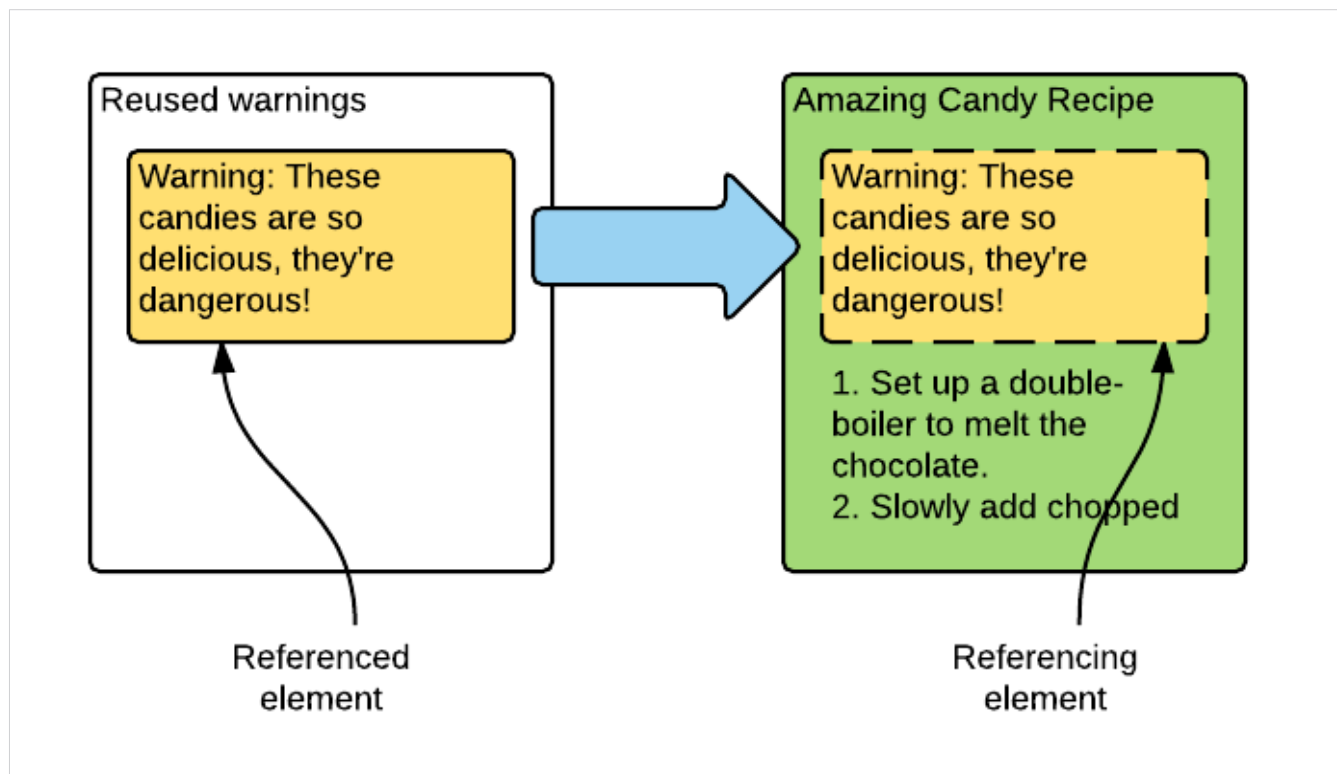
When pieces of content are repeated in multiple topics or publications, using conrefs can make your content easier to maintain. Unlike copy and paste, where content is duplicated, conrefs insert reused elements by reference into a topic.

Conref basics

A conref has two parts:

- The referenced element, which contains the content you want to reuse. This element must have an ID attribute.
- The referencing element, which marks where the reused content should be inserted

Note: If the element you want to reuse does not have an ID, your DITA editor or content management system might prompt you to create an ID for the reused element as one of the steps in creating the conref. An ID must start with a letter, number, or underscore.



Tip: In writing about content references, the word conref is often used as a verb to mean “reuse an element by using the conref attribute.”

Conrefs are resolved at publishing time by an output generator. Many DITA editors can also resolve conrefs while you are editing a topic, which helps you see what the resolution of the conref will look like.

To resolve a conref, the output generator replaces the referencing element with the referenced element and all of its content. This includes any elements contained within the referenced element. For example, if a referenced `<p>` element contains a `<uicontrol>` element, the referencing `<p>` element is replaced by the referenced `<p>` element, including the `<uicontrol>` element contained within it. Similarly, a conref to an `` element includes all the `` elements that are contained by the referenced `` element.

When the original content changes, those changes appear everywhere that content is referenced the next time it is published.

The conref attribute

A conref directly references a reused element using three pieces of information:

- The path to the topic file containing the referenced element
- The ID of the topic that contains the referenced element
- The ID of the referenced element

The path to the file and the topic ID are separated by a pound sign (#); the topic ID and the element ID are separated by a slash (/).

Note: If the referenced element is in the same file as the referencing element (not a common occurrence), the path to the file can be omitted, but the pound sign is still required.

For example, a warehouse topic (stored in the file `../warehouse/warnings.dita`) contains a reusable `<note>` element.

```
<topic id="warehouseWarnings">
  ...
  <note type="danger" id="hotWarning">Surfaces are hot.</note>
  ...
</topic>
```

To reuse this `<note>` element in another topic, create an empty `<note>` element at the appropriate location, and use the conref attribute to identify the element to be reused.

```
...
<note conref="../warehouse/warnings.dita#warehouseWarnings/hotWarning"/>
...
</note>
```

The contents of the conref attribute might look a bit intimidating or hard to remember. Most DITA authoring tools help you fill in the conref attribute.

The following table shows some examples of how the conref attribute value is formatted in various situations. You would replace the placeholders in these examples:

- topicID is the ID of the topic that contains the reused content
- targetID is the ID of the reused element
- folder is a directory in a local file system
- file.dita is the name of a DITA file
- http://example.com is the name of a network computer

Target location	Conref attribute value
In the same topic	#topicID/targetID
In a topic in a local file system	folder/file.dita#topicID/targetID
In a topic at a network location	http://example.com/file.dita#topicID/targetID

Element requirements for conrefs

In DITA, some elements are not valid without other elements inside them. For example, in a task, the `<step>` element requires a `<cmd>` element within it. When elements have required contents, those elements must be present in the referencing element, even if the referencing element will be replaced when the conref is resolved. To satisfy this requirement, you insert empty required elements inside the referencing element.

In the following example, the empty `<cmd>` element is a necessary part of the referencing element for the step.

```
<task id="referencing-element">
  <title>A task that reuses a step</title>
  ...
  <step conref="../warehouse/task-reuse.dita#warehouse-task/reusedStep">
    <cmd/>
  </step>
  ...
</task>
```

This example shows the referenced `<step>` element.

```
<task id="warehouse-task">
  <title>A warehouse task topic</title>
  ...
  <step id="reusedStep">
    <cmd>Lock out and tag the power source.</cmd>
  </step>
  ...
</task>
```

Reusing across topics

When publishing a topic that reuses content from another topic, the content references can be resolved only if the referenced elements are reachable through the publication map. For example:

- If the topics containing the referenced element and referencing element are in the same map then conrefs between those topics can be resolved through the context of that map.
- If the topics containing the referenced element and referencing element are in separate maps but those maps are part of the overall map structure of the same published map, the conrefs between those topics can be resolved through the context of the publication map.

Ordinarily, if a topic is included by a `topicref` in one of the maps, it will be published along with all the other topics. But you wouldn't want to publish a warehouse of reusable content along with your publication. To mark resources of this sort that aren't intended for publication, set the `topicref`'s `processing-role` attribute to `resource-only`.

In the following example, `task.dita` will be published, but `warnings.dita` will not. However, `warnings.dita` is available for resolving references from `task.dita` or any other topics in the map.

```
<map>
...
  <topicref href="warehouse/warnings.dita" processing-role="resource-only"/>
...
  <topicref href="topics/task.dita"/>
...
</map>
```

Note: When working in a component content management system, the system might resolve conrefs and other direct references that appear in a topic you're viewing or editing, even without a common context set through a map. But, depending on your publishing configuration, those references might not resolve when the topic is published. The best practice is to have topics with the referenced element and referencing element in the same map or one of its submaps.

Attributes of content references

It's fairly common for reused content to use attributes. For example, a `<note>` element uses the `type` attribute to specify the note type. Or, you might `conref` an element that has conditional filtering values set. When you `conref` an element that contains attributes, which attributes are used in the fully resolved element?

The answer is that the attributes from both the referenced element and the referencing element are used. If there are conflicts (both elements specify the same attribute), the attribute in the referencing element is used.

There are two exceptions:

- To force the conref to use an attribute value in the referenced element, use the special attribute value “-dita-use-conref-target” in the referencing element.
- If the referenced element has an xml:lang attribute, it is used, rather than xml:lang in the referencing element.

Best practices for using conrefs

Use warehouse folders and topics

Avoid “spaghetti” documentation by maintaining all your reusable components in warehouse folders and topics.

Organize the warehouse topics in reusable maps and incorporate those maps into your maps with `<mapref processing-role=”resource-only”>`.

Agree to naming conventions for files and IDs

Although most DITA editors will automatically assign unique IDs to elements that might be reused, it is often preferable to give human-readable IDs to reusable elements. The down side to this is that there’s a much greater chance of ID collisions. You can do two things to help avoid these collisions

- Create a naming convention for the IDs so that there’s a much lower chance of collisions
- Establish a clearing house for all ID values, so that you can quickly determine if an ID is unique. This might be as simple as a shared spreadsheet or even a text document, but can also be much more complicated, if necessary.
- Use a map in the warehouse folder to validate newly added reusable elements. The validator will flag duplicate IDs. Do this before committing content updates to your CCMS or source control.

Make sure your topics and elements are reusable

As was explained in the lesson on creating reusable topics, what might seem to work well when using conrefs in English content might not work well when the content is translated.

- Don’t use conref with the `<ph>` element to replace parts of a sentence.
- Use extreme caution when using conref to reuse whole sentences.
- When you do conref, do it with whole block elements that meet the following requirements:
 - The reused content is grammatically independent.
 - The referenced element and referencing element use the same context.

Communication is everything

Before you make changes to warehouse topics, make sure you know where the topics and their contents are used. Some content management systems provide tools for finding all referring topics or maps. If you have these tools, use them.

Involve all stakeholders in the decision to modify shared content. You might find that to satisfy all consumers of shared content, you need to create a separate version of the content. You want to avoid the situation where changes to shared content breaks—or introduces falsehoods into—documents.

Practice

1. Open the file `samples/conrefs/t_watching_wild_ducks_start.dita`.
2. After the first paragraph in the prerequisites, create a conref to the only paragraph in the topic file `samples/conrefs/c_duck_safety.dita`.
3. Check your file `samples/conrefs/t_watching_wild_ducks_start.dita` against the file `samples/conrefs/t_watching_wild_ducks.dita`.

Course VII. Advanced Reuse in DITA

Lesson 1: Using conditions

Objectives

- Identify the four fundamental conditional attributes.
- Apply the four fundamental conditional attributes.
- Apply the rev attribute.
- Create a ditaval file to filter and flag content.

This lesson shows how to use conditional filtering to create reusable topics.

Additional reading

[DITA Specification, Conditional processing \(profiling\)](#)

[DITA Style Guide, Conditional processing concepts](#)

[DITA Style Guide, Condition \(or select\) attributes](#)

[DITA Style Guide, Filtering and flagging](#)

[DITA Style Guide, DITaval elements](#)

[Conditional content in DITA](#)

[Optimizing Content Reuse with DITA](#)

Housekeeping and sample files

To view the sources for some of the examples in this course, download [reuse_advanced_samples.zip](#) now. Extract the contents and put them in a directory that you can access easily. If you have access to an output generator (usually the DITA Open Toolkit), you can use it to generate output and see these principles in action.

The samples folder contains these files, which help illustrate principles of conditional filtering and flagging and the conref push operation:

- advanced_reuse_examples.ditamap
- c_conrefpush_sources.dita
- c_conrefpush_target.dita
- c_filtering_and_flagging.dita

- domestic.ditaval
- show_rev.ditaval
- wild.ditaval

The lessons will instruct you on which files to use for the samples.

Create a local copy of each file to work in as you complete the lessons. That way, if you reach a point where your working file doesn't match the examples, or is broken for any reason, you can make a fresh copy and resume your work or start over.

In the instructions and examples, we show you the DITA code for the sample files. Most DITA editors have auto-complete or other similar features to guide you through the process of adding elements (for example, if you type the opening tag of an element, most DITA editors will automatically add the closing tag for you). Therefore, you will probably not need to create every piece of code from scratch as you work.

Review of basic reuse

In the [Introduction to reuse in DITA](#) course you learned that:

- You should keep reusability in mind when creating topics. Reusable topics are consistent, context-free, and generalized.
- You can reuse topics in many different maps.
- You can reuse maps in many different maps.
- You can reuse same topic multiple times in the same map.
- You can use content references (or “conrefs”) to reuse an element across many different topics.

This course builds on these principles and ideas by introducing you to conditions (filtering), keys, and advanced content references.

Marking conditional elements

DITA conditional filtering allows you to choose what to show and hide in your DITA topics when they are processed for output.

You mark DITA elements for filtering by adding special conditional attributes to elements; the value you use with a conditional attribute is used to determine whether the element should be hidden (filtered out) or shown.

The four basic conditional attributes and their suggested uses are:

- audience

Content is intended for a specific group of readers. The audience can be seen a number of different ways; it might be the experience level, role, security clearance, or some other way of grouping potential readers.

- platform

Content is specific to a particular hardware or software platform.

- product

Content is specific to a particular product.

- otherprops

Content is specific to a custom aspect.

For example, you might use the platform attribute to indicate whether content is intended for iPhone or Android users:

```
<p platform="android">Visit Google Play for more information.</p>
<p platform="iphone">Visit the App Store for more information.</p>
```

When generating output for Android users, you can tell the DITA filtering mechanism to hide elements where platform="iphone" and show elements where platform="android". To generate output for iPhone users, you show elements where platform has the value "iphone" and hide elements where platform has the value "android". (How to hide and show content is described later in this lesson.)

The names of the attributes are defined in the DITA specification. They are a suggestion about how you might use them, but you should not feel locked into using them specifically for audience, platform, or product filtering; it's up to you to determine how you want to use each one.

As with the attributes themselves, the values you use with each of the conditional attributes are up to you to decide. You can use any characters you want for the conditional attribute values, except for the space character. Case is important in the conditional attribute values, so choose one form of capitalization and use it consistently.

You can specify multiple values in these attributes; separate each value with one or more spaces.

The single most important thing, however, is that all content creators in your group, team, company, or organization agree on the purpose of each of the attributes and the values to use with the attributes.

Note: The four attributes listed here are just a starting point. It is possible to use DITA specialization to create additional conditional attributes that have meaning for your content. The DITA specification identifies another conditional attribute: props. This attribute is used as the basis for specialization.

Keep in mind that filtering of elements extends to DITA map elements, such as <topicref> and <mapref>. This way, you can use conditional attributes and filtering to remove entire topics or chapters from your output.

What can you filter?

The DITA filtering mechanism is quite powerful and flexible. You can use it to filter out almost any type of element in DITA topics and in DITA maps.

In DITA topics, some of the elements you can filter are:

- sections
- paragraphs
- ordered and bulleted lists
- list items
- notes
- tables
- table rows
- figures
- images

In DITA maps and bookmaps, some of the elements you can filter are:

- topic references
- map references
- key definitions
- chapters and appendixes
- frontmatter, backmatter, and their elements

Where to be careful

There are two areas where you need to exercise some caution in filtering: inline elements and elements that have required children.

It is not a good idea to filter inline elements. This is mostly a localization concern. The previous course ([Introduction to reuse in DITA](#)) covered this to some degree: what makes sense in one language when it is filtered out might not make sense (or translate well) when it is presented in another language.

You need to be careful when filtering elements that have required children, otherwise this might result in an invalid structure. If filtering removes required elements, it will result in an invalid topic. For example, if you add conditional attributes to all `` elements in an unordered list (``) and filtering removes them all, the empty `` element will be invalid. Other elements that have required children include ``, `<sl>`, `<steps>`, `<substeps>`, `<properties>`, `<table>`, `<simplatable>`. If you are not sure which child elements are required, check the [DITA 1.2 specification](#).

What you cannot filter

There are a number of elements that cannot be filtered out. These include:

- topic titles
- individual table <entry> elements
- step command (<cmd>) elements
- choicetable entries

The ditaval file

When generating output from sources that use the conditional attributes, you need to tell the output processor what to hide and what to show.

In most output generators, you use a ditaval file, which indicates which attributes with what values to filter out of your content. (Some output generators do not use ditaval files; they use filtering mechanisms of their own design, but the principles are still the same.)

The ditaval file is an XML file, but it is not a DITA file; that is, it does not use the DITA doctypes or elements. In DITA 1.2, the ditaval file is not included in DITA maps; the file path to your ditaval file is passed to output generators using a command argument. In DITA 1.3, you can use the <ditavalref> element to include a ditaval file in a DITA map.

The name of your ditaval file is up to you, although it's a good idea to use a filename that indicates the file's purpose. Typically you store the ditaval file along with your maps, although it can reside anywhere that is accessible to your output generator.

Typically you will have one ditaval file for each main map or bookmap file.

The root element in the ditaval file is the <val> element; in a typical file, the <val> element contains one or more <props> elements:

```
<val>
  <prop att="platform" val="iphone" action="exclude" />
</val>
```

This example shows the basic form of a <prop> element.

- The att attribute indicates one of the conditional attributes. In this case, platform.
- The val attribute indicates what value to match in the att attribute. In the example, all elements where platform="iphone" are filtered.

Note: When specifying the val attribute, case is important. A val attribute containing "iPhone" would not match a conditional attribute with the value "iphone".

- The action attribute indicates what to do when a match for the attribute and value is found. In this case, any element that contains platform="iphone" is excluded from output (that is, it is hidden).

Note: The DITA sources shown in the following examples are in the downloadable samples file [reuse_advanced_samples.zip](#). The DITA file is c_filtering_and_flagging.dita; the ditaval file is domestic.ditaval.

Conditional filtering example

In this example DITA topic, the second <p> element contains a sentence within a <ph> (phrase) element. The product conditional attribute has the value "wild_ducks":

```
<p>For the healthiest ducks, we recommend using our feeds.</p>
<p>All our duck feeds are composed of cracked corn, oats, rice, and milo seed.
<ph product="wild_ducks">Our wild duck feed also includes worm meal and fish meal.
</ph>
</p>
<p>All feeds are available in 5, 10, and 20 pound sacks.</p>
```

If you generate output from this content without using a ditaval file, it looks like this:

Conditional filtering and flagging

For the healthiest ducks, we recommend using our feeds.

All our duck feeds are composed of cracked corn, oats, rice, and milo seed. Our wild duck feed also includes worm meal and fish meal.

All feeds are available in 5, 10, and 20 pound sacks.

But if you use this ditaval file to filter out content where product="wild_ducks":

```
<val>
  <prop att="product" val="wild_ducks" action="exclude" />
</val>
```

The output looks like this:

Conditional filtering and flagging

For the healthiest ducks, we recommend using our feeds.

All our duck feeds are composed of cracked corn, oats, rice, and milo seed.

All feeds are available in 5, 10, and 20 pound sacks.

For more information about ditaval files and how to use them for filtering, see <http://docs.oasis-open.org/dita/v1.2/os/spec/common/about-ditaval.html#ditaval>

Flagging content

In addition to filtering, you can use conditional attributes to flag (or highlight) content that is intended for different readers or different products. You can use any conditional attribute for flagging, and you can use the `rev` attribute to indicate and flag a specific revision of your product or service.

Note: The DITA sources shown in the following examples are in the downloadable samples file [reuse_advanced_samples.zip](#). The DITA file is `c_filtering_and_flagging.dita`; the ditaval files are `wild.ditaval` and `show_rev.ditaval`.

Flagging the conditional attributes

To use the conditional attributes for flagging, use the ditaval `<prop>` element with the `action="flag"` attribute. This allows you to flag text with color, background color, text styles, and images (depending on what is supported in your transforms and the output generator). For example:

```
<prop att="platform" val="android" action="flag" color="purple"/>
```

The `<prop>` element shown here directs the output generator to use purple text on any element in which the `platform` attribute contains the value “android”.

When flagging, you can use these `<prop>` attributes to flag text:

- `color`: style the text using the specified color
- `background`: place the specified color behind the text
- `style`: format the text using a specific style (italics, bold, underline, double-underline, or overline).

The `color` and `background` attributes can use either a 6-digit hex color code (such as “#0000FF”) or a color name (aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, or yellow).

For example, if you use this example (from the previous topic):

```
<p>For the healthiest ducks, we recommend using our feeds.</p>
<p>All our duck feeds are composed of cracked corn, oats, rice, and milo seed.
<ph product="wild_ducks">Our wild duck feed also includes worm meal and fish meal.
</ph>
</p>
<p>All feeds are available in 5, 10, and 20 pound sacks.</p>
```

With this ditaval file:

```
<val>
  <prop action="flag" att="product" val="wild_ducks" color="red"
style="underline"/>
</val>
```

The output looks like this:

Conditional filtering and flagging

For the healthiest ducks, we recommend using our feeds.

All our duck feeds are composed of cracked corn, oats, rice, and milo seed. Our wild duck feed also includes worm meal and fish meal.

All feeds are available in 5, 10, and 20 pound sacks.

You can also specify images to place before and after the flagged element. Use the <startflag> and <endflag> elements as children of a <prop action="flag"> element. The imageref attribute indicates the path to the image to use.

```
<prop att="platform" val="android" action="flag">
  <startflag imageref="images/android_icon.png">
    <alt-text>Android icon</alt-text>
  </startflag>
</prop>
```

The <startflag> and <endflag> elements can optionally contain an <alt-text> element that contains alternate text for the image.

The rev attribute

You use the rev attribute to indicate and flag a specific revision of your product or service. The rev attribute is available on almost all DITA elements. You use it just as you would a conditional attribute:

```
<p rev="v2.1">If you have additional needs, consider using the custom table
feature. </p>
```

The values you use with the rev attribute are up to you and your organization. The important thing is to be consistent in how you use the values.

As with the conditional attributes you can specify multiple revision values in the rev attribute; separate the values with one or more spaces.

```
<p rev="v2.7 v2.8">The field is limited to 32 characters.</p>
```

Note: It is important to note that you cannot use the rev attribute for filtering content. Its only purpose is for flagging.

Showing rev flagging in output

You control the display of the rev attribute with the <revprop> element in the ditaval file. You can apply styling just as with the <prop> element (using the color, background color, and style attributes). You can also use the <startflag> and <endflag> elements to add images before and after the element marked with the rev attribute.

For example, if a DITA topic contains this content:

```
<p>The Duck Database tables will handle most of your duck needs.</p>
<p rev="2.1">If you have additional needs, consider using the custom table
feature.</p>
<p>For more information on the standard tables, see Appendix C.</p>
```

And you generate output using a ditaval file containing:

```
<val>
  <revprop action="flag" val="2.1" bgcolor="aqua"/>
</val>
```

The output might look like this:

The Duck Database tables will handle most of your duck needs.

If you have additional needs, consider using the custom table feature.

For more information on the standard tables, see Appendix C.

Depending on the format of your output and your output generator, you can also use the `<revprop>` element to place change bars in the margins next to text marked with a `rev` attribute. However, the capabilities and the syntax of the `changebar` attribute vary depending on the output format and the output generator. If you need to use this feature, it is best to review the documentation for your output generator.

For more information about ditaval files and how to use them for flagging, see <http://docs.oasis-open.org/dita/v1.2/os/spec/common/about-ditaval.html#ditaval>

Best practices for conditions

Do not filter sentence fragments

As described in the basic course on reusing elements, do not use filtering on anything smaller than a sentence.

It's better to have two instances of an entire sentence that are grammatically correct (and will remain correct when translated) than to risk ending up with poorly constructed—or even non-sensical—sentences.

Communicate your choices

Create a style guide or some other form of documentation so that everyone on your team, organization, or company uses conditional attributes and values consistently.

Make sure you have agreement on:

- How and why everyone uses the audience, platform, product, and otherprops attributes.

- The precise values everyone uses with those attributes and the meaning of each value. Remember that case matters when comparing attribute values with the val attribute in ditaval file.

Consider specialization

If there are not enough conditional attributes for your needs, or if the names of the attributes do not fit with how you are using them, consider using the DITA specialization mechanisms to create additional conditional attributes.

Lesson 2: Using keys

Objectives

- Define a key for a path resource
- Use a key for a path resource
- Define a key for a text string resource
- Use a key for a text string resource
- Assemble key definitions in a submap

A DITA key is a placeholder for short text strings and file paths (or URLs). When you create a reusable topic, you can use keys in places where the text strings or file paths are different each time the topic is used. You define the final value for a key in a DITA map. When the reusable topic is used in different maps, the key values can be defined differently in each map.

This lesson describes how to define and use keys for short text strings and file paths.

What is a key?

A DITA key allows you to create a placeholder for a file path or a short piece of text. You create the key name and definition (the text to use in place of the key) in a DITA map;. You refer to a key by specifying the key name in a keyref attribute. The keyref attributes are resolved when topics and maps are transformed into an output format.

The advantage of using a key is that you define the key in one place (in the map), then refer to that key by name throughout all your topics. If the file path or text changes, you only need to change the definition and the new definition is reflected everywhere that the keyref attribute is used.

Keys make it easy to create reusable topics. Because key definitions can be different for each map, a reusable topic that uses keys can contain different file paths or strings, depending on which map it is used in.

For instance, you might create a reusable topic that uses two keys:

- The key `product_name` contains the name of a product
- The key `product_image` contains the path to a picture of the product

In the map for Product A, `product_name` could be defined as “Product A” and `product_image` as “images/ProductA.png”.

In the map for Product B, `product_name` could be defined as “Product B” and `product_image` as “images/ProductB.png”.

When your topic is used in the map for Product A, the product name “Product A” is used and the product image shows the correct product. When your topic is used in the map for Product B, the correct product name and image are shown for that product.

You define keys in a DITA map with the `<keydef>` element, which is a specialization of the `<topicref>` element.

There are some differences between keys used for paths and keys used for text strings. There are differences in how you define the keys and how you use them. These two uses are described in the next two topics.

Note: This description of keys does not address scoped keys, which were introduced in DITA 1.3. The concept of scoped keys will be addressed in a future LearningDITA course.

Note: There is a third use for keys: they can be used to specify a path to conref content. These “conkeyrefs” are addressed in the next lesson (“Advanced conrefs”).

Using keys for paths

Defining the key

To define a key that will contain a path (file path or URL), add the `<keydef>` element to a map. Use these attributes with the `<keydef>` element:

- `keys`: the name of the key
- `href`: the key target
- `format`: the type of file indicated by the key

For example:

```
<keydef keys="product_image" href="images/product_B.png" format="png"/>
```

Although the key name can contain some special characters, it is a good practice to limit names to letters, numbers, and the underscore character. If you need to use other special characters, check the [DITA 1.2 specification](#).

It is also a good practice to use the `format` attribute when defining a key. Possible values include “dita”, “ditamap”, “pdf”, “html”, and graphic formats, such as “png” or “svg”.

If you're defining a key for an external resource, such as a URL, you must use `scope="external"` with the `<keydef>` element:

```
<keydef keys="our_url" href="http://www.scriptorium.com" scope="external"
format="html"/>
```

The `scope` attribute prevents the DITA Open Toolkit (or other processors) from attempting to resolve the `href` target as part of the definition. Because the `href` attribute points to a web page, the `format` attribute must contain the value `html`.

Using the key

To use a key for a file path or URL, use the `keyref` attribute instead of the `href` attribute:

```
<image keyref="product_image"/>
```

You can use keys (with the `keyref` attribute) in any element that uses an `href` attribute to indicate a file path. These elements include (but are not limited to):

- `<image>`
- `<xref>`
- `<link>`
- `<coderef>`
- `<topicref>`
- `<mapref>`

Using keys for text

You can use a key as a variable for replaceable text in your DITA topics. Typical uses of keys for text variables include:

- Product names
- Product numbers
- Company names
- Telephone numbers
- Titles of other books in a document set

Although there is no limit on the length of the string defined by a key, strings in keys are usually fairly short.

Note: The previous course ([Introduction to reuse in DITA](#)) pointed out that it was not a good idea to use replaceable text in anything smaller than a sentence. Overall, this is good advice. However, if you are

using keys for a company or product name, particularly when documenting a product that might be rebranded or OEMed, the convenience and consistency provided by keys may outweigh some considerations of localization.

To define a key that will contain a text string, use the `keys` attribute to name the key, then nest a `<topicmeta>` element containing a keyword definition inside the `<keydef>` element. For example:

```
<keydef keys="product_name">
  <topicmeta>
    <keywords>
      <keyword>Duck Database</keyword>
    </keywords>
  </topicmeta>
</keydef>
```

This syntax is quite verbose. The good thing is: you define it once, then forget about it.

Note that the string is limited to the content that is allowed in the `<keyword>` element, that is: plain text, the `<text>` element, and the `<tm>` (trademark) element. You cannot use other elements in the string to provide inline formatting or cross-references. If you need additional elements within the string, consider using a `conref` instead.

Using the key

To use a key for a text string, use the `keyref` attribute with these elements (or any element specialized from these elements):

- `<ph>`
- `<term>`
- `<keyword>`

This example show how a key can be used with a `<ph>` (phrase) element:

Congratulations on purchasing `<ph keyref="product_name"/>`!

When the DITA map includes the `<keydef>` element shown above, the output would contain:

Congratulations on purchasing Duck Database!

Advanced key uses

Compound uses of the `<keydef>` element

You can use keydefs to simultaneously define both file paths and text strings. Consider the case where you need to cross-reference a topic in many different places, but you want to use some text other than the topic title in the cross-reference. In this case, you can create a `<keydef>` element that associates a key with an href to the topic, but also uses the nested `<topicmeta>` to define a `<linktext>` element:

```
<keydef keys="initing_ddb" href="init_ddb.dita">
  <topicmeta>
    <linktext>Initializing the Duck Database</linktext>
  </topicmeta>
</keydef>
```

If you reference this key in a cross-reference (<xref> element), the link will target `init_ddb.dita` and the text for the link will be “Initializing the Duck Database”:

```
<xref keyref="initing_ddb"/>
```

Key references with fall-backs

If you use both a `keyref` and an `href` attribute in an element, a DITA output generator will attempt to resolve the `keyref` first. If the key is not defined, the `href` target is used as a fall-back.

Best practices for keys

Use the <keydef> element, rather than <topicref>

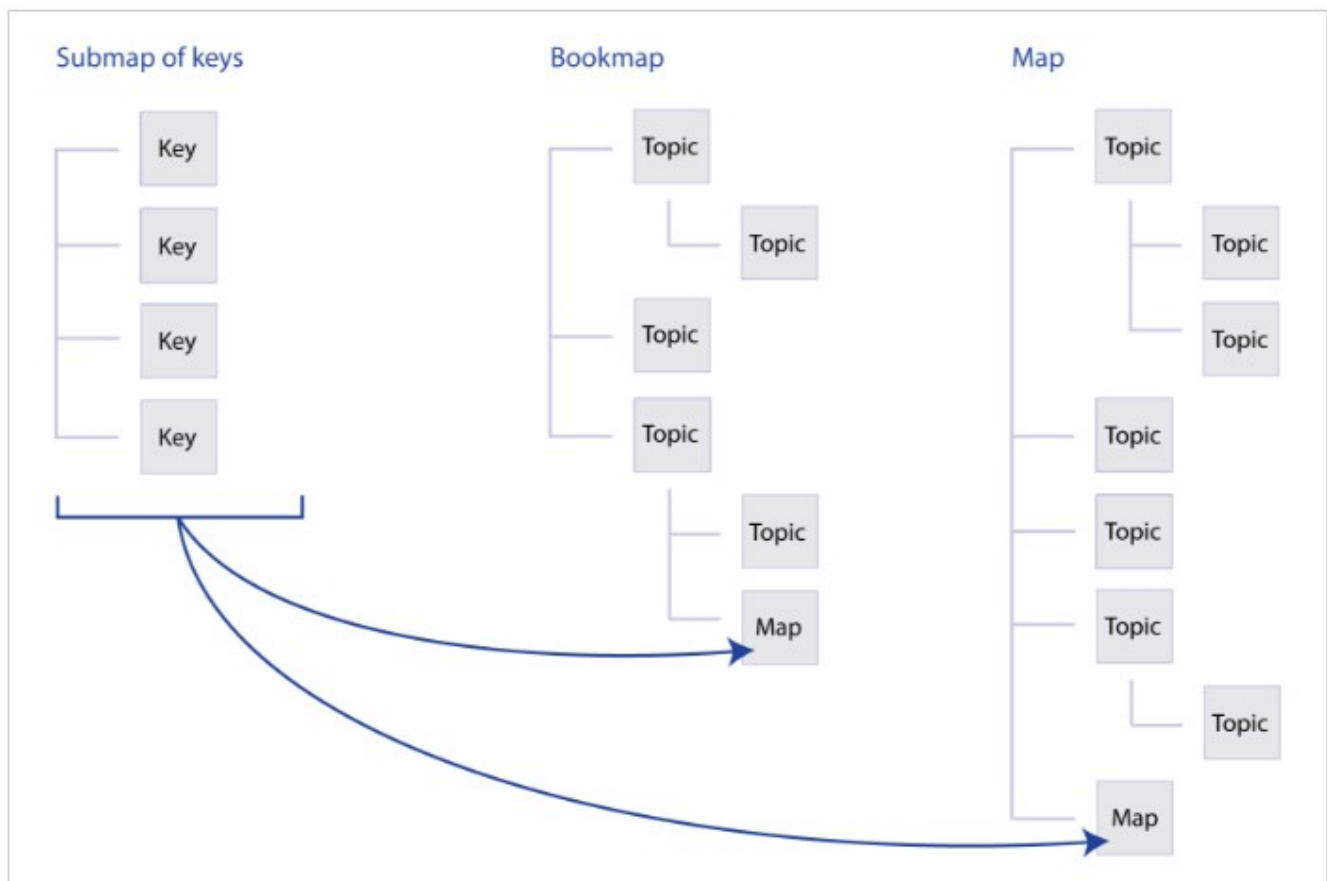
The <keydef> element is a specialization of the <topicref> element. Although you can use the <topicref> element to define keys, it’s better to use <keydef> because:

- It indicates specifically what you’re doing (semantics).
- It automatically specifies `processing-role="resource-only"`. Without this attribute, the output generator uses the referenced topic exactly where it occurs in the map, which is probably not what you want.
- It requires you to use the `keys` attribute (which helps if you tend to forget things like that).

Gather keys in submaps

It’s a good idea to gather all related keys together in a submap. There are two main reasons for doing this:

- Once you have gathered all related key definitions together in a single submap, you (and others on your team) can reuse that submap in maps or bookmaps for other output targets
- To change the key definitions used by the map or bookmap, it’s just a matter of switching out one key submap for another (rather than having to update all the `keydef` elements). Additionally, you can add multiple <mapref> elements to the map or bookmap and use DITA conditional filtering (as described in the first lesson of this course) to hide all but one of the <mapref> elements.



Adding key definitions to bookmaps

The structure of the `<bookmap>` element does not allow `<topicref>`, `<keydef>`, or `<mapref>` elements as children of the `<bookmap>` element, so there is no obvious place in a `<bookmap>` element to place your key definitions.

A number of DITA users place `<keydef>` or `<mapref>` elements as children of the `<frontmatter>` element. This ensures that they are immediately visible to people editing the bookmap.

Lesson 3: Advanced conrefs

Review of content reference basics

In the [Introduction to reuse in DITA](#) course you learned how to use the `conref` attribute to reuse elements. The `conref` attribute allows you to pull content from one element (possibly from a different topic) into another element.

To use a content reference:

- The referenced element (the information being pulled) must have an `id` attribute.

```
<p id="use_me">This is a reusable paragraph.</p>
```

- The referencing element (which is nominally empty) uses the conref attribute to specify the id of the containing topic and the id of the referenced element. For example:

```
<p conref="#my_topic/use_me"/>
```

or, if the referenced element is in another file:

```
<p conref="topic_file.dita#my_topic/use_me"/>
```

The content reference is resolved when the topic containing the referencing element is processed for output. Some DITA editors can also display the referenced content (as read-only text).

Conkeyrefs

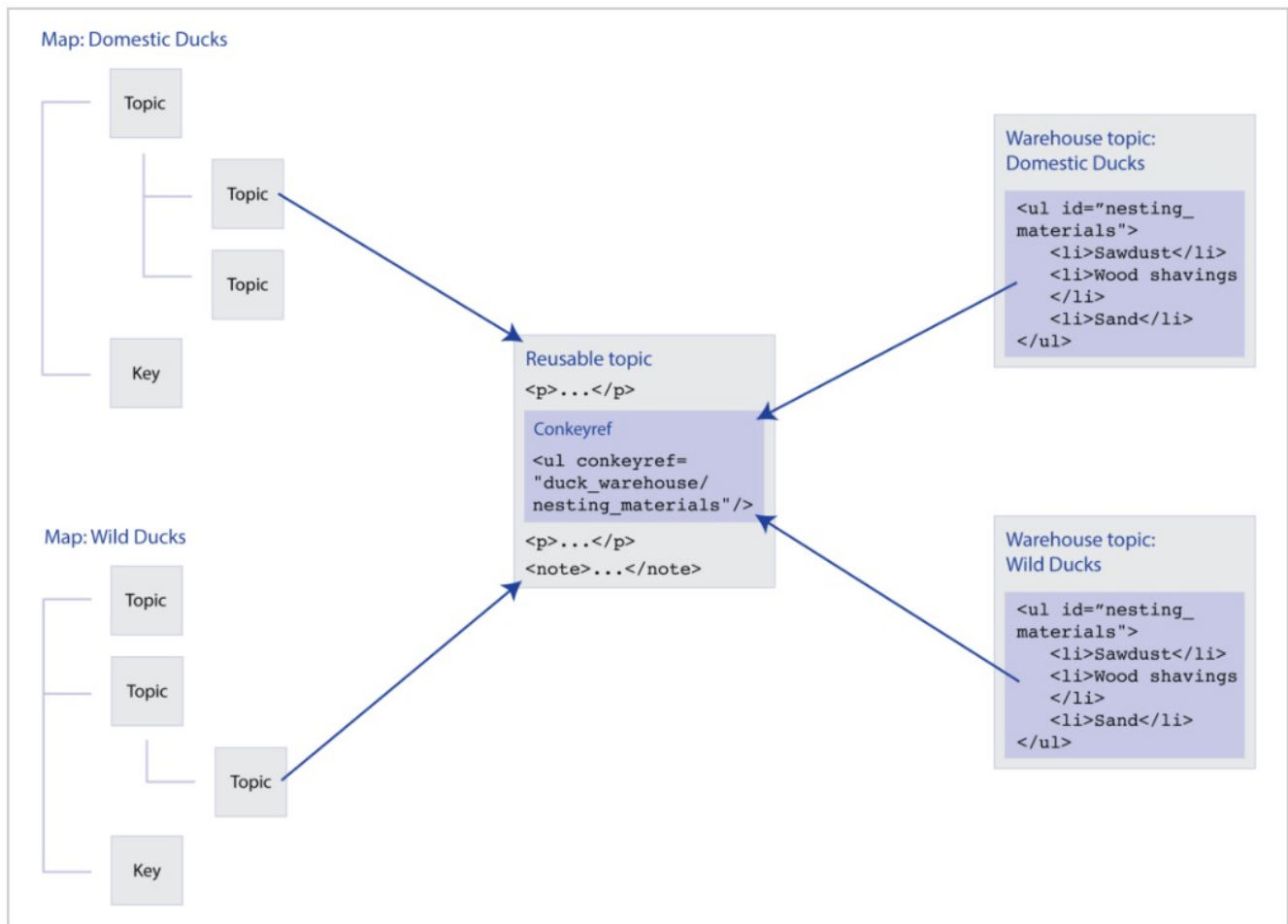
A conkeyref is a content reference that uses a key instead of a file path. As described in the [Introduction to reuse in DITA](#) course, a content reference to an element in another file might look like this:

```
<ul conref="domestic_duck_warehouse.dita#domestic_warehouse/nesting_materials">
  <li/>
</ul>
```

This conref pulls a `` element from a warehouse topic that contains reusable elements relevant to domestic ducks.

If the topic containing this conref must be reusable, a hard-coded filename will be a problem. To use the reusable topic in a map that discusses wild ducks, the conref needs to point to a different warehouse topic file (`wild_duck_warehouse.dita`).

Rather than hard code the path to a file, you can use the conkeyref attribute to create a content reference that uses a key, rather than a filename.



The first step in using a conkeyref is to create a DITA topic (domestic_duck_warehouse.dita) that contains a reusable element (including an id attribute):

```
<ul id="nesting_materials">
  <li>Sawdust</li>
  <li>Wood shavings</li>
  <li>Sand</li>
  ...
</ul>
```

In a map that references the topic that will use the conkeyref, define a key for the warehouse file:

```
<keydef keys="duck_warehouse" href="domestic_duck_warehouse.dita"/>
```

The example at the beginning of this topic shows a element with a conref. Replace the conref attribute with a conkeyref attribute that contains the key, a slash ('/'), and the id of the element to be pulled:

```
<ul conkeyref="duck_warehouse/nesting_materials">
  <li/>
```


Note: When using conkeyrefs you do not need to use the id of the topic that contains the referenced element.

When the topic containing the conkeyref is processed, the key duck_warehouse is replace with the current key definition, which is domestic_duck_warehouse.dita.

You can reuse the topic containing the conkeyref in another DITA map, but you might need it to pull content from a file that is specific to the new map. To do this, add a <keydef> element to the new map that defines the key so that it points to a different file. In this example, the new map (that addresses wild ducks) defines the duck_warehouse key to point to wild_duck_warehouse.dita:

```
<keydef keys="duck_warehouse" href="wild_duck_warehouse.dita"/>
```

The file wild_duck_warehouse.dita defines a different element; however, it must use the same id attribute:

```
<ul id="nesting_materials">
  <li>Ferns</li>
  <li>Twigs</li>
  <li>Grass</li>
  ...
</ul>
```

When this new key is used in the wile duck map, the unordered list of wild duck nesting materials is used.

Finally, note that all the same rules that apply to conrefs also apply to conkeyrefs:

- The referencing element must be the same type (element name) as the referenced element.
- The referencing element must be valid (for instance, a element must contain at least one element).

Conrefend

The conref attribute is useful for pulling content from a single element into another topic. But what if you want to pull in content from a series of elements?

To pull a sequence of two or more elements into a topic, use the conrefend attribute in combination with the conref attribute. The two attributes specify the starting and ending elements in a sequence of elements. The starting element, the ending element, and all elements in between are pulled into the referencing topic.

For example, you might have a warehouse topic (named wh.dita) that contains this element:

```
<concept id="wh_elements">
  ...
```



```

<ul>
  <li id="food_insects">small insects</li>
  <li id="food_fish">small fish and fish eggs</li>
  <li id="food_crustaceans">crustaceans</li>
  <li id="food_snails">snails and mollusks</li>
  <li id="food_frogs">frogs and salamanders</li>
  <li id="food_fruit">berries and nuts</li>
</ul>
</concept>

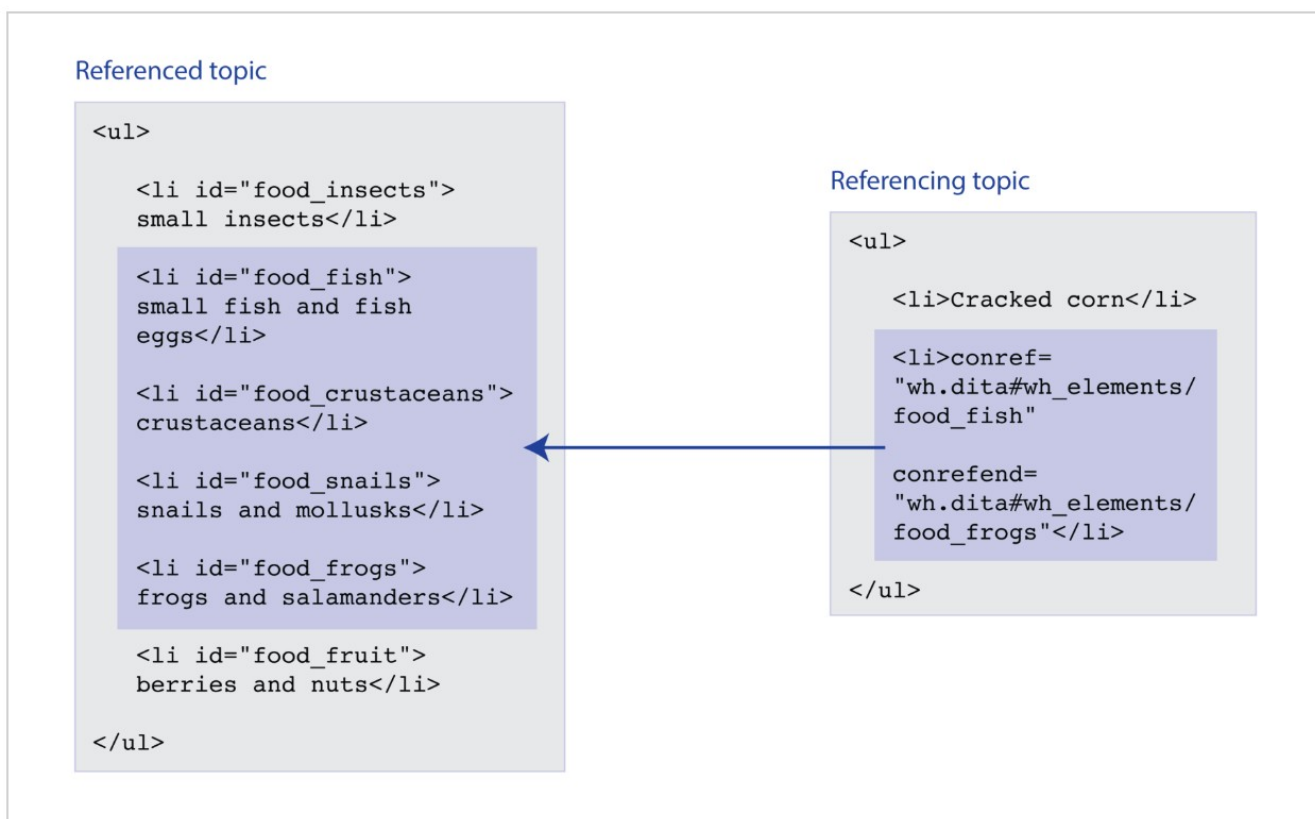
```

To pull the four list items from “fish” through “frogs” into a list in another topic, use this conref and conrefend combination:

```

<ul>
  <li>cracked corn</li>
  <li conref="wh.dita#wh_elements/food_fish"
conrefend="wh.dita#wh_elements/food_frogs"/>
  ...
</ul>

```



If you use conrefend, keep in mind that:

- Both of the elements identified by the conref and conrefend attributes must be the same element name as the referencing element. For instance, if the referencing element is a <p> element, the elements indicated by the conref and conrefend attributes must also be <p> elements.

- The elements that occur between the elements indicated by the `conref` and `conrefend` attribute do not have to be the same type. For example, if the `conref` and `conrefend` attributes indicate `<p>` (paragraph) elements, and if there are non-`<p>` elements (such as `<note>` or `<image>`) between the `conref` and `conrefend` `<p>` elements, those elements will also be included in the content reference.

You can also use `conrefend` with `conkeyrefs`. If `conrefend` is combined with a `conkeyref`, the filename in the `conrefend` attribute is ignored and is replaced with the filename used by the key specified in the `conkeyref`.

Note: Not all CCMSs support the `conrefend` attribute. Test or check with your vendor before using this attribute.

Conref push (conaction)

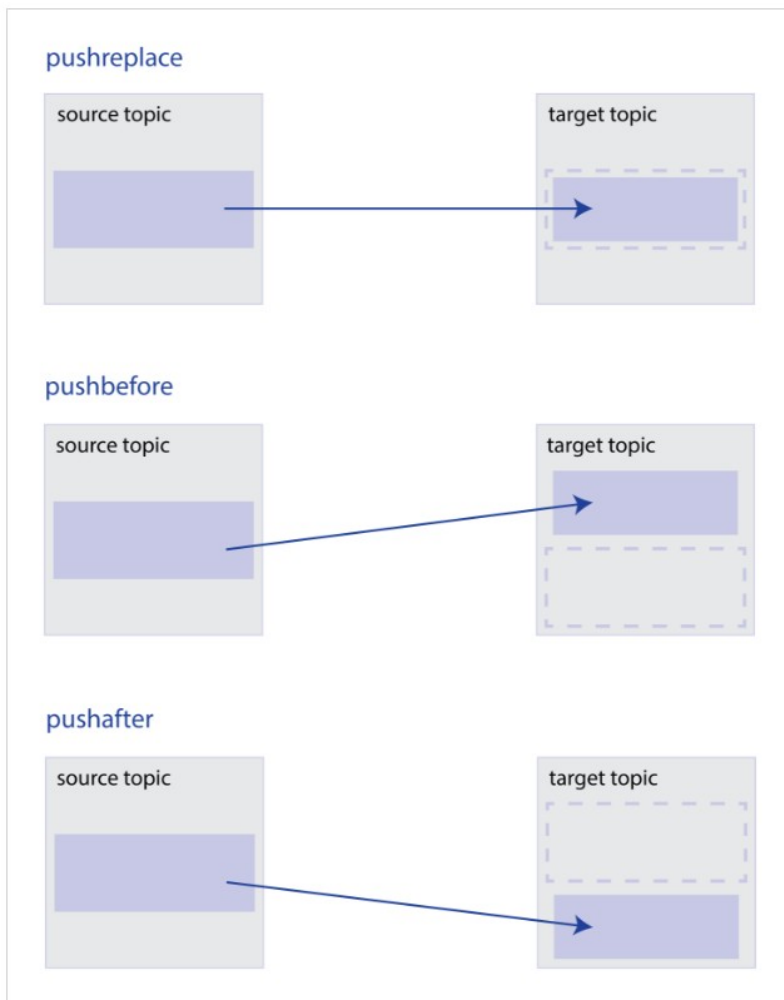
When a topic uses a content reference, content is “pulled” from the referenced topic into the referencing topic. However, there are times when you need to insert an element into an arbitrary location of a reusable topic.

For instance, you might have a reusable topic that does its job well most of the time when it is used. However, in one case (for one product, or for one customer), you need to add an extra paragraph to that reusable topic. One solution might be to add the paragraph and use conditional filtering to only display the paragraph for that one use. But what if you don’t own (or have the rights to modify) the reused topic? What if the reused topic is maintained in a publicly visible area (say on GitHub), and you don’t want your addition to be visible to the rest of the world?

The `conref push` (or `conaction`) mechanism allows you to “push” content from one topic in a map (the source topic) into another topic in the same map (the target topic).

The `conref push` mechanism allows you to push in three ways. You can push an element so that it:

- Is inserted before a specific element.
- Replaces a specific element.
- Is inserted after a specific element.



Just as with a normal conref operation, the source and target elements used in the conref push must be the same type.

To use conref push you need to do three things:

- Make sure that the target element (the element you want to push your element before, after, or in place of) has an id attribute.

For example, to push an element before, after, or in place of this element, it must have an id:

```
<p id="install_intro">This chapter describes how to install and configure the Duck Database on Windows and Macintosh.</p>
```

- Create a DITA topic containing the element to be pushed.

The topic should contain all necessary elements to make sure that the element being pushed is valid. So, if you need to conref push a `` element, your topic should contain a `<body>` (depending on the topic type), a `` element, and the `` you need to push.

The actual content and attributes of the element being pushed are described in the sections below.

- Add a <topicref> element to your map that points to the topic containing the element to be pushed.

```
<topicref href="c_conrefpush_sources.dita" processing-role="resource-only"/>
```

Note that this <topicref> element must use processing-role="resource-only" because the content should not appear in normal map order in the output.

The following sections show how to use conref push to replace an element, insert an element before a target, and insert an element after a target. All the examples use this topic as the target:

```
<concept id="c_install">
  <title>Installing Duck Database</title>
  <conbody>
    <p id="install_intro">To install the Duck Database on Windows and Macintosh,
follow these instructions.</p>
    <p>If at any time you need help in the installation process, please call our 24-
hour hot line.</p>
  </conbody>
</concept>
```

Output from this topic, without any conref push looks like this:

Installing Duck Database

To install the Duck Database on Windows and Macintosh, follow these instructions.

If at any time you need help in the installation process, please call our 24-hour hot line.

Note: The DITA sources shown in these examples are in the downloadable samples file [reuse_advanced_samples.zip](#). The target DITA file is c_conrefpush_target.dita; the source DITA file is c_conrefpush_target.dita.

Replacing the target element

To replace an element in the target topic, use the element's conref attribute to identify the element to replace, and set the conaction attribute to "pushreplace":

```
<concept id="c_conrefp_source">
  <title>Conref push sources</title>
  <conbody>
    <p conref="c_conrefpush_target.dita#c_install/install_intro"
conaction="pushreplace">
      To install the Duck Data base on Windows, follow these instructions. </p>
  </conbody>
</concept>
```

The output from the target topic now looks like this:

Installing Duck Database

To install the Duck Data base on Windows, follow these instructions.

If at any time you need help in the installation process, please call our 24-hour hot line.

Inserting an element before the target element

To insert the element before the target element, you use two elements (both must be the same as the target element):

1. The first element uses the conaction attribute set to the value “pushbefore” and contains the content to be pushed.
2. The second element doesn’t contain any content but uses both the conref attribute (to identify the target element) and the conaction attribute set to the value “mark”.

This example shows the two <p> elements:

```
<concept id="c_conrefp_source">
  <title>Conref push sources</title>
  <conbody>
    <p conaction="pushbefore">
      Make sure you have performed the pre-configuration steps.
    </p>
    <p conref="c_conrefpush_target.dita#c_install/install_intro" conaction="mark"/>
  </conbody>
</concept>
```

The output from the target topic now looks like this:

Installing Duck Database

Make sure you have performed the pre-configuration steps.

To install the Duck Database on Windows and Macintosh, follow these instructions.

If at any time you need help in the installation process, please call our 24-hour hot line.

Inserting an element after the target element

To insert the element after the target element, use the same two elements, but the element containing conref and conaction=”mark” must come before the element containing conaction=”pushafter”:

```

<concept id="c_conrefp_source">
  <title>Conref push sources</title>
  <conbody>
    <p conref="c_conrefpush_target.dita#c_install/install_intro" conaction="mark"/>
    <p conaction="pushafter">To install on Ubuntu, see the subsequent section. </p>
  </conbody>
</concept>

```

The output from the target topic now looks like this:

Installing Duck Database

To install the Duck Database on Windows and Macintosh, follow these instructions.

To install on Ubuntu, see the subsequent section.

If at any time you need help in the installation process, please call our 24-hour hot line.

Two final notes on conref push:

- You cannot use the conrefend attribute with the conref push mechanism.
- You can use the conkeyref attribute to indicate the target of a conref push.

Best practices for advanced conrefs

Should short strings use text keys or conkeyrefs?

If you want to use keys to reference a short string, such as your company name, a text key is the easiest way to do it.

However, the <keyword> element used to define text keys can only contain text, the <text> element, or the <tm> (trademark) element. If the string requires internal markup, it's better to use a warehouse file to define a <ph> (phrase) element containing the string and its markup.

For example, for the company name “Pure-H₂O”, you could not store this in a <keyword> element; you would need to use a <ph> element:

```
<ph id="company_name">Pure-H<sub>2</sub>O</ph>
```

To create a reusable topic that referenced a company name, use a <ph> element with a conkeyref to pull in the name.

To use conkeyref or conref push?

A good rule of thumb for deciding between using conkeyref and conref push:

- If the element will always be present, but will change depending on the intended target, use conkeyref.

- If the element will only be present for one or two intended targets (out of many), use conref push.

Gather keys in submaps

As recommended in the lesson on using keys, it is a good idea to use submaps to gather the key definitions for conkeyrefs.

Course VIII. Publishing Output from DITA Sources

Lesson 1: Choosing a DITA publishing environment

Objectives

- Determine what types of output you need to publish
- Evaluate different DITA publishing tools and environments
- Choose the best DITA publishing scenario for your content

This lesson covers some common DITA publishing environments and shows how you can determine which one is the best fit for your content.

Additional reading

[Perils of DITA publishing series](#)

[Publishing with the DITA Open Toolkit](#)

[Use an External DITA Open Toolkit in Oxygen XML Editor](#)

[Selecting a Component Content Management System for DITA](#)

Housekeeping and sample files

To view the sources for some of the examples in this course, download [publishing_output_samples.zip](#) now. Extract the contents and put them in a directory that you can access easily. If you have access to an output generator (usually the DITA Open Toolkit), you can use it to generate output and see these principles in action.

The samples folder contains two folders: ducks and plugins.

- The ducks folder contains a DITA map, a bookmap, an images folder, and 11 DITA topic files. You use these to test aspects of publishing output from DITA sources
- The plugins folder contains a single zip file (com.learningdita.ld_xhtml.zip), which is a DITA plugin. You use this to explore how to work with custom plugins.

The lessons will instruct you on which files to use for the samples.

Create a local copy of each file to work in as you complete the lessons. That way, if you reach a point where your working file doesn't match the examples, or is broken for any reason, you can make a fresh copy and resume your work or start over.

In the instructions and examples, we show you the DITA code for the sample files. Most DITA editors have auto-complete or other similar features to guide you through the process of adding elements (for example, if you type the opening tag of an element, most DITA editors will automatically add the closing tag for you). Therefore, you will probably not need to create every piece of code from scratch as you work.

Determining output requirements

In other courses, you learned how authoring in DITA gives your content semantic value and helps increase reuse. In this course, you will learn about the benefits of DITA for publishing and gain hands-on experience with generating various types of output from DITA source content.

When you're planning to publish your DITA content, one of the first questions you might ask yourself is, "What sort of publishing environment do I need?" The first step in choosing the right publishing environment for your DITA workflow is determining your output requirements.

You might need both print-based outputs (such as PDF) and electronic outputs (such as HTML, EPUB, or WebHelp) for your content.

Note: For some output types, such as highly-designed printed materials or complex data sheets, you might find that automated publishing does not meet your needs. However, you still want to maintain a single set of source files. In this case, you need to transform your DITA source content into a format that can be opened in a desktop publishing program (such as Adobe InDesign) so that you can manually adjust the formatting before publishing PDF or printed output.

DITA helps make publishing more efficient by:

- Separating content from formatting. In an unstructured environment, authors might maintain one set of source files for print publication and another for electronic publication. The files for print might be authored and formatted in a desktop publishing program, while the files for electronic output might be copied from the print-based source files and re-formatted, or authored separately. With DITA, authors create a single set of source files, and formatting is applied separately to create each required output type. This reduces inconsistencies between printed and electronic versions of the same content and ensures that an update made in the source will be reflected in all outputs.
- Allowing for automated formatting. Because DITA content is separated from formatting, the formatting can be applied automatically during the publishing process rather than manually. Content creators can focus on the content itself rather than how it is displayed on the page, and publishing becomes a faster, push-button process.

DITA publishing environment considerations

DITA publishing environments may include anything from open-source tools used on a standalone basis to commercial, enterprise-level systems that manage the entire content lifecycle. This lesson covers three common DITA publishing environments:

- A standalone instance of the DITA Open Toolkit (DITA OT)
- The DITA OT used in oXygen XML Editor
- The DITA OT used in a component content management system (CCMS)

Note: These DITA publishing environments will be covered in more detail later in this lesson.

The following factors can help you evaluate publishing options:

- How frequently you update your content. The more often you update your content, the more often you will need to generate new output to keep your published material up-to-date. If you have a regular, frequent update schedule, a publishing environment that allows you to schedule automated output builds will work best.
- Localization requirements. Are you translating your content into other languages? If so, are you responsible for publishing the translated content? If you're not translating, will you be in the future? All of these questions are important considerations in choosing a publishing environment. For example, if you're publishing translated content, you may have multiple variations of each output type (such as one PDF format for English, another for Spanish, and yet another for Chinese), which increases your total number of outputs.
- How much content you have. If you plan to publish a high volume of content, you will more likely need an enterprise-level, commercial publishing environment. An open-source publishing tool might work for one or two manuals that contain a few hundred pages of content; it probably won't be feasible for publishing hundreds of manuals with thousands of pages of content.
- How many outputs you need. Do you only need to publish one type of output, or do you need multiple types? For each type of output, how many different formats do you need? (For example, for PDF output, you might need one format for manuals, another for data sheets, and another for training materials, for a total of three separate PDF outputs.) If you need to publish a large number of outputs, you will probably need a commercial system; if you are only publishing one or two total outputs, a standalone environment may suit your needs.
- Future output requirements. Do you have a small number of output requirements now, but know that you will need to add more in the future? You may consider starting with a publishing environment that will accommodate these future requirements. Depending on budget concerns, you may also begin in a smaller or standalone publishing environment and move to a larger one when it becomes necessary. If you choose to start in a smaller environment, it is important to

determine whether the tools you use allow you to export your content and use it in a larger environment later.

If you need to publish PDF output, your publishing environment will need to include an FO formatter. The type of formatter you choose depends on the formatting and design of your PDF output. An open source formatter (such as Apache FOP, the default FO formatter for the DITA OT) may be capable of producing simple, straightforward designs, with limitations on some features. However, a commercial formatter (such as Antenna House Formatter or RenderX XEP) will be better for publishing PDF files with more complex page layouts.

Cost should be a factor in selecting an FO formatter; commercial formatters require licensing fees, while open source formatters do not. You should also consider your localization requirements when choosing a formatter, as different formatters have different degrees of language character support.

Publishing with a standalone DITA OT

The DITA OT is an open-source publishing engine for DITA content. The DITA OT uses plugins to transform the DITA source content into various outputs. You can add and remove plugins as necessary.

Default plugins for the DITA OT include PDF, HTML, WebHelp, and others. These plugins can be updated to change the look and feel of the outputs. DITA OT users can also install their own plugins for more highly customized output.

The DITA OT can be used in conjunction with software for DITA authoring, editing, and content management to generate output. You can also use the DITA OT by itself in a standalone environment for publishing.

Note: The focus of this course is to introduce the standalone DITA OT. Demonstrating DITA publishing through the oXygen XML Editor or a CCMS is beyond the scope of the course.

Most companies that publish output from DITA source content do so in conjunction with other software, such as XML editors or content management systems. However, a standalone DITA OT environment might work better for some situations, such as:

- Small content volume. If your company produces a very small amount of content and has an infrequent updating or publishing schedule, a standalone DITA OT may be more cost effective than a DITA OT-compatible tool or system.
- Pilot projects. If you're using a small subset of content as a proof of concept to show your manager the benefits of automated publishing, a standalone DITA OT is a cost effective way to help prove your point.
- Small teams. A company that employs only a very small documentation team may not need more than a standalone DITA OT for publishing.

- Students. If you're teaching yourself DITA—for a class, for career advancement, or to determine whether it's a good fit for your company—a standalone DITA OT is perfect for testing the publishing process.

Publishing with the oXygen XML Editor

Another way to publish content is by running the DITA OT using the oXygen XML Editor. oXygen performs all the same functions as the command line for DITA OT builds. However, with oXygen's graphical interface, publishing output is as easy as clicking a button. All you need to do is select the required output from a list of scenarios and apply the transformation to your source content. The publishing scenarios include the default DITA OT outputs, plus some custom scenarios, including WebHelp.

Publishing with oXygen makes sense for those in the following situations:

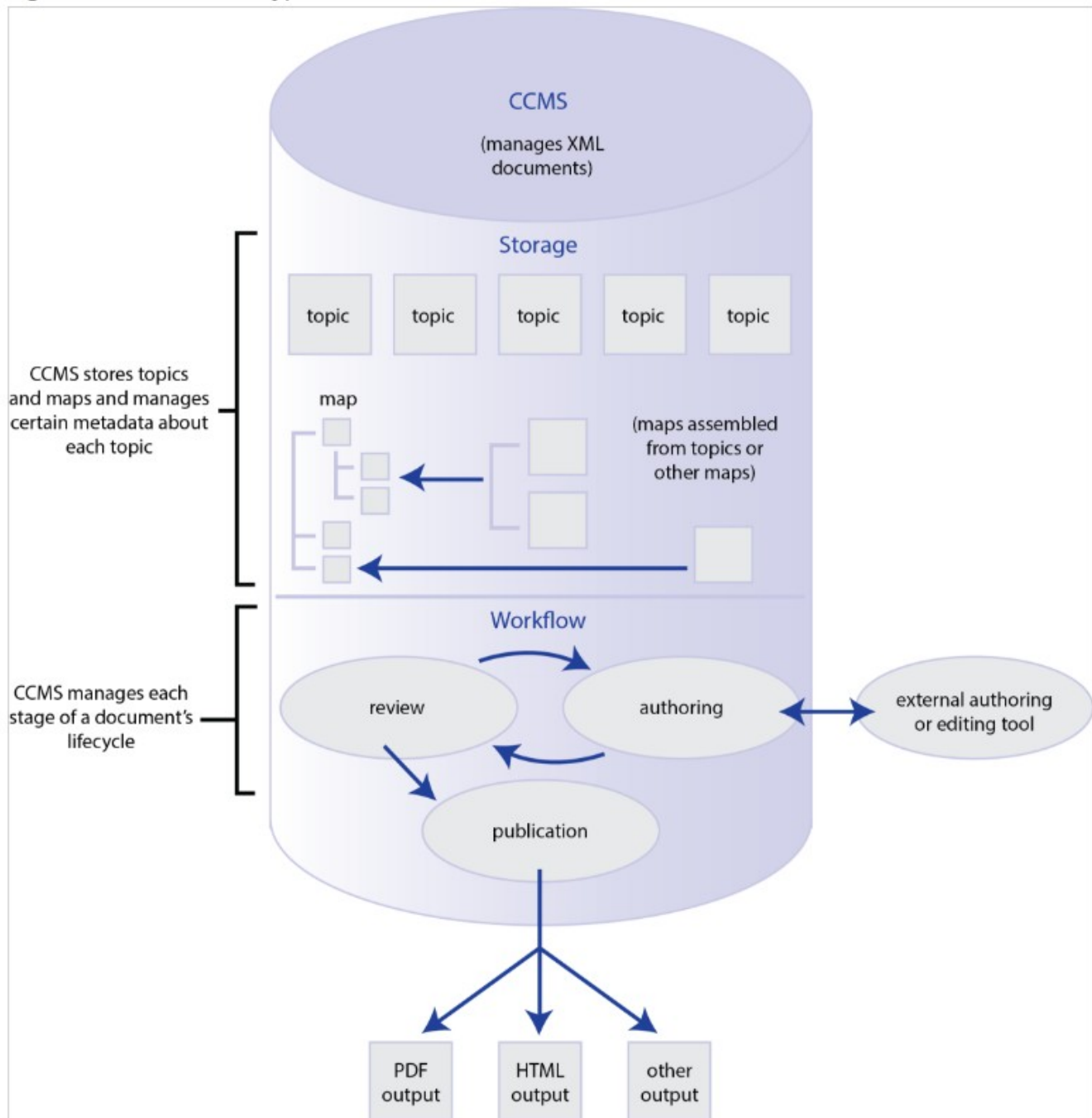
- Infrequent DITA users. If you don't publish your DITA content frequently, using oXygen's simple interface to publish output may be the right choice for you. Generating output with oXygen is less confusing and requires less effort than using the DITA OT in a standalone environment.
- Small content volume. If you don't have enough DITA content for a major CCMS installation, oXygen might be the right solution. In addition to its DITA publishing capabilities, oXygen also offers SVN integration for version control. Authors can create, edit, manage, and publish their content from a single, capable tool.
- Small content teams. For teams that are too small to justify the effort of implementing a CCMS, oXygen may be a more cost effective option. oXygen does have licensing fees, but they typically cost less than a CCMS installation.

Note: Licensing oXygen may not always be a practical investment for students or writers working on very small projects. Companies and individuals need to evaluate whether the cost of oXygen makes sense for them.

Publishing with a component content management system (CCMS)

A component content management system, or CCMS, is a software system that stores content and manages it at the topic or component level rather than at the document level. This makes a CCMS an ideal way to manage XML content. There are many CCMS options designed specifically for DITA content.

Figure 1. Overview of a typical CCMS



Different systems offer different sets of features; however, some features are common across almost all systems. Most CCMS options will connect with authoring tools (such as oXygen) and publishing tools (such as DITA OT plugins and rendering engines for PDF output). CCMS systems store DITA topics and maps and keep track of the content development workflow, including authoring, review, editing, and publishing.

Your best option for publishing may be with a CCMS if one or more of the following circumstances apply to you:

- Large content volume. If you have thousands of pages of content throughout many different documents, a CCMS will help make content storage and publishing easier. Some systems let you choose different types of installations at different price levels based on the number of topics you have. The CCMS will help you keep track of all of your DITA maps and topics and the updates you've made to them, which is especially helpful if you update frequently. A CCMS allows you to set up automatic output builds—for example, nightly, weekly, or monthly—to ensure that your published content is always up to date.
- Multiple outputs. The more outputs you need to produce, the more helpful a CCMS will be for publishing. Using a standalone DITA OT or publishing with oXygen is reasonable if you only need to generate two different PDF outputs, for example. If you need five PDF outputs, one HTML output, and two EPUB outputs, it makes more sense to use the automated publishing capabilities of a CCMS.
- Many content creators. A large content development team can benefit from the robust workflow management provided by a CCMS. The CCMS keeps track of every step of the content lifecycle, which helps content creators with version control, reviews, and approvals. The more writers, contributors, and editors you have, the more a CCMS can improve the efficiency of your content workflow.
- Reuse and localization. Managing reuse manually without support from tools is time-consuming and costly. If you're reusing a large percentage of your content, you probably have a business case for working in a CCMS. If you're also localizing content for other regions, your business case is even stronger.

Lesson 2: Setting up your DITA publishing environment

Objectives

- Download and install the DITA OT
- Set up a folder structure for your source and output files that works with the DITA OT
- Run a test build with the DITA OT

The DITA OT is an open-source, standalone publishing environment for DITA content. This lesson shows how to download and install the DITA OT and set it up to generate output.

Additional reading

[DITA Open Toolkit Overview](#)

[Installing the distribution package](#)

[Building output using the dita command](#)

[Error messages and troubleshooting](#)

Installing the DITA OT

In the previous lesson, you learned about different DITA publishing environments. The rest of this course covers publishing with a standalone DITA OT.

The following instructions show you how to download and install the DITA OT on your machine.

[Video: Installing the DITA Open Toolkit](#)

Practice

1. In a web browser, go to the DITA Open Toolkit website (<http://www.dita-ot.org/>), where the open source DITA OT is available.
2. Navigate to the Download page (<http://www.dita-ot.org/download>).

The latest DITA OT distribution package will be displayed at the top of the page. The page also contains a link to the latest version that is under development for testing, followed by a list of previous versions of the DITA OT.

Note: At the time of this course's release, the latest version of the DITA OT was 2.4.4.

3. Click on the button to download the latest distribution package of the DITA OT.



The DITA OT will be downloaded as a zip file.

4. Locate the DITA OT zip file in your system folder structure to ensure that the download completed successfully.
5. Choose a folder on your machine to install the DITA OT.

Note: Certain operations of the DITA OT don't do well with spaces or special characters in file and path names. Therefore, it's best to install the OT at the root level of your machine in a folder that only uses letters, numbers, underscores (_), or hyphens (-) in its name.

Example on Windows:

C:\dita-ot-2.4.4

Example on Mac:

/Users/[yourname]/dita-ot-2.4.4

6. Move the DITA OT zip file you downloaded into the folder you chose in the previous step.
7. Unzip the DITA OT folder.

On Windows: Right-click the zip file and click “Extract” to unzip the contents into the current location.

On Mac: Double-click the zip file. This will automatically unzip its contents into the current location.

Installing Java

The DITA OT requires Java. The DITA Open Toolkit website includes the minimum Java requirements for each version of the DITA OT.

When you try to generate output, the DITA OT will alert you if your system does not meet the Java requirements. For the DITA OT, you will most likely need to download the Java Development Kit (JDK).

On Windows, you may see the following error message if you try to run the DITA OT when Java has not been installed on your system:

“java.exe” is not recognized as an internal or external command, operable program or batch file.

On Mac, a pop-up window may appear indicating that you need a different version of Java and providing a link where you can download the correct version.

Follow the instructions below to ensure that your version of the JDK is up to date. Separate instructions are provided for Windows and Mac users.

[Video: Installing Java](#)

Practice

Installing Java on Windows

1. Download and install the latest version of the Java Development Kit (JDK) from the Oracle website (<http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html>).

Note: At the time of this course’s release, the latest version of the JDK was version 8.

2. Open File Explorer, right-click on This PC, and then click on Properties. A new window will open.
3. Click on the Advanced System Settings link in the left column of the new window. Another window will open.
4. Click on the Environment Variables button found in the lower right of the new window.
5. Find JAVA_HOME under the System Variables panel, and make sure that the value points to the directory where the JDK is installed.

If JAVA_HOME does not exist, or exists but does not display the correct location, click New. Enter JAVA_HOME under Variable name. Enter the location of the JDK (for example, C:/Program Files/Java/[jdk_folder_name]) under Variable value.

Practice

Installing Java on Mac

1. Download and install the latest version of the Java Development Kit (JDK) from the Oracle website (<http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html>).

Note: At the time of this course's release, the latest version of the JDK was version 8.

2. Click on Accept License Agreement when asked. A popup will appear if you've clicked Download without selecting Accept.
3. Under the download column select the Java link that is specific to your system. Mac users choose MAC OS X.

A .dmg file will begin downloading.

4. Once the download is complete, click on the icon in your downloads folder.

A window will appear with instructions.

5. Double click on the icon.
6. Click on Install, Run, or equivalent.

Source and output folders

Once you have installed the DITA OT and the required Java Development Kit, the next step is setting up a folder structure for testing its publishing capabilities. Establishing folders for your DITA source files and published output files will help you avoid confusion and make the process of using the DITA OT run more smoothly.

Source folders

When you're working in a standalone DITA OT publishing environment, a well-organized folder structure helps you keep track of your DITA source content. Here are some best practices for setting up your source folders:

- Don't disrupt the DITA hierarchy. The way you store your DITA content in folders may interfere with the DITA OT's ability to parse the content and generate some output types from it. Therefore, it is best to avoid storing a map file in a folder that lives below the topic files referenced in the map.
- Manage your shared content. If you have topics that are referenced in multiple maps, keep them together in a folder (named something like "shared" or "reused") where they are easy to find.

Output folders

By default, the DITA OT generates output in the DITA OT folder (previous versions of the DITA OT used a default output folder called out). The DITA OT replaces the previously generated output with new output each time you generate the same type of output from the same source content. For example, if you generated PDF output from the file `m_ducks.ditamap` yesterday and you generate PDF output of the same file today, yesterday's PDF is deleted and today's PDF replaces it. (If you generate PDF output from a different map, or generate HTML output from the same map, yesterday's PDF remains in place.)

Note: If you generate a new version of a previously generated PDF file, the DITA OT will not overwrite the previous version if you have it open in Adobe Reader or Acrobat Pro.

To make sure the DITA OT does not replace your output, set up an out folder with subfolders. That way, when you run the DITA OT, you can instruct it to generate output into the subfolder of your choice. There are many ways you can organize your output subfolders, including:

- By set of source content
- By output type
- By date
- By test iteration

Practice

Use these instructions to set up a sources folder on your system for testing.

1. Create a new folder on your machine and name it sources.
2. Locate the samples folder you downloaded at the beginning of this course.
3. Copy the ducks folder from the samples folder into the sources folder you just created.

You are now ready to test the DITA OT.

Testing the DITA OT installation

Now that you have installed the DITA OT and the required Java Development Kit and set up your folder structure, you're ready to run your first test build. The test build shows you how the DITA OT transforms DITA source files into output. Because PDF is a common output requirement for many companies that publish DITA content, the test build involves generating PDF output.

Follow the instructions below to run a test build and publish a sample PDF file using the DITA OT. Separate instructions are provided for Windows and Mac users.

Note: These instructions only apply to the 2.x versions of the DITA OT.

[Video: Testing the DITA OT](#)

Practice

1. Open a command line.

On Windows: Click on Start > Run and type cmd.

Note: If Run does not appear in the Start menu, type Run to search for it.

On Mac: Launch the Terminal application.

2. In the command line, type cd followed by the filepath to your dita-ot-2.4.4 folder.

Note: To avoid typing a filepath, locate the file or folder in Windows Explorer (on Windows) or the Finder (on Mac), then drag it directly into the command line.

3. Type the following command:

On Windows:

```
bin\dita -i [filepath]/sources/ducks/m_ducks.ditamap -f pdf2 -o out/ducks_output -v
```

On Mac:

```
./bin/dita -i [filepath]/sources/ducks/m_ducks.ditamap -f pdf2 -o out/ducks_output -v
```

Note: Typing `-o out/ducks_output` generates an output folder and subfolder called `out/ducks_output` inside the DITA OT folder.

4. Press ENTER to run the DITA OT.

Note: The command line displays a log of the DITA OT's build process.

5. When the build has finished, look in the `ducks_output` folder to see your generated output. You should see a file called `m_ducks.pdf`.

Note: If you do not see any output, your build was likely not successful. This course covers troubleshooting for build failures in a subsequent lesson.

Keep the file `m_ducks.pdf` in your `ducks_output` folder for use later in this course.

Lesson 3: Generating output

Objectives

- Generate basic PDF output using the DITA OT
- Generate basic HTML output using the DITA OT
- Troubleshoot common problems with generating output using the OT

This lesson shows how to publish DITA content in PDF and HTML output formats using the DITA OT.

Additional reading

[Plugins for the DITA Open Toolkit](#)

[DITA Open Toolkit Parameter Reference](#)

[Webcast: Pros and cons of the DITA Open Toolkit](#)

[Demystifying DITA to PDF Publishing](#)

[DITA vs. XHTML for Producing Content](#)

DITA OT commands and parameters

In the previous lesson, you tested the DITA OT by entering the following command into the command line:

On Windows:

```
bin\dita -i [filepath]/sources/m_ducks.ditamap -f pdf2 -o [output-dir] -v
```

On Mac:

```
./bin/dita -i [filepath]/sources/m_ducks.ditamap -f pdf2 -o [filepath]/outputs/  
m_ducks_output -v
```

These are the standard commands and parameters of a basic DITA OT build. Here is what these commands and parameters mean:

- `bin\dita` (Windows) or `./bin/dita` (Mac) is the command that generates output from DITA
- `-i` (the input parameter) indicates that the filepath that follows it to a DITA map or file needs to be transformed into output.
- `-f` (the transtype parameter) indicates that the string that follows it is the output transformation type (also called the “transtype”). Some common transtype values include “pdf2” for PDF output, “xhtml” for HTML output, and “webhelp” for WebHelp output.

- -o (the output parameter) indicates where to store the generated output. This parameter is optional; if you do not use it to specify an output directory, the output is placed in the DITA OT's default "out" folder.
- -v (the verbose parameter) indicates verbose output, which displays a log of the DITA OT build in the command window. This parameter is optional, but it is useful to watch the DITA OT run in realtime to help troubleshoot errors.

You can use additional parameters to add more specifications about the build you're running. To explore all the options available to you, see the [DITA Open Toolkit Parameter Reference](#).

Generating PDF output

In an earlier lesson, your first test build with the DITA OT used the pdf2 plugin to generate PDF output. In this lesson, you will see how the pdf2 plugin handles maps and bookmaps differently.

For map content, the pdf2 plugin generates a cover page and table of contents by default.

For bookmap content, the pdf2 plugin generates a cover page. Any subsequent structures in the output reflect the specifications set in the bookmap. For example, the pdf2 plugin will generate a table of contents if the bookmap uses the <toc> element in the <booklists> element to specify that there should be one; however, if the bookmap does not contain the <toc> element, the pdf2 plugin will not build one.

During your first DITA OT test build, you generated the file m_ducks.pdf from content referenced from a map. The following instructions show you how to generate PDF output from the same content stored in a bookmap and compare the two different output files.

Practice

1. Type the following command into the command line:

On Windows:

```
bin\dita -i [filepath]/sources/ducks/b_ducks.ditamap -f pdf2 -o out/ducks_output -v
```

On Mac:

```
./bin/dita -i [filepath]/sources/ducks/b_ducks.ditamap -f pdf2 -o out/ducks_output -v
```

2. Press ENTER to run the DITA OT.
3. Navigate to your ducks_output folder and locate the PDF file you just generated, b_ducks.pdf.
4. Open b_ducks.pdf in a PDF viewer and compare it with m_ducks.pdf, which you generated previously. Note the differences between the two outputs.

The file m_ducks.pdf includes a table of contents for the entire document:

Contents

Wild ducks.....	3
Types of wild ducks.....	3
Wild duck species.....	3
Watching wild ducks.....	3
Domestic ducks.....	4
Duckling growth and development.....	5
Feeding ducklings.....	6
Duck weight.....	7
Using the Duck Database.....	7
Writing about ducks.....	7
tNav.....	8

By contrast, the file b_ducks.pdf includes a mini-table of contents for each chapter:

Chapter

1

Wild ducks

Topics:

- [Types of wild ducks](#)
- [Wild duck species](#)
- [Watching wild ducks](#)

North America is home to a variety of wild ducks.

Generating HTML output

The DITA OT can produce HTML output using the xhtml and html5 plugins. This lesson uses the xhtml plugin to demonstrate the process of publishing HTML output.

To generate XHTML output, use the same content as the PDF exercise. Seeing the same content in two different output types gives you a firsthand look at the way DITA separates content from formatting.

[Video: Generating HTML Output](#)

Practice

1. Type the following commands into the command line:

On Windows:

```
bin\dita -i [filepath]/sources/ducks/m_ducks.ditamap -f xhtml -o out/ducks_output -v
```

On Mac:

```
./bin/dita -i [filepath]/sources/ducks/m_ducks.ditamap -f xhtml -o out/ducks_output -v
```

2. Press ENTER to run the DITA OT.
3. Navigate to your ducks_output folder.

In addition to the PDF files you generated earlier, you should now see multiple XHTML files and an images folder.

4. Open the file index.html in a browser.

This index file serves as an electronic table of contents, showing the hierarchy of your map with links to each topic file. The DITA OT's default xhtml plugin does not provide navigation, but the index file can help you find the topics you need.

5. Open the file c_duckling_growth.html in a browser to see how the xhtml plugin displays images and tables.

The default XHTML output looks like this:

Duckling growth and development



Figure 1. Ducklings running

Age	Milestone
7 weeks	Attempt flight for the first time
12-14 weeks	Reach adult body weight
1 year	Capable of reproduction

Table 1. Typical development of mallard ducks

Age	Milestone
7 weeks	Attempt flight for the first time
12-14 weeks	Reach adult body weight
1 year	Capable of reproduction

Parent topic: [Domestic ducks](#)

Note: Fonts and spacing may vary depending on which browser and version you are using.

Troubleshooting common issues with generating output

Many people encounter problems with the DITA OT when running a build for the first time. In the DITA OT's log file, a problem is typically indicated by the message "build failed." However, you may also have a successful build but find issues with the output. Here are some common problems and tips for troubleshooting them:

- Incorrectly typed commands. If your build fails, check your commands for extra spaces, transposed letters, missing parameters, or other typographical errors.
- Wrong version of Java. If you do not have the correct version of Java, the log file will show that the build failed or even refused to run. Follow the previous instructions in this lesson on installing Java to solve this issue.
- Invalid source content. If one of the topic files in your map or the map file itself is invalid, the DITA OT may not be able to build the output. If you encounter a build failure and the log shows errors relating to a particular source file, open that file and verify that its structure is valid.
- Output missing. If the log indicates that the build was successful, but you are unable to find your output, double check your -o parameter to make sure you specified an output location. If you did not, the output will be built in the DITA OT folder.
- Output replaced. Every time you run the DITA OT with the same set of commands on the same source files, your most recent build will replace your previous build. If you wish to keep each iteration of output you build (for example, to keep track of testing or troubleshooting results), you can either rename the previous output before you run a new build or set the -o parameter to a different output directory every time you run the build.
- Incorrect appearance of output. Sometimes your output doesn't look the way you expected, even if it builds successfully. For example, the images in a PDF may be so large they run off the page, or the columns in a table may not be optimal for text display. These issues are the result of not specifying measurements for your images and tables in the source files.

To control the specific size of your images, you can use the <image> element's width attribute.

Domestic ducks | 6




Figure 1: Ducklings running

Age	Milestone
7 weeks	Attempt flight for the first time
12-14 weeks	Reach adult body weight
1 year	Capable of reproduction

Table 1: Typical development of mallard ducks

Age	Milestone
7 weeks	Attempt flight for the first time
12-14 weeks	Reach adult body weight
1 year	Capable of reproduction

Feeding ducklings

Ducklings need **round-the-clock** access to both *food* and *water* during the first two weeks of their lives.
 Ducklings need *round-the-clock* access to both *food* and *water* during the first two weeks of their lives.
 Because duck feed has a dry, gritty texture, ducklings need plenty of H₂O to keep their bills clean.

Domestic ducks | 6




Figure 2. Image with a width specified vs. image with no width specified in PDF output

For tables, you can use the <colspec> element to adjust the column widths from their default setting of equal widths into widths that work better for your content.

Note: Visual XML editors (such as oXygen) allow you to modify table column widths through a graphical interface that manages the <colspec> values for you.

Table 1: Typical development of mallard ducks

Age	Milestone
7 weeks	Attempt flight for the first time
12-14 weeks	Reach adult body weight
1 year	Capable of reproduction

Table 1: Typical development of mallard ducks

Age	Milestone
7 weeks	Attempt flight for the first time
12-14 weeks	Reach adult body weight
1 year	Capable of reproduction

Figure 3. Table with colspecs adjusted vs. table with default colspecs in PDF output

More complex issues require deeper troubleshooting. If you encounter a build failure that doesn't match one of these common issues, use the DITA OT's log for your build as a starting point for identifying the issue. Once you know what's causing the problem, you may need to run more test builds to continue troubleshooting it until you can resolve it. The DITA OT's documentation on log files (<http://www.dita-ot.org/2.4/user-guide/log-files.html>) is a helpful resource for troubleshooting.

Note: Unfortunately, we cannot offer one-on-one technical support for this free site.

Lesson 4: Custom plugins

Objectives

- Install a custom plugin into the DITA OT environment
- Generate output using a custom DITA OT plugin
- Update the custom plugin to make simple changes to the appearance of your output
- Evaluate your need for custom DITA OT plugins

This lesson shows how to install, use, and make simple modifications to a custom DITA OT plugin, and evaluate your need for customized output.

Additional reading

[Hacking the DITA Open Toolkit](#)

[Creating DITA-OT plug-ins](#)

[Installing plug-ins](#)

[Publishing XHTML output as a frameset](#)

[But what if I want a whole new transform dot dot dot?](#)

Why customize plugins?

The DITA OT comes with default plugins for generating several output types. As you saw in the previous lesson, these include the pdf2 plugin for PDF output and the xhtml plugin for HTML output. The default plugins are best considered as demos, or the basis for building “production” plugins.

The default plugins are helpful when you’re learning to publish content using the DITA OT. However, they are rarely used for publishing in real-world situations.

Here are some reasons why the default plugins don’t usually work for companies that publish content:

- Look and feel. The default plugins provided with the DITA OT use very basic, generic styling. Companies need the output to match the look and feel of their branding. The output needs to use company colors and fonts, incorporate the company logo, and include images or symbols associated with the company brand.

Even for content where there are no corporate branding requirements—for example, academic articles, research, reports, or internal content—the group publishing the content will probably want to improve on the look and feel of the default plugins.

- Functionality. In some cases, the default plugins cannot support certain aspects of the source content’s structure. If you’re using a DITA specialization, the default plugins will probably not display your content in the output as you intended.

- Output type. You may need to generate a type of output that isn't included in the DITA OT's default plugins. One example of this is companies that need to send their DITA source content into InDesign for complex, highly-designed formatting before publishing it to PDF output. The DITA OT does not provide a default DITA-to-InDesign plugin.

If the DITA OT's default plugins don't suit your needs, you can install custom plugins and use them to generate output. Some custom plugins are based on the underlying structure of the DITA OT's default plugins; others, such as DITA-to-InDesign plugins, are created independently.

You can develop custom plugins yourself, find them on the web, or hire or contract with someone to develop them. Custom plugins are often developed by consultants as part of a content strategy solution.

When you're evaluating your need for custom plugins, ask yourself:

- Whether your customization requirements are based on look and feel, structure, or both
- How the structure of your source content affects the DITA OT's publishing capabilities
- What types of output you need and how much they differ from the DITA OT's default outputs

Installing a custom plugin

This lesson shows how to install a custom plugin and generate output from it using the DITA OT. The custom plugin used in this lesson is called `ld_xhtml`, and is based on the default `xhtml` plugin. The `ld_xhtml` plugin modifies the behavior of the `xhtml` plugin and adds links to its own CSS file.

[Video: Installing Custom Plugins](#)

Practice

1. Locate the zip file `com.learningdita.ld_xhtml.zip` in the `samples/plugins` folder you downloaded at the beginning of this course.
2. Type the following commands into the command line:

On Windows:

```
bin\dita -install [filepath]/com.learningdita.ld_xhtml.zip -v
```

On Mac:

```
./bin/dita -install [filepath]/com.learningdita.ld_xhtml.zip -v
```

Note: The `-install` parameter integrates and installs the custom `xhtml` plugin and enables the DITA OT to generate output from it.

3. Press ENTER to run the DITA OT.
4. To test the integration by generating output, type the following commands into the command line:

On Windows:

```
bin\dita -i [filepath]/sources/ducks/m_ducks.ditamap -f ld_xhtml -o out/custom_output -v
```

On Mac:

```
./bin/dita -i [filepath]/sources/ducks/m_ducks.ditamap -f ld_xhtml -o out/custom_output -v
```

5. Press ENTER to run the DITA OT.
6. Look in the dita-ot-2.4.4/out/custom_output folder to see the generated XHTML output.
7. Open the file c_wild_duck_species.html in a browser. Compare it to the default version of c_wild_duck_species.html (in your dita-ot-2.4.4/out/ducks_output folder) that you generated previously.

The default XHTML output looks like this:

Wild duck species

The longest species of dabbling ducks in North America are:

1. Northern pintail
2. Mallard
3. American black duck

Note: Although the northern pintail is the longest dabbling duck, the mallard is generally considered the largest because of its heavier body weight.

Parent topic: [Wild ducks](#)

The custom XHTML output looks like this:

Wild duck species

The longest species of dabbling ducks in North America are:

1. Northern pintail
2. Mallard
3. American black duck

Note: Although the northern pintail is the longest dabbling duck, the mallard is generally considered the largest because of its heavier body weight.

Parent topic: [Wild ducks](#)

The custom XHTML output has different fonts and colors than the default XHTML output.

Updating a custom plugin

One of the most common reasons for updating the styling of a custom plugin is a change in company branding. You may also need to update a custom plugin to accommodate changes to the source content's structure—for example, adding a new element to your specialization, or specializing for the first time.

Customizing DITA OT plugins is difficult and requires lots of testing to ensure that updates to the plugin have the desired results in the output. Special knowledge is required to customize plugins or update custom plugins, such as knowledge of Ant, XSL, and DTDs for the base plugin architecture, plus other knowledge of the output types, including XHTML, JavaScript, and XSL-FO. For this reason,

most companies hire an expert—whether an outside consultant or an in-house employee—to customize plugins.

CSS, or cascading stylesheets, can be used to modify the appearance of output from the xhtml plugin. Because many people have common knowledge of CSS, a company may not need to hire an expert if the only required output type is XHTML and all branding aspects can be handled through changes to the CSS.

The following instructions show you how to make simple updates to the CSS for your custom ld_xhtml plugin and generate XHTML output to test the results.

Practice

1. Navigate to your dita-ot-2.4.4/out/custom_output folder and open the file c_wild_duck_species.html in a browser.

The XHTML output looks like this:

Wild duck species

The longest species of dabbling ducks in North America are:

1. Northern pintail
2. Mallard
3. American black duck

Note: Although the northern pintail is the longest dabbling duck, the mallard is generally considered the largest because of its heavier body weight.

Parent topic: [Wild ducks](#)

2. Navigate to your dita-ot-2.4.4/plugins/com.learningdita.ld_xhtml/resource folder and open the file ld_style.css in a text editor.
3. Make the following changes to ld_style.css (areas where changes are required are indicated in bold, highlighted text):

```
.p {
  margin-top: 1em;
  font-family: "Lingua Franca", Palatino, serif;
  font-size: 0.9em;
  color: #D48028;
}
.topictitle1 {
  margin-top: 0;
  margin-bottom: .1em;
  font-size: 2em;
  font-family: "Gill Sans", Helvetica, sans-serif;
  font-weight: normal;
  color: #86CCCC;
}
```

4. Save your changes to `ld_style.css`.
5. Enter the following commands into the command line to test your updates to the custom plugin:

On Windows:

```
bin\dita -i [filepath]/sources/ducks/m_ducks.ditamap -f ld_xhtml -o out/  
custom_output -v
```

On Mac:

```
./bin/dita -i [filepath]/sources/ducks/m_ducks.ditamap -f ld_xhtml -o out/  
custom_output -v
```

6. Press ENTER to run the DITA OT.
7. Look inside your `custom_output` folder to view your generated custom XHTML output.
8. Open `c_wild_duck_species.html` in a browser to see the CSS changes you made.

The XHTML output now looks like this:

Wild duck species

The longest species of dabbling ducks in North America are:

1. Northern pintail
2. Mallard
3. American black duck

Note: Although the northern pintail is the longest dabbling duck, the mallard is generally considered the largest because of its heavier body weight.

Parent topic: [Wild ducks](#)

Best practices for custom plugins

If you use custom DITA plugins, eventually they will probably need to be updated or refreshed. You might need to make further updates to the appearance of your custom plugins' output and test those changes. You may also need to add new custom plugins as your output requirements grow.

Here are some best practices for maintaining custom DITA plugins:

- Establish a consistent way to collect specifications. When you're developing a new custom plugin, you will need to gather specifications for how the output should look. For print-based outputs such as PDF, these specs typically include fonts, sizes, colors, page margins, rules for image and table display, placement of running headers and footers, and so on. For online formats such as HTML, these specs are also function-based; for example, what kind of navigation and search will work best for your audience?

Creating a custom plugin will be easier if you have a clear and consistent method for gathering these specs. A good starting point is a checklist or questionnaire that your company's marketing department

can answer to provide specs about the look and feel of the output. Once you have these specs and have created your first draft of the custom plugin, you will most likely need several rounds of testing and review to fine-tune the output. The better specs you have before you begin developing the plugin, the more efficient the testing process will be.

- Support special structures; don't misuse structures. A custom plugin is a great way to ensure that a specialized DITA structure looks its best in the various required output formats. However, custom plugins should not be used to support "tag abuse" or purposeful misuse of the DITA structure, standard or specialized.

For example, it may be tempting to create hundreds of outputclasses in the DITA source to represent exceptions to the PDF styling, and hundreds of matching rules in the custom PDF plugin to display those outputclasses. However, this only makes your DITA source content and custom plugin more complex and difficult to maintain. Adding lots of formatting overrides to the source also defeats the purpose of separating content from formatting for automation.

- Document custom plugin updates. When you need to make a change to the custom plugin, add a comment with your name or company name, the date, and a brief explanation of your update. Comment out what you're changing in the plugin rather than replacing it completely; you never know when you may be asked to revert a change back to its original state. If you ever need to transfer your custom plugins over to another developer, this will show them all the changes to the plugin over time.

In addition to documenting changes, it can also be helpful to keep the specs you gathered during plugin development as documentation. The result will be a style guide you can use to ensure your output looks correct as you create new DITA content and generate test output before final publication.

Course IX. LearningDITA Live 2018 Recordings

Beginner sessions

DITA overview

New to DITA? Not sure what all the fuss is about? In this session, you'll learn DITA basics so that the rest of LearningDITA Live will make sense to you. No prior knowledge of DITA or XML is required for this session.

<https://youtu.be/N--DU9IGxq0>

You got DITA in my minimalism! You got minimalism in my DITA!

Although you don't need DITA to write according to minimalism principles and you can certainly take a non-minimal approach when writing in DITA, the two go together like chocolate and peanut butter. In this presentation, Dawn explores the relationship between DITA, minimalism, and other technical communication best practices, demonstrating how each aligns with and complements the other. She provides guidance for forming your DITA information model and authoring guidelines to reflect writing best practices while maximizing the benefits DITA offers.

https://youtu.be/O73_QGTqC-8

Teaching (and learning) with DITA at the college level

The professor is the student in this adventure, twelve years in the making, of teaching DITA as an authoring standard and methodology for technical documentation. The cast features several generations of Professional and Technical Writing students in a Department of English. There is drama about XML and XSLT code, but also comedy over successful transformations, suspense because of new software tools, and plain terror due to incomprehensible error messages.

Teaching DITA is a constant learning process with ups and downs. This session summarizes ideas and approaches for conquering the standard without getting lost inside XML tags.

<https://youtu.be/stTHyhoO3mc>

Getting started with DITA and Adobe FrameMaker 2017 in less than one hour

It's often heard: DITA is difficult and expensive and "only for the big ones." But that's not true. Getting started with DITA can be very easy and fast and does not bust your budget at all. In this session, Stefan Gentz, Worldwide Technical Communication Evangelist at Adobe, will give a kick start to DITA authoring and publishing.

You will learn how to create your first topic with typical technical documentation elements like 3D graphics, images and video, tables, links. You will structure the content and attribute it to multiple audiences and finally publish it to PDF and HTML5 And all this in less than one hour! And if we want to go bigger, we can push our freshly created DITA Content into Adobe Experience Manager and take advantage of Adobe's enterprise-class DITA CCMS with full online DITA Editing in the browse, online review, translation management and much more.

This session was canceled due to a scheduling conflict. Adobe intends to provide a post-event recording.

Intermediate sessions

The business value of DITA

Content creators tend to focus on technical aspects of DITA, like formatting automation and the enforcement of structure patterns. But business leaders care about recurring costs and time to market. In this presentation, Sarah describes how to communicate the value of DITA in terms that your executives will understand: accelerating time to market, reducing recurring costs, and improving the customer experience.

If you believe that your organization needs DITA, but are struggling to explain the rationale to your leadership, this is the session for you.

<https://youtu.be/8r6Ma32X3s0>

Improving the quality of your DITA documentation

An exploration of various ways to improve the overall quality of DITA documentation.

The methods that we will discuss include: integrating a style guide into your writing process, imposing Schematron rules to help maintain consistency and accuracy throughout a documentation team, using Schematron quick fixes to offer automatic proposals that will help authors solve documentation problems, and several ways to offer inline hints to help the author know what type of content is expected in a certain context.

<https://youtu.be/N8iRpQJKQvk>

Localization strategy for DITA

Many organizations move to DITA primarily to resolve localization issues. Content reuse and multichannel publishing can expedite publishing and reduce the amount of content that requires translation.

If you need to publish content in multiple languages and are working in DITA or thinking about it, this session is for you. We will discuss the benefits and caveats of localizing DITA content and best practices for localization.

<https://youtu.be/C-Wr4MuN17I>

Managing DITA projects in the real world

A DITA implementation is not just a matter of picking tools. In fact, software selection is usually the easiest part of a DITA project. To ensure project success, you need to identify and align the requirements of affected departments. Learn what it really takes to implement DITA: managing people, content modeling, evaluating tools and vendors, understanding outputs, considering conversion, and more.

https://youtu.be/Apalg06l_AM

Resources

- [XML calculator](#)
- [Change management](#)
- [Delivery format specifications](#)
- [Best practices](#)
- [Conversion tips](#)
- [FOP deficiencies](#)
- [DITA skills](#)
- Free DITA training
 - [English](#)
 - [German](#)
 - [Chinese](#)
- [Podcast on DITA projects](#)

Faster content, better healthcare: improving cancer diagnostics with electronic delivery

Quick access to cancer staging information leads to faster diagnoses and improves patient care—which is a challenge when the American Joint Committee on Cancer (AJCC) Cancer Staging Manual is only available in print. What if the manual’s content were integrated into electronic medical records to help doctors find the information they need via API instead? This case study shows how Scriptorium worked with the AJCC to build a DITA specialization that allowed them to implement such a solution.

<https://youtu.be/YfEMWDxt8rI>

Starting small with structured content management

Getting started with structured content management can be daunting if you've been tasked with completely overhauling your organization's content strategy. There are so many things to consider that deciding where to begin can seem like one of your biggest challenges. There are plenty of clichés appropriate to this situation: the journey of a thousand miles begins with a single step, don't try to boil the ocean, and so on. By thinking about the project as a whole, it's easy to become overwhelmed. In this presentation, I'll give you some ideas for getting started in small, manageable steps, without ever losing sight of your vision.

<https://youtu.be/0SOv1b93mys>

Delivering a consistent content experience. Everywhere. Everyday.

Increasing presence in global markets is exciting for any company, but does not come without growing pains. Crown Equipment Corporation needed to expedite content authoring and translation while maintaining consistency across technical communication, training, user and software content, in dozens of languages for different geographic markets.

<https://youtu.be/RuRrKXVcaFk>

When conversion makes sense

This session will discuss how to design a conversion to DITA for maximum effectiveness. We'll talk about what's automatable, how to decide on automation vs. a manual process vs. re-authoring, and how to identify redundant content in your document so that it can be standardized in modules. You'll gain insight on when you should do it yourself, what trouble areas are likely to show, and the value of planning ahead.

<https://youtu.be/oq7u0HsxqtA>

Advanced sessions

DITA for agile documentation

Based on a project in the automotive industry, this case study describes authoring and delivering developer documentation for an agile project. It shows how to integrate technical writing with the infrastructure and processes of the development and test teams.

Ulrike will also show how to combine information from different sources in content delivery, including DITA-based documentation, test cases, API reference, and JIRA issues. Using metadata and semantic DITA elements, the system generates relationships among information items that add semantic value to the content delivery and enable search filters and facets.

https://youtu.be/u1AeO_J7d6A

DITA specialization overview

Specialization in DITA enables you to create new designs (topic types, elements, attributes) from existing designs. It is one of the cornerstones of DITA. The flexibility introduced by specialization is one of the reasons for DITA's success. This session explores the various reasons for employing DITA specialization, the different types of specializations that are possible, and how to plan specializations.

<https://youtu.be/bHj0tvJFNWQ>

Developing learning content in DITA

DITA includes a Learning and Training specialization for learning content. In this session, Simon describes how to use the L&T topics and elements to plan, create, organize, and publish instructional content. The session also covers the advantages and disadvantages of creating DITA-based learning content.

<https://youtu.be/IfzNo5CxKMY>

InDesign and DITA

For most PDF requirements, we can publish from DITA via XSL-FO. DITA and InDesign provides an alternative when the final PDF or print output needs custom formatting that goes beyond what we can do in an automated workflow. In this session, Jake describes what factors require InDesign-based publishing, and provides an overview of the process.

<https://youtu.be/ms1w8E0WIKa>