

Eberlein Consulting

DITA training, information architecture, and
strategy

Architectural specification: Base

Contents

- Introduction to DITA..... 5**
 - DITA terminology and notation.....5
 - Basic concepts.....9
 - File extensions.....10
 - Producing different deliverables from a single source.....11

- DITA markup12**
 - DITA topics.....12
 - The topic as the basic unit of information.....12
 - The benefits of a topic-based architecture.....12
 - Disciplined, topic-oriented writing.....13
 - Information typing.....14
 - Generic topics.....14
 - Topic structure.....15
 - Topic content.....16
 - DITA maps.....17
 - Definition of DITA maps.....17
 - Purpose of DITA maps.....17
 - DITA map elements.....18
 - DITA map attributes.....20
 - Examples of DITA maps.....23
 - Subject scheme maps and their usage.....27
 - Subject scheme maps.....27
 - Defining controlled values for attributes.....27
 - Binding controlled values to an attribute.....28
 - Processing controlled attribute values.....29
 - Extending subject schemes.....30
 - Scaling a list of controlled values to define a taxonomy.....31
 - Classification maps.....32
 - Examples of subject scheme maps.....32
 - DITA metadata.....37
 - Metadata elements.....37
 - Metadata attributes.....38
 - Metadata in maps and topics.....40
 - Cascading of metadata attributes in a DITA map.....41
 - Reconciling topic and map metadata elements.....43
 - Map-to-map cascading behaviors.....46
 - Context hooks and window metadata for user assistance.....49

- DITA addressing.....51**
 - ID attribute.....51
 - DITA linking.....52
 - URI-based (direct) addressing.....53
 - Indirect key-based addressing.....55
 - Core concepts for working with keys.....55

Key scopes.....	57
Using keys for addressing.....	58
Addressing keys across scopes.....	59
Cross-deliverable addressing and linking.....	60
Processing key references.....	62
Processing key references for navigation links and images.....	62
Processing key references on <topicref> elements.....	63
Processing key references to generate text or link text.....	63
Examples of keys.....	65

DITA processing.....80

Navigation.....	80
Table of contents.....	80
Indexes.....	80
Content reference (conref).....	81
Conref overview.....	81
Processing conrefs.....	82
Processing attributes when resolving conrefs.....	82
Processing xrefs and conrefs within a conref.....	83
Conditional processing (profiling).....	85
Conditional processing values and groups.....	85
Filtering.....	86
Flagging.....	88
Conditional processing to generate multiple deliverable types.....	88
Examples of conditional processing.....	89
Branch filtering.....	91
Overview of branch filtering.....	91
Branch filtering: Single condition set for a branch.....	91
Branch filtering: Multiple condition sets for a branch.....	92
Branch filtering: Impact on resource and key names.....	92
Branch filtering: Implications of processing order.....	95
Examples of branch filtering.....	96
Chunking.....	105
Using the @chunk attribute.....	105
Chunking examples.....	107
Translation and localization.....	110
The @xml:lang attribute.....	110
The @dir attribute.....	112
Processing documents with different values of the @domains attribute.....	113
Sorting.....	114

Configuration, specialization, generalization, and constraints 115

Overview of DITA extension facilities.....	115
Configuration.....	116
Overview of document-type shells.....	116
Rules for document-type shells.....	117
Equivalence of document-type shells.....	117
Conformance of document-type shells.....	118
Specialization.....	118
Overview of specialization.....	118
Modularization.....	119

- Vocabulary modules..... 119
- Specialization rules for element types..... 120
- Specialization rules for attributes..... 121
- @class attribute rules and syntax..... 121
- @domains attribute rules and syntax..... 122
- Specializing to include non-DITA content..... 125
- Sharing elements across specializations..... 127
- Generalization..... 127
 - Overview of generalization..... 127
 - Element generalization..... 128
 - Processor expectations when generalizing elements..... 128
 - Attribute generalization..... 130
 - Generalization with cross-specialization dependencies..... 131
- Constraints..... 131
 - Overview of constraints..... 131
 - Constraint rules..... 132
 - Constraints, processing, and interoperability..... 133
 - Weak and strong constraints..... 133
 - Conref compatibility with constraints..... 133
 - Examples: Constraints..... 136

Coding practices for DITA grammar files..... 142

- Recognized XML-document grammar mechanisms..... 142
- Normative versions of DITA grammar files..... 142
- DTD coding requirements..... 143
 - DTD: Overview of coding requirements..... 143
 - DTD: Coding requirements for document-type shells..... 144
 - DTD: Coding requirements for element type declarations..... 147
 - DTD: Coding requirements for structural modules..... 150
 - DTD: Coding requirements for element domain modules..... 151
 - DTD: Coding requirements for attribute domain modules..... 152
 - DTD: Coding requirements for constraint modules..... 152
- RELAX NG coding requirements..... 154
 - RELAX NG: Overview of coding requirements..... 154
 - RELAX NG: Coding requirements for document-type shells..... 155
 - RELAX NG: Coding requirements for element type declarations..... 158
 - RELAX NG: Coding requirements for structural modules..... 160
 - RELAX NG: Coding requirements for element domain modules..... 162
 - RELAX NG: Coding requirements for attribute domain modules..... 163
 - RELAX NG: Coding requirements for constraint modules..... 164
- XML Schema coding requirements..... 165
 - XML Schema: Overview and limitations of coding requirements..... 165
 - XML Schema: Coding requirements for document-type shells..... 166
 - XML Schema: Coding requirements for element type declarations..... 169
 - XML Schema: Coding requirements for structural modules..... 170
 - XML Schema: Coding requirements for attribute domain modules..... 171
 - XML Schema: Coding requirements for constraint modules..... 171

Introduction to DITA

The Darwin Information Typing Architecture (DITA) is an XML-based architecture for authoring, producing, and delivering topic-oriented, information-typed content that can be reused and single-sourced in a variety of ways. While DITA historically has been driven by the requirements of large-scale technical documentation authoring, management, and delivery, it is a standard that is applicable to any kind of publication or information that might be presented to readers, including interactive training and educational materials, standards, reports, business documents, trade books, travel and nature guides, and more.

DITA is designed for creating new document types and describing new information domains based on existing types and domains. The process for creating new types and domains is called specialization. Specialization enables the creation of specific, targeted XML grammars that can still use tools and design rules that were developed for more general types and domains; this is similar to how classes in an object-oriented system can inherit the methods of ancestor classes.

Because DITA topics are conforming XML documents, they can be readily viewed, edited, and validated using standard XML tools, although realizing the full potential of DITA requires using DITA-aware tools.

DITA terminology and notation

The DITA specification uses specific notation and terms to define the components of the DITA standard.

Notation

The following conventions are used throughout the specification:

attribute types

Attribute names are preceded by @ to distinguish them from elements or surrounding text, for example, the @props or the @class attribute.

element types

Element names are delimited with angle brackets (< and >) to distinguish them from surrounding text, for example, the <keyword> or the <prolog> element.

In general, the unqualified use of the term *map* or *topic* can be interpreted to mean "a <map> element and any specialization of a <map> element" or "a <topic> element or any specialization of a <topic> element." Similarly, the unqualified use of an element type name (for example, <p>) can be interpreted to mean the element type or any specialization of the element type.

Normative and non-normative information

The DITA specification contains normative and non-normative information:

Normative information

Normative information is the formal portion of the specification that describes the rules and requirements that make up the DITA standard and which must be followed.

Non-normative information

Non-normative information includes descriptions that provide background, examples, notes, and other useful information that are not formal requirements or rules that must be followed.

All information in the specification should be considered normative unless it is an example, a note, an appendix, or is explicitly labeled as non-normative. The DITA specification contains examples to help clarify or illustrate

specific aspects of the specification. Because examples are specific rather than general, they might not illustrate all aspects or be the only way to accomplish or implement an aspect of the specification. Therefore all examples are non-normative.

Basic DITA terminology

The following terminology is used to discuss basic DITA concepts:

DITA document

An XML document that conforms to the requirements of this specification. A DITA document *MUST* have as its root element one of the following elements:

- `<map>` or a specialization of the `<map>` element
- `<topic>` or a specialization of the `<topic>` element
- `<dita>`, which cannot be specialized, but which allows documents with multiple sibling topics

DITA document type

A unique set of structural modules, domain modules, and constraint modules that taken together provide the XML element and attribute declarations that define the structure of DITA documents.

DITA document-type shell

A set of DTD, XSD, or RELAX NG declarations that implement a DITA document type by using the rules and design patterns that are included in the DITA specification. A DITA document-type shell includes and configures one or more structural modules, zero or more domain modules, and zero or more constraint modules. With the exception of the optional declarations for the `<dita>` element and its attributes, DITA document-type shells do not declare any element or attribute types directly.

DITA element

An XML element instance whose type is a DITA element type. DITA elements must exhibit a `@class` attribute that has a value that conforms to the rules for specialization hierarchy specifications.

DITA element type

An element type that is either one of the base element types that are defined by the DITA specification, or a specialization of one of the base element types.

map instance

An occurrence of a map type in a DITA document.

map type

A map or a specialization of map that defines a set of relationships among topic instances.

structural type instance

An occurrence of a topic type or a map type in a DITA document.

topic instance

An occurrence of a topic type in a DITA document.

topic type

A topic or a specialization of topic that defines a complete unit of content.

Specialization terminology

The following terminology is used to discuss DITA specialization:

base type

An element or attribute type that is not a specialization. All base types are defined by the DITA specification.

extension element

Within a vocabulary module, an element type that can be extended, replaced, or constrained for use in a DITA document type.

generalization

The process by which a specialized element is transformed into a less-specialized ancestor element or a specialized attribute is transformed into a less-specialized ancestor attribute. The original specialization-hierarchy information can be preserved in the generalized instance; this allows the original specialized type to be recreated from the generalized instance.

specialization

- (1) The act of defining new element or attribute types as a semantic refinement of existing element or attribute types
- (2) An element or attribute type that is a specialization of a base type
- (3) A process by which a generalized element is transformed into one of its more specialized element types or a generalized attribute is transformed into a more specialized attribute.

specialization hierarchy

The sequence of element or attribute types, from the most general to most specialized, from which a given element or attribute type is specialized. The specialization hierarchy for a DITA element is formally declared through its `@class` attribute.

structural type

A topic type or map type.

DITA modules

The following terminology is used to discuss DITA modules:

attribute domain module

A domain module that defines a specialization of either the `@base` or `@props` attribute.

constraint module

A set of declarations that imposes additional constraints onto the element or attribute types that are defined in a specific vocabulary module.

domain module

A vocabulary module that defines a set of element types or an attribute type that supports a specific subject or functional area.

element domain module

A domain module that defines one or more element types for use within maps or topics.

structural module

A vocabulary module that defines a top-level map type or topic type.

vocabulary module

A set of element or attribute declarations.

Linking and addressing terms

The following terminology is used to discuss linking and addressing terms:

referenced element

An element that is referenced by another DITA element. See also *referencing element*.

Example

Consider the following code sample from a `installation-reuse.dita` topic. The `<step>` element that it contains is a referenced element; other DITA topics reference the `<step>` element by using the `@conref` attribute.

```
<step id="run-startcmd-script">
  <cmd>Run the startcmd script that is applicable to your operating-system
  environment.</cmd>
</step>
```

referencing element

An element that references another DITA element by specifying an addressing attribute. See also *referenced element* and *addressing attribute*

Example

The following `<step>` element is a referencing element. It uses the `@conref` attribute to reference a `<step>` element in the `installation-reuse.dita` topic.

```
<step conref="installation-reuse.dita#reuse/run-startcmd-script">
  <cmd/>
</step>
```

addressing attribute

An attribute, such as `@conref`, `@conkeyref`, `@keyref`, and `@href`, that specifies an address.

Terminology related to keys

The following terminology is used to discuss keys:

resource

For the purposes of keys and key resolution, one of the following:

- An object addressed by URI
- Metadata specified on a resource, such as a `@scope` or `@format` attribute
- Text or metadata located within a `<topicmeta>` element

key

A name for a resource. See [Using keys for addressing](#) on page 58 for more information.

key definition

A `<topicref>` element that binds one or more key names to zero or more resources.

key reference

An attribute that references a key, such as `@keyref` or `@conkeyref`.

key space

A list of key definitions that are used to resolve key references.

effective key definition

The definition for a key within a key space that is used to resolve references to that key. A key might have multiple definitions within a key space, but only one of those definitions is effective.

key scope

A map or section of a map that defines its own key space and serves as the resolution context for its key references.

Map terms

root map

The DITA map that is provided as input for a processor.

submap

A DITA map that is referenced with a `@scope` attribute that evaluates as "local". The value of the scope attribute might be explicitly set, be defaulted, or cascade from another element.

peer map

A DITA map that is referenced with a `@scope` attribute that evaluates as "peer". The value of the scope attribute might be explicitly set, be defaulted, or cascade from another element.

map branch

A `<topicref>` element or a specialization of `<topicref>`, along with any child elements and all resources that are referenced by the original element or its children.

Basic concepts

DITA has been designed to satisfy requirements for information typing, semantic markup, modularity, reuse, interchange, and production of different deliverable forms from a single source. These topics provide an overview of the key DITA features and facilities that serve to satisfy these requirements.

DITA topics

In DITA, a topic is the basic unit of authoring and reuse. All DITA topics have the same basic structure: a title and, optionally, a body of content. Topics can be generic or more specialized; specialized topics represent more specific information types or semantic roles, for example, `<concept>`, `<task>`, or `<reference>`. Topics can be generic or more specialized; specialized topics represent more specific information types or semantic roles, for example, `<concept>`, `<task>`, `<reference>`, or `<learningContent>`. See [DITA topics](#) for more information.

DITA maps

DITA maps are documents that organize topics and other resources into structured collections of information. DITA maps specify hierarchy and the relationships among the topics; they also provide the contexts in which keys are defined and resolved. DITA maps *SHOULD* have `.ditamap` as the file extension. See [DITA maps](#) for more information.

Information typing

Information typing is the practice of identifying types of topics, such as concept, reference, and task, to clearly distinguish between different types of information. Topics that answer different reader questions (How ...? What is ...?) can be categorized with different information types. The base information types provided by DITA specializations (for example, technical content, machine industry, and learning and

training) provide starter sets of information types that can be adopted immediately by many technical and business-related organizations. See [Information typing](#) for more information.

DITA addressing

DITA provides two addressing mechanisms. DITA addresses either are direct URI-based addresses, or they are indirect key-based addresses. Within DITA documents, individual elements are addressed by unique identifiers specified on the `@id` attribute. DITA defines two fragment-identifier syntaxes; one is the full fragment-identifier syntax, and the other is an abbreviated fragment-identifier syntax that can be used when addressing non-topic elements from within the same topic. See [DITA addressing](#) for more information.

Content reuse

The DITA `@conref`, `@conkeyref`, `@conrefend`, and `@conaction` attributes provide mechanisms for reusing content within DITA topics or maps. These mechanisms can be used both to pull and push content. See [Content reuse](#) for more information.

Conditional processing

Conditional processing, also known as profiling, is the filtering or flagging of information based on processing-time criteria. See [Conditional processing](#) for more information.

Configuration

A document type shell is an XML grammar file that specifies the elements and attributes that are allowed in a DITA document. The document type shell integrates structural modules, domain modules, and constraint modules. In addition, a document type shell specifies whether and how topics can nest. See [Configuration](#) on page 116 for more information.

Specialization

The specialization feature of DITA allows for the creation of new element types and attributes that are explicitly and formally derived from existing types. This facilitates interchange of conforming DITA content and ensures a minimum level of common processing for all DITA content. It also allows specialization-aware processors to add specialization-specific processing to existing base processing. See [Specialization](#) for more information.

Constraints

Constraint modules define additional constraints for vocabulary modules in order to restrict content models or attribute lists for specific element types, remove certain extension elements from an integrated domain module, or replace base element types with domain-provided, extension element types. See [Constraints](#) for more information.

File extensions

DITA uses certain file extensions for topics, maps, and conditional processing profiles.

Files that contain DITA content *SHOULD* use the following file extensions:

DITA topics

- *.dita (preferred)
- *.xml

DITA maps

*.ditamap

Conditional processing profiles

*.ditaval

Producing different deliverables from a single source

DITA is designed to enable the production of multiple deliverable formats from a single set of DITA content. This means that many rendition details are specified neither in the DITA specification nor in the DITA content; the rendition details are defined and controlled by the processors.

Like many XML-based applications for human-readable documentation, DITA supports the separation of content from presentation. This is necessary when content is used in different contexts, since authors cannot predict how or where the material that they author will be used. The following features and mechanisms enable users to produce different deliverable formats from a single source:

DITA maps

Different DITA maps can be optimized for different delivery formats. For example, you might have a book map for printed output and another DITA map to generate online help; each map uses the same content set.

Specialization

The DITA specialization facility enables users to create XML elements that can provide appropriate rendition distinctions. Because the use of specializations does not impede interchange or interoperability, DITA users can safely create the specializations that are demanded by their local delivery and rendition requirements, with a minimum of additional impact on the systems and business processes that depend on or use the content. While general XML practices suggest that element types should be semantic, specialization can be used to define element types that are purely presentational in nature. The highlighting domain is an example of such a specialization.

Conditional processing

Conditional processing makes it possible to have a DITA topic or map that contains delivery-specific content.

Content referencing

The conref mechanism makes it possible to construct delivery-specific maps or topics from a combination of generic components and delivery-context-specific components.

Key referencing

The keyref mechanism makes it possible to have key words be displayed differently in different deliverables. It also allows a single link to resolve to different targets in different deliverables.

@outputclass attribute

The @outputclass attribute provides a mechanism whereby authors can indicate specific rendition intent where necessary. Note that the DITA specification does not define any values for the @outputclass attribute; the use of the @outputclass attribute is processor specific.

While DITA is independent of any particular delivery format, it is a standard that supports the creation of human-readable content. As such, it defines some fundamental document components including paragraphs, lists, and tables. When there is a reasonable expectation that such basic document components be rendered consistently, the DITA specification defines default or suggested renderings.

DITA markup

Topics and maps are the basic building blocks of the Darwin Information Typing Architecture (DITA). Metadata attributes and values can be added to DITA topics and maps, as well as to elements within topics, to allow for conditional publishing and content reuse.

DITA topics and maps are XML documents that conform to the XML specification. As such, they can be viewed, edited, validated, and processed with standard XML tools, although some DITA-specific features, such as content reference, key reference, and specialization require DITA-specific processing for full implementation and validation.

DITA topics

DITA topics are the basic units of DITA content and the basic units of reuse. Each topic contains a single subject. Topics may be of specific specialized information types, such as task, concept, or reference, or may be generic, that is, without a specified information type.

The topic as the basic unit of information

In DITA, a topic is the basic unit of authoring and reuse. All DITA topics have the same basic structure: a title and, optionally, a body of content. Topics can be generic or more specialized; specialized topics represent more specific information types or semantic roles, for example, `<concept>`, `<task>`, or `<reference>`. Topics can be generic or more specialized; specialized topics represent more specific information types or semantic roles, for example, `<concept>`, `<task>`, `<reference>`, or `<learningContent>`.

DITA topics consist of content units that can be as generic as sets of paragraphs and unordered lists or as specific as sets of instructional steps in a procedure or cautions to be considered before a procedure is performed. Content units in DITA are expressed using XML elements and can be conditionally processed using metadata attributes.

Classically, a DITA topic is a titled unit of information that can be understood in isolation and used in multiple contexts. It should be short enough to address a single subject or answer a single question but long enough to make sense on its own and be authored as a self-contained unit. However, DITA topics also can be less self-contained units of information, such as topics that contain only titles and short descriptions and serve primarily to organize subtopics or links or topics that are designed to be nested for the purposes of information management, authoring convenience, or interchange.

DITA topics are used by reference from DITA maps. DITA maps enable topics to be organized in a hierarchy for publication. Large units of content, such as complex reference documents or book chapters, are created by nesting topic references in a DITA map. The same set of DITA topics can be used in any number of maps.

DITA topics also can be used and published individually; for example, one can represent an entire deliverable as a single DITA document that consists of a root topic and nested topics. This strategy can accommodate the migration of legacy content that is not topic-oriented; it also can accommodate information that is not meaningful outside the context of a parent topic. However, the power of DITA is most fully realized by storing each DITA topic in a separate XML document and using DITA maps to organize how topics are combined for delivery. This enables a clear separation between how topics are authored and stored and how topics are organized for delivery.

The benefits of a topic-based architecture

Topics enable the development of usable and reusable content.

While DITA does not require the use of any particular writing practice, the DITA architecture is designed to support authoring, managing, and processing of content that is designed to be reused. Although DITA provides

significant value even when reuse is not a primary requirement, the full value of DITA is realized when content is authored with reuse in mind. To develop topic-based information means creating units of standalone information that are meaningful with little or no surrounding context.

By organizing content into topics that are written to be reusable, authors can achieve several goals:

- Content is readable when accessed from an index or search, not just when read in sequence as part of an extended narrative. Since most readers do not read technical and business-related information from beginning to end, topic-oriented information design ensures that each unit of information can be read independently.
- Content can be organized differently for online and print delivery. Authors can create task flows and concept hierarchies for online delivery and create a print-oriented hierarchy to support a narrative content flow.
- Content can be reused in different collections. Since a topic is written to support random access (as by search), it should also be understandable when included as part of various product deliverables. Topics permit authors to refactor information as needed, including only the topics that apply to each unique scenario.
- Content is more manageable in topic form whether managed as individual files in a traditional file system or as objects in a content management system.
- Content authored in topics can be translated and updated more efficiently and less expensively than information authored in larger or more sequential units.
- Content authored in topics can be filtered more efficiently, encouraging the assembly and deployment of information subsets from shared information repositories.

Topics written for reuse should be small enough to provide opportunities for reuse but large enough to be coherently authored and read. When each topic is written to address a single subject, authors can organize a set of topics logically and achieve an acceptable narrative content flow.

Disciplined, topic-oriented writing

Topic-oriented writing is a disciplined approach to writing that emphasizes modularity and reuse of concise units of information: topics. Well-designed DITA topics can be reused in many contexts, as long as writers are careful to avoid unnecessary transitional text.

Conciseness and appropriateness

Readers who are trying to learn or do something quickly appreciate information that is written in a structure that is easy to follow and contains only the information needed to complete that task or grasp a fact. Recipes, encyclopedia entries, car repair procedures--all serve up a uniquely focused unit of information. The topic contains everything required by the reader.

Locational independence

A well-designed topic is reusable in other contexts to the extent that it is context free, meaning that it can be inserted into a new document without revision of its content. A context-free topic avoids transitional text. Phrases like "As we considered earlier ..." or "Now that you have completed the initial step ..." make little sense if a topic is reused in a new context in which the relationships are different or no longer exist. A well-designed topic reads appropriately in any new context because the text does not refer the reader outside the topic.

Navigational independence

Most print publications or web pages are a mixture of content and navigation. Internal links lead a reader through a sequence of choices as he or she navigates through a website. DITA supports the separation of navigation from content by assembling independent topics into DITA maps. Nonetheless, writers may want to provide links within a topic to additional topics or external resources. DITA does not prohibit such linking within individual topics. The DITA relationship table enables links between topics and to external content. Since it is defined in the DITA map, it is managed independently of the topic content.

Links in the content are best used for cross-references within a topic. Links from within a topic to additional topics or external resources should be avoided because they limit the reusability of the topic. To link from a term or keyword to its definition, use the DITA keyref facility to avoid creating topic-to-topic dependencies that are difficult to maintain. See [Key-based addressing](#).

Information typing

Information typing is the practice of identifying types of topics, such as concept, reference, and task, to clearly distinguish between different types of information. Topics that answer different reader questions (How ...? What is ...?) can be categorized with different information types. The base information types provided by DITA specializations (for example, technical content, machine industry, and learning and training) provide starter sets of information types that can be adopted immediately by many technical and business-related organizations.



Note: The specializations of `<topic>`, such `<concept>`, `<task>`, `<reference>`, and `<learningContent>` are not included in the DITA 1.3 base edition.

Information typing has a long history of use in the technical documentation field to improve information quality. It is based on extensive research and experience, including Robert Horn's Information Mapping and Hughes Aircraft's STOP (Sequential Thematic Organization of Proposals) technique. Note that many DITA topic types are not necessarily closely connected with traditional Information Mapping.

Information typing is a practice designed to keep documentation focused and modular, thus making it clearer to readers, easier to search and navigate, and more suitable for reuse. Classifying information by type helps authors perform the following tasks:

- Develop new information more consistently
- Ensure that the correct structure is used for closely related kinds of information (retrieval-oriented structures like tables for reference information and simple sequences of steps for task information)
- Avoid mixing content types, thereby losing reader focus
- Separate supporting concept and reference information from tasks, so that users can read the supporting information if needed and ignore if it is not needed
- Eliminate unimportant or redundant detail
- Identify common and reusable subject matter

DITA currently defines a small set of well-established information types that reflects common practices in certain business domains, for example, technical communication and instruction and assessment. However, the set of possible information types is unbounded. Through the mechanism of specialization, new information types can be defined as specializations of the base topic type (`<topic>`) or as refinements of existing topics types, for example, `<concept>`, `<task>`, `<reference>`, or `<learningContent>`.

You need not use any of the currently-defined information types. However, where a currently-defined information type matches the information type of your content, the currently-defined information type should be used, either directly, or as a base for specialization. For example, information that is procedural in nature should use the task information type or a specialization of task. Consistent use of established information types helps ensure smooth interchange and interoperability of DITA content.

Generic topics

The element type `<topic>` is the base topic type from which all other topic types are specialized. All topics have the same basic structure.

For authors, typed content is preferred to support consistency in writing and presentation to readers. The generic topic type should only be used if authors are not trained in information typing or when a specialized topic type is inappropriate. The OASIS DITA standard provides several specialized topic types, including concept, task, and reference that are critical for technical content development.

For those pursuing specialization, new specialized topic types should be specialized from appropriate ancestors to meet authoring and output requirements.

Topic structure

All topics have the same basic structure, regardless of topic type: title, description or abstract, prolog, body, related links, and nested topics.

All DITA topics must have an XML identifier (the `@id` attribute) and a title. The basic topic structure consists of the following parts, some of which are optional:

Topic element

The topic element holds the required `@id` attribute and contains all other elements.

Title

The title contains the subject of the topic.

Alternate titles

Titles specifically for use in navigation or search. When not provided, the base title is used for all contexts.

Short description or abstract

A short description of the topic or a longer abstract with an embedded short description. The short description may be used both in topic content (as the first paragraph), in generated summaries that include the topic, and in links to the topic. Alternatively, the abstract lets you create more complex introductory content and uses an embedded short description element to define the part of the abstract that is suitable for summaries and link previews.

While short descriptions aren't required, they can make a dramatic difference to the usability of an information set and should generally be provided for all topics.

Prolog

The prolog is the container for topic metadata, such as change history, audience, product, and so on.

Body

The topic body contains the topic content: paragraphs, lists, sections, and other content that the information type permits.

Related links

Related links connect to other topics. When an author creates a link as part of a topic, the topic becomes dependent on the other topic being available. To reduce dependencies between topics and thereby increase the reusability of each topic, authors may use DITA maps to define and manage links between topics, instead of embedding links directly in each related topic.

Nested topics

Topics can be defined inside other topics. However, nesting requires special care because it can result in complex documents that are less usable and less reusable. Nesting may be appropriate for information that is first converted from desktop publishing or word processing files or for topics that are unusable independent from their parent or sibling topics.

The rules for topic nesting can be configured in a document-type shells. For example, the standard DITA configuration for concept topics only allows nested concept topics. However, local configuration of the concept topic type could allow other topic types to nest or disallow topic nesting entirely. In addition, the `@chunk` attribute enables topics to be equally re-usable regardless of whether they are separate or nested. The standard DITA configuration for ditabase document-type documents allows unrestricted topic nesting

and may be used for holding sets of otherwise unrelated topics that hold re-usable content. It may also be used to convert DITA topics from non-DITA legacy source without first determining how individual topics should be organized into separate XML documents.

Topic content

The content of all topics, regardless of topic type, is built on the same common structures.

Topic body

The topic body contains all content except for that contained in the title or the short description/abstract. The topic body can be constrained to remove specific elements from the content model; it also can be specialized to add additional specialized elements to the content model. The topic body can be generic while the topic title and prolog are specialized.

Sections and examples

The body of a topic might contain divisions, such as sections and examples. They might contain block-level elements like titles and paragraphs and phrase-level elements like API names or text. It is recommended that sections have titles, whether they are entered directly into the `<title>` element or rendered using a fixed or default title.

Either body divisions or untitled sections or examples may be used to delimit arbitrary structures within a topic body. However, body divisions may nest, but sections and examples cannot contain sections.

`<sectiondiv>`

The `<sectiondiv>` element enables the arbitrary grouping of content within a section for the purpose of content reuse. The `<sectiondiv>` element does not include a title. Content that requires a title should use `<section>` or `<example>`.

`<bodydiv>`

The `<bodydiv>` element enables the arbitrary grouping of content within the body of a topic for the purpose of content reuse. The `<bodydiv>` element does not include a title. Content that requires a title should use `<section>` or `<example>`.

`<div>`

The `<div>` element enables the arbitrary grouping of content within a topic. The `<div>` element does not include a title. Content that requires a title should use `<section>` or `<example>` or, possibly, `<fig>`.

Block-level elements

Paragraphs, lists, figures, and tables are types of "block" elements. As a class of content, they can contain other blocks, phrases, or text, though the rules vary for each structure.

Phrases and keywords

Phrase level elements can contain markup to label parts of a paragraph or parts of a sentence as having special semantic meaning or presentation characteristics, such as `<uicontrol>` or ``. Phrases can usually contain other phrases and keywords as well as text. Keywords can only contain text.

Images

Images can be inserted to display photographs, illustrations, screen captures, diagrams, and more. At the phrase level, they can display trademark characters, icons, toolbar buttons, and so forth.

Multimedia

The `<object>` element enables authors to include multimedia, such as diagrams that can be rotated and expanded. The `<foreign>` element enables authors to include media within topic content, for example, SVG graphics, MathML equations, and so on.

DITA maps

This topic collection contains information about DITA maps and the purposes that they serve. It also includes high-level information about DITA map elements, attributes, and metadata.

Definition of DITA maps

DITA maps are documents that organize topics and other resources into structured collections of information. DITA maps specify hierarchy and the relationships among the topics; they also provide the contexts in which keys are defined and resolved. DITA maps *SHOULD* have `.ditamap` as the file extension.

Maps draw on a rich set of existing best practices and standards for defining information models, such as hierarchical task analysis. They also support the definition of non-hierarchical relationships, such as matrices and groups, which provide a set of capabilities that has similarities to Resource Description Framework (RDF) and ISO topic maps.

DITA maps use `<topicref>` elements to reference DITA topics, DITA maps, and non-DITA resources, for example, HTML and TXT files. The `<topicref>` elements can be nested or grouped to create relationships among the referenced topics, maps, and non-DITA files; the `<topicref>` elements can be organized into hierarchies in order to represent a specific order of navigation or presentation.

DITA maps impose an architecture on a set of topics. Information architects can use DITA maps to specify what DITA topics are needed to support a given set of user goals and requirements; the sequential order of the topics; and the relationships that exist among those topics. Because DITA maps provide this context for topics, the topics themselves can be relatively context-free; they can be used and reused in multiple different contexts.

DITA maps often represent a single deliverable, for example, a specific Web site, a printed publication, or the online help for a product. DITA maps also can be subcomponents for a single deliverable, for example, a DITA map might contain the content for a chapter in a printed publication or the troubleshooting information for an online help system. The DITA specification provides specialized map types; book maps represent printed publications, subject scheme maps represent taxonomic or ontological classifications, and learning maps represent formal units of instruction and assessment. However, these map types are only a starter set of map types reflecting well-defined requirements.

DITA maps establish relationships through the nesting of `<topicref>` elements and the application of the `@collection-type` attribute. Relationship tables may also be used to associate topics with each other based on membership in the same row; for example, task topics can be associated with supporting concept and reference topics by placing each group in cells of the same row. During processing, these relationships can be rendered in different ways, although they typically result in lists of "Related topics" or "For more information" links. Like many aspects of DITA, the details about how such linking relationships are presented is determined by the DITA processor.

DITA maps also define keys and organize the contexts (*key scopes*) in which key references are resolved.

Purpose of DITA maps

DITA maps enable the scalable reuse of content across multiple contexts. They can be used by information architects, writers, and publishers to plan, develop, and deliver content.

DITA maps support the following uses:

Defining an information architecture

Maps can be used to define the topics that are required for a particular audience, even before the topics themselves exist. DITA maps can aggregate multiple topics for a single deliverable.

Defining what topics to build for a particular output

Maps reference topics that are included in output processing. Information architects, authors, and publishers can use maps to specify a set of topics that are processed at the same time, instead of processing each topic individually. In this way, a DITA map can serve as a manifest or bill of materials.

Defining navigation

Maps can define the online navigation or table of contents for a deliverable.

Defining related links

Maps define relationships among the topics they reference. These relationships are defined by the nesting of elements in the DITA map, relationship tables, and the use of elements on which the `@collection-type` attribute is set. On output, these relationships might be expressed as related links or the hierarchy of a table of contents (TOC).

Defining an authoring context

The DITA map can define the authoring framework, providing a starting point for authoring new topics and integrating existing ones.

Defining keys and key scopes

Maps can define keys, which provide an indirect addressing mechanism that enhances portability of content. The keys are defined by `<topicref>` elements or specializations of `<topicref>` elements, such as `<keydef>`. The `<keydef>` element is a convenience element; it is a specialized type of a `<topicref>` element with the following attributes:

- A required `@keys` attribute
- A `@processing-role` attribute with a default value of "resource-only".

Maps also define the context or contexts for resolving key-based references, such as elements that specify the `@keyref` or `@conkeyref` attribute. Elements within a map structure that specify a `@keyscope` attribute create a new context for key reference resolution. Key references within such elements are resolved against the set of effective key definitions for that scope.

Specialized maps can provide additional semantics beyond those of organization, linking, and indirection. For example, the `subjectScheme` map specialization adds the semantics of taxonomy and ontology definition.

DITA map elements

A DITA map describes the relationships among a set of DITA topics. The DITA map and `map-group` domain elements organize topics into hierarchies, groups, and relationships; they also define keys.

A DITA map is composed of the following elements:

`<map>`

The `<map>` element is the root element of the DITA map.

`<topicref>`

The `<topicref>` elements are the basic elements of a map. A `<topicref>` element can reference a DITA topic, a DITA map, or a non-DITA resource. A `<topicref>` element also can have a title, short description, and the same kind of prolog-level metadata that is available in topics.

The `<topicref>` elements can be nested to create a hierarchy, which can be used to define a table of contents (TOC) for print output, online navigation, and parent/child links. Hierarchies can be annotated using the `@collection-type` attribute to define a particular type of relationship, such as a set of choices, a sequence, or a family. These collection types can affect link generation, and they may be interpreted differently for different outputs.

`<reltable>`

Relationship tables are defined with the `<reltable>` element. Relationship tables can be used to define relationships among DITA topics or among DITA topics and non-DITA resources. In a relationship table, the columns define common attributes, metadata, or information types (for example, task or troubleshooting) for the resources that are referenced in that column. The rows define relationships between the resources in different cells of the same row.

The `<relrow>`, `<relcell>`, `<relheader>`, and `<relcolspec>` elements are used to define the components of the relationship table. Relationships defined in the relationship table also can be further refined by using the `@collection-type` attribute.

`<topicgroup>`

The `<topicgroup>` element defines a group or collection outside of a hierarchy or relationship table. It is a convenience element that is equivalent to a `<topicref>` element without an `@href` attribute or navigation title. Groups can be combined with hierarchies and relationship tables, for example, by including a `<topicgroup>` element within a set of siblings in a hierarchy or within a table cell. The `<topicref>` elements so grouped can then share inherited attributes and linking relationships with no effect on the navigation or table of contents.

`<topicmeta>`

Most map-level elements, including the map itself, can contain metadata inside the `<topicmeta>` element. Metadata typically is applied to an element and its descendants.

`<ux-window>`

The `<ux-window>` element enables authors to define windowing information for the display of output topics that are appropriate to the delivery platform. Window management is important in user assistance and help system outputs, as well as for other hypertext and electronic delivery modes.

`<topichead>`

The `<topichead>` element provides a navigation title; it is a convenience element that is equivalent to a `<topicref>` element with a navigation title but no associated resource.

`<anchor>`

The `<anchor>` element provides an integration point that another map can reference in order to insert its navigation into the referenced map's navigation tree. For those familiar with Eclipse help systems, this serves the same purpose as the `<anchor>` element in that system. It might not be supported for all output formats.

`<navref>`

The `<navref>` element represents a pointer to another map which should be preserved as a transcluding link in the result deliverable rather than resolved when the deliverable is produced. Output formats that support such linking can integrate the referenced resource when displaying the referencing map to an end user.

<keydef>

Enables authors to define keys. This element is a convenience element; it is a specialization of `<topicref>` that sets the default value of the `@processing-role` attribute to "resource-only". Setting the `@processing-role` attribute to resource-only ensures that the resource referenced by the key definition is not directly included in the navigation that is defined by the map.

<mapref>

Enables authors to reference an entire DITA map, including hierarchy and relationship tables. This element is a convenience element; it is a specialization of `<topicref>` that sets the default value of the `@format` attribute to "ditamap". The `<mapref>` element represents a reference from a parent map to a subordinate map.

<topicset>

Enables authors to define a branch of navigation in a DITA map so that it can be referenced from another DITA map.

<topicsetref>

Enables authors to reference a navigation branch that is defined in another DITA map.

<anchorref>

Enables authors to define a map fragment that is pushed to the location defined by an anchor.

DITA map attributes

DITA maps have unique attributes that are designed to control the way that relationships are interpreted for different output purposes. In addition, DITA maps share many metadata and linking attributes with DITA topics.

DITA maps often encode structures that are specific to a particular medium or output, for example, Web pages or a PDF document. Attributes, such as `@deliveryTarget` and `@toc`, are designed to help processors interpret the DITA map for each kind of output. Many of these attributes are not available in DITA topics; individual topics, once separated from the high-level structures and dependencies associated with a particular kind of output, should be entirely reusable regardless of the intended output format.

@collection-type

The `@collection-type` attribute specifies how the children of a `<topicref>` element relate to their parent and to each other. This attribute, which is set on the parent element, typically is used by processors to determine how to generate navigation links in the rendered topics. For example, a `@collection-type` value of "sequence" indicates that children of the specifying `<topicref>` element represent an ordered sequence of topics; processors might add numbers to the list of child topics or generate next/previous links for online presentation. This attribute is available in topics on the `<linklist>` and `<linkpool>` elements, where it has the same behavior. Where the `@collection-type` attribute is available on elements that cannot directly contain elements (such as `<reltable>` or `<topicref>`), the behavior of the attribute is reserved for future use.

@linking

By default, the relationships between the topics that are referenced in a map are reciprocal:

- Child topics link to parent topics and vice versa.
- Next and previous topics in a sequence link to each other.
- Topics in a family link to their sibling topics.
- Topics referenced in the table cells of the same row in a relationship table link to each other. A topic referenced within a table cell does not (by default) link to other topics referenced in the same table cell.

This behavior can be modified by using the `@linking` attribute, which enables an author or information architect to specify how a topic should participate in a relationship. The following values are valid:

linking="none"

Specifies that the topic does not exist in the map for the purposes of calculating links.

linking="sourceonly"

Specifies that the topic will link to its related topics but not vice versa.

linking="targetonly"

Specifies that the related topics will link to it but not vice versa.

linking="normal"

Default value. It specifies that linking will be reciprocal (the topic will link to related topics, and they will link back to it).

Authors also can create links directly in a topic by using the `<xref>` or `<link>` elements, but in most cases map-based linking is preferable, because links in topics create dependencies between topics that can hinder reuse.

Note that while the relationships between the topics that are referenced in a map are reciprocal, the relationships merely *imply* reciprocal links in generated output that includes links. The rendered navigation links are a function of the presentation style that is determined by the processor.

@toc

Specifies whether topics are excluded from navigation output, such as a Web site map or an online table of contents. By default, `<topicref>` hierarchies are included in navigation output; relationship tables are excluded.

@navtitle

Specifies a navigation title. This is a shorter version of the title that is used in the navigation only. By default, the `@navtitle` attribute is ignored; it serves only to help the DITA map author keep track of the title of the topic.



Note: The `@navtitle` attribute is deprecated in favor of the `<navtitle>` element. When both a `<navtitle>` element and a `@navtitle` attribute are specified, the `<navtitle>` element should be used.

@locktitle

If `@locktitle` is set to "yes", the `<navtitle>` element or `@navtitle` attribute is used if it is present. Otherwise, the `<navtitle>` element or `@navtitle` attribute is ignored and the navigation title is retrieved from the referenced file.



Note: The `@navtitle` attribute is deprecated in favor of the `<navtitle>` element. When both a `<navtitle>` element and a `@navtitle` attribute are specified, the `<navtitle>` element should be used.

@print

Specifies whether the topic should be included in printed output.



Note: Beginning with DITA 1.3, the `@print` attribute is deprecated. It is replaced with a conditional processing attribute: `@deliveryTarget`. See [@deliveryTarget](#) for more details.

@search

Specifies whether the topic should be included in search indexes.

@chunk

Specifies that the processor generates an interim set of DITA topics that are used as the input for the final processing. This can produce the following output results:

- Multi-topic files are transformed into smaller files, for example, individual HTML files for each DITA topic.
- Individual DITA topics are combined into a single file.

Specifying a value for the @chunk attribute on a <map> element establishes chunking behavior that applies to the entire map, unless overridden by @chunk attributes that are set on more specific elements in the DITA map. For a detailed description of the @chunk attribute and its usage, see [Chunking](#) on page 105.

@copy-to

In most situations, specifies whether a duplicate version of the topic is created when it is transformed. This duplicate version can be either literal or virtual. The value of the @copy-to attribute specifies the uniform resource identifier (URI) by which the topic can be referenced by a @conref attribute, <topicref> element, or <xref> element. The duplication is a convenience for output processors that use the URI of the topic to generate the base address of the output. The @keys and @keyref attributes provide an alternative mechanism; they enable references to topics in specific-use contexts.

The @copy-to attribute also can be used to specify the name of a new chunk when topics are being chunked; it also can be used to determine the name of the stub topic that is generated from a <topicref> element that contains a title but does not specify a target. In both of those cases, no duplicate version of the topic is generated.

For information on how the @copy-to attribute can be used with the @chunk attribute, see [Chunking](#) on page 105.

@processing-role

Specifies whether the topic or map referenced should be processed normally or treated as a resource that is only included in order to resolve key or content references.

processing-role="normal"

The topic is a readable part of the information set. It is included in navigation and search results. This is the default value for the <topicref> element.

processing-role="resource-only"

The topic should be used only as a resource for processing. It is not included in navigation or search results, nor is it rendered as a topic. This is the default value for the <keydef> element.

If the @processing-role attribute is not specified locally, the value cascades from the closest element in the containment hierarchy.

@cascade

Specifies whether the default rules for the cascading of metadata attributes in a DITA map apply. In addition to the following specified values, processors also *MAY* define additional values.

cascade="merge"

The metadata attributes cascade; the values of the metadata attributes are additive. This is the processing default for the @cascade attribute and was the only defined behavior for DITA 1.2 and earlier.

cascade="nomerge"

The metadata attributes cascade; however, they are not additive for `<topicref>` elements that specify a different value for a specific metadata attribute. If the cascading value for an attribute is already merged based on multiple ancestor elements, that merged value continues to cascade until a new value is encountered (that is, setting `cascade="nomerge"` does not undo merging that took place on ancestors).

For more information, see [Example: How the cascade attribute functions](#) on page 26.

@keys

Specifies one or more key names.

@keyscope

Defines a new scope for key definition and resolution, and gives the scope one or more names. For more information about key scopes, see [Indirect key-based addressing](#) on page 55.

Attributes in the list above are used exclusively or primarily in maps, but many important map attributes are shared with elements in topics. DITA maps also use many of the following attributes that are used with linking elements in DITA topics, such as `<link>` and `<xref>`:

- @format
- @href
- @keyref
- @scope
- @type

The following metadata and reuse attributes are used by both DITA maps and DITA topics:

- @product, @platform, @audience, @otherprops, @rev, @status, @importance
- @dir, @xml:lang, @translate
- @id, @conref, @conrefend, @conkeyref, @conaction
- @props and any attribute specialized from @props (such as @deliveryTarget)
- @search

When new attributes are specialized from @props or @base as a domain, they can be incorporated into both map and topic structural types.

Examples of DITA maps

This section of the specification contains simple examples of DITA maps. The examples illustrate a few of the ways that DITA maps are used.

Example: DITA map that references a subordinate map

This example illustrates how one map can reference a subordinate map using either `<mapref>` or the basic `<topicref>` element.

The following code sample illustrates how a DITA map can use the specialized `<mapref>` element to reference another DITA map:

```
<map>
  <title>DITA work at OASIS</title>
  <topicref href="oasis-dita-technical-committees.dita">
    <topicref href="dita_technical_committee.dita"/>
    <topicref href="dita_adoption_technical_committee.dita"/>
  </topicref>
```



```
<mapref href="oasis-processes.ditamap"/>
<!-- ... -->
</map>
```

The `<mapref>` element is a specialized `<topicref>` intended to make it easier to reference another map; use of `<mapref>` is not required for this task. This map also could be tagged in the following way:

```
<map>
  <title>DITA work at OASIS</title>
  <topicref href="oasis-dita-technical-committees.dita">
    <topicref href="dita_technical_committee.dita"/>
    <topicref href="dita_adoption_technical_committee.dita"/>
  </topicref>
  <topicref href="oasis-processes.ditamap" format="ditamap"/>
  <!-- ... -->
</map>
```

With either of the above examples, during processing, the map is resolved in the following way:

```
<map>
  <title>DITA work at OASIS</title>
  <topicref href="oasis-dita-technical-committees.dita">
    <topicref href="dita_technical_committee.dita"/>
    <topicref href="dita_adoption_technical_committee.dita"/>
  </topicref>
  <!-- Contents of the oasis-processes.ditamap file -->
  <topicref href="oasis-processes.dita">
    <!-- ... -->
  </topicref>
  <!-- ... -->
</map>
```

Example: DITA map with a simple relationship table

This example illustrates how to interpret a basic three-column relationship table used to maintain links between concept, task, and reference material.

The following example contains the markup for a simple relationship table:

```
<map>
<!-- ... -->
<reltable>
  <relheader>
    <relcolspec type="concept"/>
    <relcolspec type="task"/>
    <relcolspec type="reference"/>
  </relheader>
  <relrow>
    <relcell>
      <topicref href="A.dita"/>
    </relcell>
    <relcell>
      <topicref href="B.dita"/>
    </relcell>
    <relcell>
      <topicref href="C1.dita"/>
      <topicref href="C2.dita"/>
    </relcell>
  </relrow>
</reltable>
</map>
```

A DITA-aware tool might represent the relationship table graphically:

type="concept"	type="task"	type="reference"
A	B	C1 C2

When the output is generated, the topics contain the following linkage:

A

Links to B, C1, and C2

B

Links to A, C1, and C2

C1, C2

Links to A and B

Example: How the @collection-type and @linking attributes determine links

In this scenario, a simple map establishes basic hierarchical and relationship table links. The @collection-type and @linking attributes are then added to modify how links are generated.

The following example illustrates how linkage is defined in a DITA map:

```
<topicref href="A.dita" collection-type="sequence">
  <topicref href="A1.dita"/>
  <topicref href="A2.dita"/>
</topicref>
<reltable>
  <relrow>
    <relcell><topicref href="A.dita"/></relcell>
    <relcell><topicref href="B.dita"/></relcell>
  </relrow>
</reltable>
```

Figure 1: Simple linking example

When the output is generated, the topics contain the following linkage. Sequential (next/previous) links between A1 and A2 are present because of the @collection-type attribute on the parent:

A

Links to A1, A2 as children

Links to B as related

A1

Links to A as a parent

Links to A2 as next in the sequence

A2

Links to A as a parent

Links to A1 as previous in the sequence

B

Links to A as related

The following example illustrates how setting the `@linking` attribute can change the default behavior:

```
<topicref href="A.dita" collection-type="sequence">
  <topicref href="B.dita" linking="none"/>
  <topicref href="A1.dita"/>
  <topicref href="A2.dita"/>
</topicref>
<reltable>
  <relrow>
    <relcell><topicref href="A.dita"/></relcell>
    <relcell linking="sourceonly"><topicref href="B.dita"/></relcell>
  </relrow>
</reltable>
```

Figure 2: Linking example with the `@linking` attribute

When the output is generated, the topics contain the following linkage:

A

- Links to A1, A2 as children
- Does not link to B as a child or related topic

A1

- Links to A as a parent
- Links to A2 as next in the sequence
- Does not link to B as previous in the sequence

A2

- Links to A as a parent
- Links to A1 as previous in the sequence

B

- Links to A as a related topic

Example: How the `@cascade` attribute functions

The following example illustrates how the `@cascade` attribute can be used to fine tune how the values for the `@platform` attribute apply to topics referenced in a DITA map.

Here a DITA map contains a collection of topics that apply to Windows, Linux, and Macintosh OS; it also contains a topic that is only applicable to users running the application on Linux.

```
<map product="PuffinTracker" platform="win linux mac" cascade="nomerge">
  <title>Puffin Tracking Software</title>
  <topicref href="intro.dita" navtitle="Introduction"/>
  <topicref href="setup.dita" navtitle="Setting up the product"/>
  <topicref href="linux-instructions.dita" navtitle="Linux instructions" platform="linux"/>
</map>
```

The values of the `@platform` attribute set at the map level cascade throughout the map and apply to the "Introduction" and "Setting up the product" topics. However, since the value of the `@cascade` attribute is set to "nomerge", the value of the `@platform` attribute for the "Linux instructions" topic does not merge with the values that cascade from above in the DITA map. The effective value of the `@platform` attribute for `linux-instructions.dita` is "linux".

The same results are produced by the following mark-up:

```
<map product="PuffinTracker" platform="win linux mac">
  <title>Puffin Tracking Software</title>
  <topicref href="intro.dita" navtitle="Introduction"/>
  <topicref href="setup.dita" navtitle="Setting up the product"/>
  <topicref href="linux-instructions.dita" navtitle="Linux instructions" platform="linux"
  cascade="nomerge"/>
</map>
```

Subject scheme maps and their usage

Subject scheme maps can be used to define controlled values and subject definitions. The controlled values can be bound to attributes, as well as element and attribute pairs. The subject definitions can contain metadata and provide links to more detailed information; they can be used to classify content and provide semantics that can be used in taxonomies and ontologies.

A DITA map can reference a subject scheme map by using a `<mapref>` element. Processors also *MAY* provide parameters by which subject scheme maps are referenced.

Subject scheme maps

Subject scheme maps use key definitions to define collections of controlled values and subject definitions.

Controlled values are keywords that can be used as values for attributes. For example, the `@audience` attribute can take a value that identifies the users that are associated with a particular product. Typical values for a medical-equipment product line might include "therapist", "oncologist", "physicist", and "radiologist". In a subject scheme map, an information architect can define a list of these values for the `@audience` attribute. Controlled values can be used to classify content for filtering and flagging at build time.

Subject definitions are classifications and sub-classifications that compose a tree. Subject definitions provide semantics that can be used in conjunction with taxonomies and ontologies. In conjunction with the classification domain, subject definitions can be used for retrieval and traversal of the content at run time when used with information viewing applications that provide such functionality.

Key references to controlled values are resolved to a key definition using the same precedence rules as apply to any other key. However, once a key is resolved to a controlled value, that key reference does not typically result in links or generated text.

Defining controlled values for attributes

Subject scheme maps can define controlled values for DITA attributes without having to define specializations or constraints. The list of available values can be modified quickly to adapt to new situations.

Each controlled value is defined using a `<subjectdef>` element, which is a specialization of the `<topicref>` element. The `<subjectdef>` element is used to define both a subject category and a list of controlled values. The parent `<subjectdef>` element defines the category, and the children `<subjectdef>` elements define the controlled values.

The subject definitions can include additional information within a `<topicmeta>` element to clarify the meaning of a value:

- The `<navtitle>` element can provide a more readable value name.
- The `<shortdesc>` element can provide a definition.

In addition, the `<subjectdef>` element can reference a more detailed definition of the subject, for example, another DITA topic or an external resource..

The following behavior is expected of processors:

- Authoring tools *SHOULD* use these lists of controlled values to provide lists from which authors can select values when they specify attribute values.
- Authoring tools *MAY* give an organization a list of readable labels, a hierarchy of values to simplify selection, and a shared definition of the value.
- An editor *MAY* support accessing and displaying the content of the subject definition resource in order to provide users with a detailed explanation of the subject.
- Tools *MAY* produce a help file, PDF, or other readable catalog to help authors better understand the controlled values.

Example: Controlled values that provide additional information about the subject

The following code fragment illustrates how a subject definition can provide a richer level of information about a controlled value:

```
<subjectdef keys="terminology" href="https://www.oasis-open.org/policies-guidelines/
keyword-guidelines">
  <subjectdef keys="rfc2119" href="rfc-2119.dita">
    <topicmeta>
      <navtitle>RFC-2119 terminology</navtitle>
      <shortdesc>The normative terminology that the DITA TC uses for the DITA
specification</shortdesc>
    </topicmeta>
  </subjectdef>
  <subjectdef keys="iso" href="iso-terminology.dita">
    <topicmeta>
      <navtitle>ISO keywords</navtitle>
      <shortdesc>The normative terminology used by some other OASIS technical
committees</shortdesc>
    </topicmeta>
  </subjectdef>
</subjectdef>
```

The content of the `<navtitle>` and `<shortdesc>` elements provide additional information that a processor might display to users as they select attribute values or classify content. The resources referenced by the `@href` attributes provide even more detailed information; a processor might render clickable links as part of a user interface that implements a progressive disclosure strategy

Binding controlled values to an attribute

The controlled values defined in a subject scheme map can be bound to an attribute or an element and attribute pair. This affects the expected behavior for processors and authoring tools.

The `<enumerationdef>` element binds the set of controlled values to an attribute. Valid attribute values are those that are defined in the set of controlled values; invalid attribute values are those that are not defined in the set of controlled values. An enumeration can specify an empty `<subjectdef>` element. In that case, no value is valid for the attribute. An enumeration also can specify an optional default value by using the `<defaultSubject>` element.

If an enumeration is bound, processors *SHOULD* validate attribute values against the controlled values that are defined in the subject scheme map. For authoring tools, this validation prevents users from entering misspelled or undefined values. Recovery from validation errors is implementation specific.

The default attribute values that are specified in a subject scheme map apply only if a value is not otherwise specified in the DITA source or as a default value by the XML grammar.

To determine the effective value for a DITA attribute, processors check for the following in the order outlined:

1. An explicit value in the element instance
2. A default value in the XML grammar
3. Cascaded value within the document
4. Cascaded value from a higher level document to the document
5. A default controlled value, as specified in the `<defaultSubject>` element
6. A value set by processing rules

Example: Binding a list of controlled values to the @audience attribute

The following example illustrates the use of the `<subjectdef>` element to define controlled values for types of users. It also binds the controlled values to the `@audience` attribute:

```
<subjectScheme>
  <!-- Define types of users -->
  <subjectdef keys="users">
    <subjectdef keys="therapist"/>
    <subjectdef keys="oncologist"/>
    <subjectdef keys="physicist"/>
    <subjectdef keys="radiologist"/>
  </subjectdef>

  <!-- Bind the "users" subject to the @audience attribute.
       This restricts the @audience attribute to the following
       values: therapist, oncologist, physicist, radiologist -->
  <enumerationdef>
    <attributedef name="audience"/>
    <subjectdef keyref="users"/>
  </enumerationdef>
</subjectScheme>
```

When the above subject scheme map is used, the only valid values for the `@audience` attribute are "therapist", "oncologist", "physicist", and "radiologist". Note that "users" is not a valid value for the `@audience` attribute; it merely identifies the parent or container subject.

Example: Binding an attribute to an empty set

The following code fragment declares that there are no valid values for the `@outputclass` attribute.

```
<subjectScheme>
  <enumerationdef>
    <attributedef name="outputclass"/>
    <subjectdef/>
  </enumerationdef>
</subjectScheme>
```

Processing controlled attribute values

An enumeration of controlled values can be defined with hierarchical levels by nesting subject definitions. This affects how processors perform filtering and flagging.

The following algorithm applies when processors apply filtering and flagging rules to attribute values that are defined as a hierarchy of controlled values and bound to an enumeration:

1. If an attribute specifies a value in the taxonomy, and a DITAVAL or other categorization tool is configured with that value, the rule matches.
2. Otherwise, if the parent value in the taxonomy has a rule, that matches.
3. Otherwise, continue up the chain in the taxonomy until a matching rule is found.

The following behavior is expected of processors:

- Processors *SHOULD* be aware of the hierarchies of attribute values that are defined in subject scheme maps for purposes of filtering, flagging, or other metadata-based categorization.
- Processors *SHOULD* validate that the values of attributes that are bound to controlled values contain only valid values from those sets. (The list of controlled values is not validated by basic XML parsers.) If the controlled values are part of a named key scope, the scope name is ignored for the purpose of validating the controlled values.
- Processors *SHOULD* check that all values listed for an attribute in a DITAVAL file are bound to the attribute by the subject scheme before filtering or flagging. If a processor encounters values that are not included in the subject scheme, it *SHOULD* issue a warning.

Example: A hierarchy of controlled values and conditional processing

The following example illustrates a set of controlled values that contains a hierarchy.

```
<subjectScheme>
  <subjectdef keys="users">
    <subjectdef keys="therapist">
      <subjectdef keys="novice-therapist"/>
      <subjectdef keys="expert-therapist"/>
    </subjectdef>
    <subjectdef keys="oncologist"/>
    <subjectdef keys="physicist"/>
    <subjectdef keys="radiologist"/>
  </subjectdef>
  <enumerationdef>
    <attributedef name="audience"/>
    <subjectdef keyref="users"/>
  </enumerationdef>
</subjectScheme>
```

Processors that are aware of the hierarchy that is defined in the subject scheme map will handle filtering and flagging in the following ways:

- If "therapist" is excluded, both "novice-therapist" and "expert-therapist" are by default excluded (unless they are explicitly set to be included).
- If "therapist" is flagged and "novice-therapist" is not explicitly flagged, processors automatically should flag "novice" since it is a type of therapist.

Extending subject schemes

The `<schemeref>` element provides a mechanism for extending a subject scheme. This makes it possible to add new relationships to existing subjects and extend enumerations of controlled values.

The `<schemeref>` element provides a reference to another subject scheme map. Typically, the referenced subject-scheme map defines a base set of controlled values that are extended by the current subject-scheme map. The values in the referenced subject-scheme map are merged with the values in the current subject-scheme map; the result is equivalent to specifying all of the values in a single subject scheme map.

Scaling a list of controlled values to define a taxonomy

Optional classification elements make it possible to create a taxonomy from a list of controlled values.

A taxonomy differs from a controlled values list primarily in the degree of precision with which the metadata values are defined. A controlled values list sometimes is regarded as the simplest form of taxonomy. Regardless of whether the goal is a simple list of controlled values or a taxonomy:

- The same core elements are used: `<subjectScheme>` and `<subjectdef>`.
- A category and its subjects can have a binding that enumerates the values of an attribute.

Beyond the core elements and the attribute binding elements, sophisticated taxonomies can take advantage of some optional elements. These optional elements make it possible to specify more precise relationships among subjects. The `<hasNarrower>`, `<hasPart>`, `<hasKind>`, `<hasInstance>`, and `<hasRelated>` elements specify the kind of relationship in a hierarchy between a container subject and its contained subjects.

While users who have access to sophisticated processing tools benefit from defining taxonomies with this level of precision, other users can safely ignore this advanced markup and define taxonomies with hierarchies of `<subjectdef>` elements that are not precise about the kind of relationship between the subjects.

Example: A taxonomy defined using subject scheme elements

The following example defines San Francisco as both an instance of a city and a geographic part of California.

```
<subjectScheme>
  <hasInstance>
    <subjectdef keys="city" navtitle="City">
      <subjectdef keys="la" navtitle="Los Angeles"/>
      <subjectdef keys="nyc" navtitle="New York City"/>
      <subjectdef keys="sf" navtitle="San Francisco"/>
    </subjectdef>
    <subjectdef keys="state" navtitle="State">
      <subjectdef keys="ca" navtitle="California"/>
      <subjectdef keys="ny" navtitle="New York"/>
    </subjectdef>
  </hasInstance>
  <hasPart>
    <subjectdef keys="place" navtitle="Place">
      <subjectdef keyref="ca">
        <subjectdef keyref="la"/>
        <subjectdef keyref="sf"/>
      </subjectdef>
      <subjectdef keyref="ny">
        <subjectdef keyref="nyc"/>
      </subjectdef>
    </subjectdef>
  </hasPart>
</subjectScheme>
```

Sophisticated tools can use this subject scheme map to associate content about San Francisco with related content about other California places or with related content about other cities (depending on the interests of the current user).

The subject scheme map also can define relationships between subjects that are not hierarchical. For instance, cities sometimes have "sister city" relationships. An information architect could add a `<subjectRelTable>` element to define these associative relationships, with a row for each sister-city pair and the two cities in different columns in the row.

Classification maps

A classification map is a DITA map in which the classification domain has been made available.

The classification domain provides elements that enable map authors to indicate information about the subject matter of DITA topics. The subjects must be defined in subjectScheme maps, and the map authors references the subjects using the @keyref attribute.

Examples of subject scheme maps

This section contains examples and scenarios that illustrate the use of subject scheme maps.

Example: How hierarchies defined in a subject scheme map affect filtering

This scenario demonstrates how a processor evaluates attribute values when it performs conditional processing for an attribute that is bound to a set of controlled values.

A company defines a subject category for "Operating system," with a key set to "os. There are sub-categories for Linux, Windows, and z/OS, as well as specific Linux variants: Red Hat Linux and SuSE Linux. The company then binds the values that are enumerated in the "Operating system" category to the @platform attribute.

```

<!-- This examples uses @navtitle rather than <navtitle> solely
to conserve space. Best practises for translate include using <navtitle>. -->
<subjectScheme>
  <subjectdef keys="os" navtitle="Operating system">
    <subjectdef keys="linux" navtitle="Linux">
      <subjectdef keys="redhat" navtitle="RedHat Linux"/>
      <subjectdef keys="suse" navtitle="SuSE Linux"/>
    </subjectdef>
    <subjectdef keys="windows" navtitle="Windows"/>
    <subjectdef keys="zos" navtitle="z/OS"/>
  </subjectdef>
  <enumerationdef>
    <attributedef name="platform"/>
    <subjectdef keyref="os"/>
  </enumerationdef>
</subjectScheme>

```

The enumeration limits valid values for the @platform attribute to the following: "linux", "redhat", "suse", "windows", and "zos". If any other values are encountered, processors validating against the scheme should issue a warning.

The following table illustrates how filtering and flagging operate when the above map is processed by a processor. The first two columns provide the values specified in the DITaval file; the third and fourth columns indicate the results of the filtering or flagging operation

att="platform" val="linux"	att="platform" val="redhat"	How platform="redhat" is evaluated	How platform="linux" is evaluated
action="exclude"	action="exclude"	Excluded.	Excluded.
	action="include" or action="flag"	Excluded. This is an error condition, because if all "linux" content is excluded, "redhat" also is excluded. Applications may recover by	Excluded.

att="platform" val="linux"	att="platform" val="redhat"	How platform="redhat" is evaluated	How platform="linux" is evaluated
		generating an error message.	
	Unspecified	Excluded, because "redhat" is a kind of "linux", and "linux" is excluded.	Excluded.
action="include"	action="exclude"	Excluded, because all "redhat" content is excluded.	Included.
	action="include"	Included.	Included.
	action="flag"	Included and flagged with the "redhat" flag.	Included.
	Unspecified	Included, because all "linux" content is included.	Included.
action="flag"	action="exclude"	Excluded, because all "redhat" content is excluded.	Included and flagged with the "linux" flag.
	action="include"	Included and flagged with the "linux" flag, because "linux" is flagged and "redhat" is a type of "linux".	Included and flagged with the "linux" flag.
	action="flag"	Included and flagged with the "redhat" flag, because a flag is available that is specifically for "redhat".	Included and flagged with the "linux" flag.
	Unspecified	Included and flagged with the "linux" flag, because "linux" is flagged and "redhat" is a type of linux	Included and flagged with the "linux" flag.
Unspecified	action="exclude"	Excluded, because all "redhat" content is excluded	If the default for @platform values is "include", this is included. If the default for @platform values is "exclude", this is excluded.
	action="include"	Included.	Included, because all "redhat" content is included, and general Linux content also applies to RedHat

att="platform" val="linux"	att="platform" val="redhat"	How platform="redhat" is evaluated	How platform="linux" is evaluated
	action="flag"	Included and flagged with the "redhat" flag.	Included, because all "redhat" content is included, and general Linux content also applies to RedHat
	Unspecified	If the default for @platform values is "include", this is included. If the default for @platform values is "exclude", this is excluded.	If the default for @platform values is "include", this is included. If the default for @platform values is "exclude", this is excluded.

Example: Extending a subject scheme

You can extend a subject scheme by creating another subject scheme map and referencing the original map using a `<schemeref>` element. This enables information architects to add new relationships to existing subjects and extend enumerations of controlled values.

A company uses a common subject scheme map (`baseOS.ditamap`) to set the values for the `@platform` attribute.

```
<subjectScheme>
  <subjectdef keys="os" navtitle="Operating system">
    <subjectdef keys="linux" navtitle="Linux">
      <subjectdef keys="redhat" navtitle="RedHat Linux"/>
      <subjectdef keys="suse" navtitle="SuSE Linux"/>
    </subjectdef>
    <subjectdef keys="windows" navtitle="Windows"/>
    <subjectdef keys="zos" navtitle="z/OS"/>
  </subjectdef>
  <enumerationdef>
    <attributedef name="platform"/>
    <subjectdef keyref="os"/>
  </enumerationdef>
</subjectScheme>
```

The following subject scheme map extends the enumeration defined in `baseOS.ditamap`. It adds "macos" as a child of the existing "os" subject; it also adds special versions of Windows as children of the existing "windows" subject:

```
<subjectScheme>
  <schemeref href="baseOS.ditamap"/>
  <subjectdef keyref="os">
    <subjectdef keys="macos" navtitle="Macintosh"/>
  </subjectdef>
  <subjectdef keyref="windows">
    <subjectdef keys="winxp" navtitle="Windows XP"/>
    <subjectdef keys="winvis" navtitle="Windows Vista"/>
  </subjectdef>
</subjectScheme>
```

Note that the references to the subjects that are defined in `baseOS.ditamap` use the `@keyref` attribute. This avoids duplicate definitions of the keys and ensures that the new subjects are added to the base enumeration.

The effective result is the same as the following subject scheme map:

```
<subjectScheme>
  <subjectdef keys="os" navtitle="Operating system">
    <subjectdef keys="linux" navtitle="Linux">
      <subjectdef keys="redhat" navtitle="RedHat Linux"/>
      <subjectdef keys="suse" navtitle="SuSE Linux"/>
    </subjectdef>
    <subjectdef keys="macos" navtitle="Macintosh"/>
    <subjectdef keys="windows" navtitle="Windows">
      <subjectdef keys="winxp" navtitle="Windows XP"/>
      <subjectdef keys="win98" navtitle="Windows Vista"/>
    </subjectdef>
    <subjectdef keys="zos" navtitle="z/OS"/>
  </subjectdef>
  <enumerationdef>
    <attributedef name="platform"/>
    <subjectdef keyref="os"/>
  </enumerationdef>
</subjectScheme>
```

Example: Extending a subject scheme upwards

You can broaden the scope of a subject category by creating a new subject scheme map that defines the original subject category as a child of a broader category.

The following subject scheme map creates a "Software" category that includes operating systems as well as applications. The subject scheme map that defines the operation system subjects is pulled in by reference, while the application subjects are defined directly in the subject scheme map below.

```
<subjectScheme>
  <schemeref href="baseOS.ditamap"/>
  <subjectdef keys="sw" navtitle="Software">
    <subjectdef keyref="os"/>
    <subjectdef keys="app" navtitle="Applications">
      <subjectdef keys="apacheserv" navtitle="Apache Web Server"/>
      <subjectdef keys="mysql" navtitle="MySQL Database"/>
    </subjectdef>
  </subjectdef>
</subjectScheme>
```

If the subject scheme that is defined in `baseOS.ditamap` binds the "os" subject to the `@platform` attribute, the app subjects that are defined in the extension subject scheme do not become part of that enumeration, since they are not part of the "os" subject

To enable the upward extension of an enumeration, information architects can define the controlled values in one subject scheme map and bind the controlled values to the attribute in another subject scheme map. This approach will let information architects bind an attribute to a different set of controlled values with less rework.

An adopter would use the extension subject scheme as the subject scheme that governs the controlled values. Any subject scheme maps that are referenced by the extension subject scheme are effectively part of the extension subject scheme.

Example: Defining values for @deliveryTarget

You can use a subject scheme map to define the values for the @deliveryTarget attribute. This filtering attribute, which is new in DITA 1.3, is intended for use with a set of hierarchical, controlled values.

In this scenario, one department produces electronic publications (EPUB, EPUB2, EPUB3, Kindle, etc.) while another department produces traditional, print-focused output. Each department needs to exclude a certain category of content when they build documentation deliverables.

The following subject scheme map provides a set of values for the @deliveryTarget attribute that accommodates the needs of both departments.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE subjectScheme PUBLIC "-//OASIS//DTD DITA Subject Scheme Map//EN"
"subjectScheme.dtd">
<subjectScheme>
  <subjectHead>
    <subjectHeadMeta>
      <navtitle>Example of values for the @deliveryTarget attribute</navtitle>
      <shortdesc>Provides a set of values for use with the
        @deliveryTarget conditional-processing attribute. This set of values is
        illustrative only; you can use any values with the @deliveryTarget
        attribute.</shortdesc>
    </subjectHeadMeta>
  </subjectHead>
  <subjectdef keys="deliveryTargetValues">
    <topicmeta><navtitle>Values for @deliveryTarget attributes</navtitle></topicmeta>
    <!-- A tree of related values -->
    <subjectdef keys="print">
      <topicmeta><navtitle>Print-primary deliverables</navtitle></topicmeta>
      <subjectdef keys="pdf">
        <topicmeta><navtitle>PDF</navtitle></topicmeta>
      </subjectdef>
      <subjectdef keys="css-print">
        <topicmeta><navtitle>CSS for print</navtitle></topicmeta>
      </subjectdef>
      <subjectdef keys="xsl-fo">
        <topicmeta><navtitle>XSL-FO</navtitle></topicmeta>
      </subjectdef>
      <subjectdef keys="afp">
        <topicmeta><navtitle>Advanced Function Printing</navtitle></topicmeta>
      </subjectdef>
      <subjectdef keys="ms-word">
        <topicmeta><navtitle>Microsoft Word</navtitle></topicmeta>
      </subjectdef>
      <subjectdef keys="indesign">
        <topicmeta><navtitle>Adobe InDesign</navtitle></topicmeta>
      </subjectdef>
      <subjectdef keys="open-office">
        <topicmeta><navtitle>Open Office</navtitle></topicmeta>
      </subjectdef>
    </subjectdef>
    <subjectdef keys="online">
      <topicmeta><navtitle>Online deliverables</navtitle></topicmeta>
      <subjectdef keys="html-based">
        <topicmeta><navtitle>HTML-based deliverables</navtitle></topicmeta>
        <subjectdef keys="html">
          <topicmeta><navtitle>HTML</navtitle></topicmeta>
          <subjectdef keys="html5">
            <topicmeta><navtitle>HTML5</navtitle></topicmeta>
          </subjectdef>
        </subjectdef>
      </subjectdef>
      <subjectdef keys="help">
        <topicmeta><navtitle>Contextual help</navtitle></topicmeta>
        <subjectdef keys="htmlhelp">
```

```

    <topicmeta><navtitle>HTML Help</navtitle></topicmeta>
  </subjectdef>
  <subjectdef keys="webhelp">
    <topicmeta><navtitle>Web help</navtitle></topicmeta>
  </subjectdef>
  <subjectdef keys="javahelp">
    <topicmeta><navtitle>Java Help</navtitle></topicmeta>
  </subjectdef>
  <subjectdef keys="eclipseinfocenter">
    <topicmeta><navtitle>Eclipse InfoCenter</navtitle></topicmeta>
  </subjectdef>
</subjectdef>
<subjectdef keys="epub">
  <topicmeta><navtitle>EPUB</navtitle></topicmeta>
  <subjectdef keys="epub2">
    <topicmeta><navtitle>EPUB2</navtitle></topicmeta>
  </subjectdef>
  <subjectdef keys="epub3">
    <topicmeta><navtitle>EPUB3</navtitle></topicmeta>
  </subjectdef>
  <subjectdef keys="ibooks">
    <topicmeta><navtitle>iBooks</navtitle></topicmeta>
  </subjectdef>
  <subjectdef keys="nook">
    <topicmeta><navtitle>nook</navtitle></topicmeta>
  </subjectdef>
</subjectdef>
<subjectdef keys="kindle">
  <topicmeta><navtitle>Amazon Kindle</navtitle></topicmeta>
  <subjectdef keys="kindle8">
    <topicmeta><navtitle>Kindle Version 8</navtitle></topicmeta>
  </subjectdef>
</subjectdef>
</subjectdef>
</subjectdef>
</subjectdef>
</subjectdef>
</subjectdef>
<enumerationdef>
  <attributedef name="deliveryTarget"/>
  <subjectdef keyref="deliveryTargetValues"/>
</enumerationdef>
</subjectScheme>

```

DITA metadata

Metadata can be applied in both DITA topics and DITA maps. Metadata that is assigned in DITA topics can be supplemented or overridden by metadata that is assigned in a DITA map; this design facilitates the reuse of DITA topics in different DITA maps and use-specific contexts.

Metadata elements

The metadata elements, many of which map to Dublin core metadata, are available in topics and DITA maps. This design enables authors and information architects to use identical metadata markup in both topics and maps.

The `<metadata>` element is a wrapper element that contains many of the metadata elements. In topics, the `<metadata>` element is available in the `<prolog>` element. In maps, the `<metadata>` element is available in the `<topicmeta>` element.

In DITA maps, the metadata elements also are available directly in the `<topicmeta>` element. Collections of metadata can be shared between DITA maps and topics by using the `conref` or `keyref` mechanism.

In general, specifying metadata in a `<topicmeta>` element is equivalent to specifying it in the `<prolog>` element of a referenced topic. The value of specifying the metadata at the map level is that the topic then can be reused in other maps where different metadata might apply. Many items in the `<topicmeta>` element also cascade to nested `<topicref>` elements within the map.



Note: Not all metadata elements are available in the `<metadata>` element. However, they are available in either the topic `<prolog>` element or the map `<topicmeta>` element.

Metadata attributes

Certain attributes are common across most DITA elements. These attributes support content referencing, conditional processing, application of metadata, and globalization and localization.

Conditional processing attributes

The metadata attributes specify properties of the content that can be used to determine how the content should be processed. Specialized metadata attributes can be defined to enable specific business-processing needs, such as semantic processing and data mining.

Metadata attributes typically are used for the following purposes:

- Filtering content based on the attribute values, for example, to suppress or publish profiled content
- Flagging content based on the attribute values, for example, to highlight specific content on output
- Performing custom processing, for example, to extract business-critical data and store it in a database

Typically `@audience`, `@platform`, `@product`, `@otherprops`, `@props`, `@deliveryTarget`, and specializations of the `@props` attributes are used for filtering; the same attributes plus the `@rev` attribute are used for flagging. The `@status` and `@importance` attributes, as well as custom attributes specialized from `@base`, are used for application-specific behavior, such as identifying metadata to aid in search and retrieval.

Filtering and flagging attributes

The following conditional-processing attributes are available on most elements:

`@product`

The product that is the subject of the discussion.

`@platform`

The platform on which the product is deployed.

`@audience`

The intended audience of the content.

`@deliveryTarget`

The intended delivery target of the content, for example "html", "pdf", or "epub". This attribute is a replacement for the now deprecated `@print` attribute.

The `@deliveryTarget` attribute is specialized from the `@props` attribute. It is defined in the `deliveryTargetAttDomain`, which is integrated into all OASIS-provided document-type shells. If this domain is not integrated into a given document-type shell, the `@deliveryTarget` attribute will not be available.

`@rev`

The revision or draft number of the current document. (This is used only for flagging.)

@otherprops

Other properties that do not require semantic identification.

@props

A generic conditional processing attribute that can be specialized to create new semantic conditional-processing attributes.

Other metadata attributes

Other attributes are still considered metadata on an element, but they are not designed for filtering or flagging.

@importance

The degree of priority of the content. This attribute takes a single value from an enumeration.

@status

The current state of the content. This attribute takes a single value from an enumeration.

@base

A generic attribute that has no specific purpose, but is intended to act as the basis for specialized attributes that have a simple value syntax like the conditional processing attributes (one or more alphanumeric values separated by whitespace or parenthesized groups of values).

@outputclass

Provides a label on one or more element instances, typically to specify a role or other semantic distinction. As the `@outputclass` attribute does not provide a formal type declaration or the structural consistency of specialization, it should be used sparingly, usually only as a temporary measure while a specialization is developed. For example, `<uicontrol>` elements that define button labels could be distinguished by adding an `@outputclass` attribute:

```
<uicontrol outputclass="button">Cancel</uicontrol>
```

The value of the `@outputclass` attribute can be used to trigger XSLT or CSS rules, while providing a mapping to be used for future migration to a more specialized set of user interface elements.

Translation and localization attributes

DITA elements have several attributes that support localization and translation.

@xml:lang

Identifies the language of the content, using the standard language and country codes. For instance, French Canadian is identified by the value `fr-CA`. The `@xml:lang` attribute asserts that all content and attribute values within the element bearing the attribute are in the specified language, except for contained elements that declare a different language.

@translate

Determines whether the element requires translation. A default value can often be inferred from the element type. For example, `<apiname>` may be untranslated by default, whereas `<p>` may be translated by default.

@dir

Determines the direction in which the content should be rendered.

Architectural attributes

The architectural attributes specify the version of DITA that the content supports; they also identify the DITA domains, structural types, and specializations that are in use by the content.

The architectural attributes should not be marked up in the source DITA map and topics. Instead, the values of the architectural attributes are handled by the processor when the content is processed, preferably through defaults set in the XML grammar. This practice ensures that the DITA content instances do not specify invalid values for the architectural attributes.

The architectural attributes are as follows:

@class

This attribute identifies the specialization hierarchy for the element type. Every DITA element (except the `<dita>` element that is used as the root of a ditabase document) *MUST* declare a `@class` attribute.

@domains

This attribute identifies the domain modules (and optionally the structural modules) that are used in a map or topic. Each module also declares its module dependencies. The root element of every topic and map *MUST* declare a `@domains` attribute.

@DITAArchVersion

This attribute identifies the version of the DITA architecture that is used by the XML grammar. The root element of every topic and map *MUST* declare a `@DITAArchVersion` attribute. The attribute is declared in a DITA namespace to allow namespace-sensitive tools to detect DITA markup.

To make the document instance usable in the absence of an XML grammar, a normalization process can set the architectural attributes in the document instance.

Metadata in maps and topics

Topic metadata can be specified in a DITA map as well as in the topics that the map references. By default, metadata in the map supplements or overrides metadata that is specified at the topic level, unless the `@lockmeta` attribute of the `<topicmeta>` element is set to "no".

Where metadata about topics can be specified

Information about topics can be specified as metadata on the map, as attributes on the `<topicref>` element, or as metadata attributes or elements in the topic itself:

DITA map: Metadata elements

At the map level, properties can be set by using metadata elements. They can be set for an individual topic, for a set of topics, or globally for the entire document. The metadata elements are authored within a `<topicmeta>` element, which associates metadata with the parent element and its children. Because the topics in a branch of the hierarchy typically have some common subjects or properties, this is a convenient mechanism to define properties for a set of topics. For example, the `<topicmeta>` element in a `<relcolspec>` can associate metadata with all the topics that are referenced in the `<reltable>` column.

A map can override or supplement everything about a topic except its primary title and body content. All the metadata elements that are available in a topic also are available in a map. In addition, a map may provide alternate titles and a short description. The alternate titles can override their equivalents in the topic. The short description in the map *MAY* override the short description in the topic if the `<topicref>` element specifies a `@copy-to` attribute.

DITA map: Attributes of the <topicref> element

At the map level, properties can be set as attributes of the <topicref> element.

DITA topic

Within a topic, authors can either set metadata attributes on the root element or add metadata elements in the <prolog> element.

How metadata set at both the map and topic level intersects

In a topic, the metadata elements apply to the entire topic. In a map, they supplement or override any metadata that is provided in the referenced topics. When the same metadata element or attribute is specified in both a map and a topic, by default the value in the map takes precedence; the assumption here is that the author of the map has more knowledge of the reusing context than the author of the topic. The @lockmeta attribute on the <topicmeta> element controls whether map-specified values override values in the referenced topic.

The <navtitle> element is an exception to the rule of how metadata specified by the <topicmeta> element cascades. The content of the <navtitle> element is used as a navigation title only if the @locktitle attribute of the parent <topicref> element is set to "yes".

Cascading of metadata attributes in a DITA map

Certain map-level attributes cascade throughout a map, which facilitates attribute and metadata management. When attributes *cascade*, they apply to the elements that are children of the element where the attributes were specified. Cascading applies to a containment hierarchy, as opposed to a element-type hierarchy.

The following attributes cascade when set on the <map> element or when set within a map:

- @audience, @platform, @product, @otherprops, @rev
- @props and any attribute specialized from @props
- @linking, @toc, @print, @search
- @format, @scope, @type
- @xml:lang, @dir, @translate
- @processing-role
- @cascade

Cascading is additive for attributes that accept multiple values, except when the @cascade attribute is set to avoid adding values to attributes. For attributes that take a single value, the closest value defined on a containing element takes effect. In a relationship table, row-level metadata is considered more specific than column-level metadata, as shown in the following containment hierarchy:

- <map> (most general)
 - <topicref> container (more specific)
 - <topicref> (most specific)
 - <reltable> (more specific)
 - <relcolspec> (more specific)
 - <relrow> (more specific)
 - <topicref> (most specific)

Merging of cascading attributes

The @cascade attribute can be used to modify the additive nature of attribute cascading (though it does not turn off cascading altogether). The attribute has two predefined values: "merge" and "nomerge".

cascade="merge"

The metadata attributes cascade; the values of the metadata attributes are additive. This is the processing default for the @cascade attribute and was the only defined behavior for DITA 1.2 and earlier.

cascade="nomerge"

The metadata attributes cascade; however, they are not additive for <topicref> elements that specify a different value for a specific metadata attribute. If the cascading value for an attribute is already merged based on multiple ancestor elements, that merged value continues to cascade until a new value is encountered (that is, setting cascade="nomerge" does not undo merging that took place on ancestors).

Implementers *MAY* define their own custom, implementation-specific tokens. To avoid name conflicts between implementations or with future additions to the standard, implementation-specific tokens *SHOULD* consist of a prefix that gives the name or an abbreviation for the implementation followed by a colon followed by the token or method name.

For example, a processor might define the token "appToken:audience" in order to specify cascading and merging behaviors for **only** the @audience attribute. The following rules apply:

- The predefined values for the @cascade attribute *MUST* precede any implementation-specific tokens, for example, cascade="merge appToken:audience".
- Tokens can apply to a set of attributes, specified as part of the @cascade value. In that case, the syntax for specifying those values consists of the implementation-specific token, followed by a parenthetical group that uses the same syntax as groups within the @audience, @platform, @product, and @otherprops attributes. For example, a token that applies to only @platform and @product could be specified as cascade="appname:token(platform product)".

Examples of the @cascade attribute in use

Consider the following code examples:

```
<map audience="a b" cascade="merge">
  <topicref href="topic.dita" audience="c"/>
</map>
```

Figure 3: Map A

```
<map audience="a b" cascade="nomerge">
  <topicref href="topic.dita" audience="c"/>
</map>
```

Figure 4: Map B

For map A, the values for the attribute are merged, and the effective value of the @audience attribute for topic.dita is "a b c". For map B, the values for the attribute are not additive, and the effective value of the @audience attribute for topic.dita is "c".

In the following example, merging is active at the map level but turned off below:

```
<map platform="a" product="x" cascade="merge">
  <topicref href="one.dita" platform="b" product="y">
    <topicref href="two.dita" cascade="nomerge" product="z"/>
  </topicref>
</map>
```

Figure 5: Map C

In map C, the reference to one.dita has effective merged values of "a b" for @platform and "x y" for @product.

The reference to `two.dita` turns off merging, so the explicit `@product` value of "z" is used (it does not merge with ancestor values). The `@platform` attribute is not present, so the already-merged value of "a b" continues to cascade and the effective value of `@platform` on this reference.

Order for processing cascading attributes in a map

When determining the value of an attribute, processors *MUST* evaluate each attribute on each individual element in a specific order; this order is specified in the following list. Applications *MUST* continue through the list until a value is established or until the end of the list is reached (at which point no value is established for the attribute). In essence, the list provides instructions on how processors can construct a map where all attribute values are set and all cascading is complete.

For example, in the case of `<topicref toc="yes">`, applications *MUST* stop at item 2 on page 43 in the list; a value is specified for `@toc` in the document instance, so `@toc` values from containing elements will not cascade to that specific `<topicref>` element. The `toc="yes"` setting on that `<topicref>` element will cascade to contained elements, provided those elements reach item 5 on page 43 below when evaluating the `@toc` attribute.

For attributes within a map, the following processing order *MUST* occur:

1. The `@conref` and `@keyref` attributes are evaluated.
2. The explicit values specified in the document instance are evaluated. For example, a `<topicref>` element with the `@toc` attribute set to "no" will use that value.
3. The default or fixed attribute values are evaluated. For example, the `@toc` attribute on the `<reltable>` element has a default value of "no".
4. The default values that are supplied by a controlled values file are evaluated.
5. The attributes cascade.
6. The processing-supplied default values are applied.
7. After the attributes are resolved within the map, they cascade to referenced maps.



Note: The processing-supplied default values do not cascade to other maps. For example, most processors will supply a default value of `toc="yes"` when no `@toc` attribute is specified. However, a processor-supplied default of `toc="yes"` *MUST* not override a value of `toc="no"` that is set on a referenced map. If the `toc="yes"` value is explicitly specified, is given as a default through a DTD, XSD, RNG, or controlled values file, or cascades from a containing element in the map, it *MUST* override a `toc="no"` setting on the referenced map. See [Map-to-map cascading behaviors](#) on page 46 for more details.

8. Repeat steps 1 on page 43 to 4 on page 43 for each referenced map.
9. The attributes cascade within each referenced map.
10. The processing-supplied default values are applied within each referenced map.
11. Repeat the process for maps referenced within the referenced maps.

Reconciling topic and map metadata elements

The `<topicmeta>` element in maps contains numerous elements that can be used to declare metadata. These metadata elements have an effect on the parent `<topicref>` element, any child `<topicref>` elements, and – if a direct child of the `<map>` element – on the map as a whole.

For each element that can be contained in the `<topicmeta>` element, the following table addresses the following questions:

How does it apply to the topic?

This column describes how the metadata specified within the `<topicmeta>` element interacts with the metadata specified in the topic. In most cases, the properties are additive. For example, when the

<audience> element is set to "user" at the map level, the value "user" is added during processing to any audience metadata that is specified within the topic.

Does it cascade to other topics in the map?


This column indicates whether the specified metadata value cascades to nested <topicref> elements. For example, when an <audience> element is set to "user" at the map level, all child <topicref> elements implicitly have an <audience> element set to "user" also. Elements that can apply only to the specific <topicref> element, such as <linktext>, do not cascade.

What is the purpose when specified on the <map> element?

The map element allows metadata to be specified for the entire map. This column describes what effect, if any, an element has when specified at this level.

Table 1: Topicmeta elements and their properties

Element	How does it apply to the topic?	Does it cascade to child <topicref> elements?	What is the purpose when set on the <map> element?
<audience>	Add to the topic	Yes	Specify an audience for the entire map
<author>	Add to the topic	Yes	Specify an author for the entire map
<category>	Add to the topic	Yes	Specify a category for the entire map
<copyright>	Add to the topic	Yes	Specify a copyright for the entire map
<critdates>	Add to the topic	Yes	Specify critical dates for the entire map
<data>	Add to the topic	No, unless specialized for a purpose that cascades	No stated purpose, until the element is specialized
<data-about>	Add the property to the specified target	No, unless specialized for a purpose that cascades	No stated purpose, until the element is specified
<foreign>	Add to the topic	No, unless specialized for a purpose that cascades	No stated purpose, until the element is specified
<keywords>	Add to the topic	No	No stated purpose
<linktext>	Not added to the topic; applies only to links created based on this occurrence in the map	No	No stated purpose
<metadata>	Add to the topic	Yes	Specify metadata for the entire map
<navtitle>	Not added to the topic; applies only to navigation that is created based on this occurrence in the map. The navigation title	No	No stated purpose

Element	How does it apply to the topic?	Does it cascade to child <topicref> elements?	What is the purpose when set on the <map> element?
	will be used whenever the @locktitle attribute on the containing <topicref> element is set to "yes".		
<othermeta>	Add to the topic	No	Define metadata for the entire map
<permissions>	Add to the topic	Yes	Specify permissions for the entire map
<proinfo>	Add to the topic	Yes	Specify product info for the entire map
<publisher>	Add to the topic	Yes	Specify a publisher for the map
<resourceid>	Add to the topic	No	Specify a resource ID for the map
<searchtitle>	Replace the one in the topic. If multiple <searchtitle> elements are specified for a single target, processors may choose to issue a warning.	No	No stated purpose
<shortdesc>	Only added to the topic when the <topicref> element specifies a @copy-to attribute. Otherwise, it applies only to links created based on this occurrence in the map.  Note: Processors <i>MAY</i> or <i>MAY NOT</i> implement this behavior.	No	Provide a description of the map
<source>	Add to the topic	No	Specify a source for the map
<unknown>	Add to the topic	No, unless specialized for a purpose that cascades	No stated purpose, until the element is specified
<ux-window>	Not added to the topic	No	Definitions are global, so setting at map level is equivalent to setting anywhere else.

Example of metadata elements cascading in a DITA map

The following code sample illustrates how an information architect can apply certain metadata to all the DITA topics in a map:

```
<map title="DITA maps" xml:lang="en-us">
  <topicmeta>
    <author>Kristen James Eberlein</author>
    <copyright>
      <copyyear year="2009"/>
      <copyrholder>OASIS</copyrholder>
    </copyright>
  </topicmeta>
  <topicref href="dita_maps.dita" navtitle="DITA maps">
    <topicref href="definition_ditamaps.dita" navtitle="Definition of DITA maps"></
topicref>
  <topicref href="purpose_ditamaps.dita" navtitle="Purpose of DITA maps"></
topicref>
  <!-- ... -->
</topicref>
</map>
```

The author and copyright information cascades to each of the DITA topics referenced in the DITA map. When the DITA map is processed to XHTML, for example, each XHTML file contains the metadata information.

Map-to-map cascading behaviors

When a DITA map (or branch of a DITA map) is referenced by another DITA map, by default, certain rules apply. These rules pertain to the cascading behaviors of attributes, metadata elements, and roles assigned to content (for example, the role of "Chapter" assigned by a `<chapter>` element). Attributes and elements that cascade within a map generally follow the same rules when cascading from one map to another map, but there are some exceptions and additional rules that apply.

Cascading of attributes from map to map

Certain elements cascade from map to map, although some of the attributes that cascade within a map do not cascade from map to map.

The following attributes cascade from map to map:

- @audience, @platform, @product, @otherprops, @rev
- @props and any attribute specialized from @props
- @linking, @toc, @print, @search
- @type
- @translate
- @processing-role
- @cascade

Note that the above list excludes the following attributes:

@format

The @format attribute must be set to "ditamap" in order to reference a map or a branch of a map, so it cannot cascade through to the referenced map.

@xml:lang and @dir

Cascading behavior for @xml:lang is defined in [The xml:lang attribute](#) on page 110. The @dir attribute work the same way.

@scope

The value of the @scope attribute describes the map itself, rather than the content. When the @scope attribute is set to "external", it indicates that the referenced map itself is external and unavailable, so the value cannot cascade into that referenced map.

The @class attribute is used to determine the processing roles that cascade from map to map. See [Cascading of roles from map to map](#) on page 48 for more information.

As with values that cascade within a map, the cascading is additive if the attribute permits multiple values (such as @audience). When the attribute only permits one value, the cascading value overrides the top-level element.

Example of attributes cascading between maps

For example, assume the following references in test.ditamap:

```
<map>
  <topicref href="a.ditamap" format="ditamap" toc="no"/>
  <mapref href="b.ditamap" audience="developer"/>
  <topicref href="c.ditamap#branch1" format="ditamap" print="no"/>
  <mapref href="c.ditamap#branch2" platform="myPlatform"/>
</map>
```

- The map a.ditamap is treated as if toc="no" is specified on the root <map> element. This means that the topics that are referenced by a.ditamap do not appear in the navigation generated by test.ditamap (except for branches within the map that explicitly set toc="yes").
- The map b.ditamap is treated as if audience="developer" is set on the root <map> element. If the @audience attribute is already set on the root <map> element within b.ditamap, the value "developer" is added to any existing values.
- The element with id="branch1" within the map c.ditamap is treated as if print="no" is specified on that element. This means that the topics within the branch with id="branch1" do not appear in the printed output generated by test.ditamap (except for nested branches within that branch that explicitly set print="yes").
- The element with id="branch2" within the map c.ditamap is treated as if platform="myPlatform" is specified on that element. If the @platform attribute is already specified on the element with id="branch", the value "myPlatform" is added to existing values.

Cascading of metadata elements from map to map

Elements that are contained within <topicmeta> or <metadata> elements follow the same rules for cascading from map to map as the rules that apply within a single DITA map.

For a complete list of which elements cascade within a map, see the column "Does it cascade to child <topicref> elements?" in the topic [Reconciling topic and map metadata elements](#) on page 43.



Note: It is possible that a specialization might define metadata that should replace rather than add to metadata in the referenced map, but DITA (by default) does not currently support this behavior.

For example, consider the following code examples:

```
<map>
  <topicref href="a.ditamap" format="ditamap">
    <topicmeta>
```



```

    <shortdesc>This map contains information about Acme defects.</shortdesc>
  </topicmeta>
</topicref>
<topicref href="b.ditamap" format="ditamap">
  <topicmeta>
    <audience type="programmer"/>
  </topicmeta>
</topicref>
<mapref href="c.ditamap" format="ditamap"/>
<mapref href="d.ditamap" format="ditamap"/>
</map>

```

Figure 6: test-2.ditamap

```

<map>
  <topicmeta>
    <audience type="writer"/>
  </topicmeta>
  <topicref href="b-1.dita"/>
  <topicref href="b-2.dita"/>
</map>

```

Figure 7: b.ditamap

When test-2.ditamap is processed, the following behavior occurs:

- Because the `<shortdesc>` element does not cascade, it does not apply to the DITA topics that are referenced in a.ditamap.
- Because the `<audience>` element cascades, the `<audience>` element in the reference to b.ditamap combines with the `<audience>` element that is specified at the top level of b.ditamap. The result is that the b-1.dita topic and b-2.dita topic are processed as though they each contained the following child `<topicmeta>` element:

```

<topicmeta>
  <audience type="programmer"/>
  <audience type="writer"/>
</topicmeta>

```

Cascading of roles from map to map

When specialized `<topicref>` elements (such as `<chapter>` or `<mapref>`) reference a map, they typically imply a semantic role for the referenced content.

The semantic role reflects the `@class` hierarchy of the referencing `<topicref>` element; it is equivalent to having the `@class` attribute from the referencing `<topicref>` cascade to the top-level `<topicref>` elements in the referenced map. Although this cascade behavior is not universal, there are general guidelines for when `@class` values should be replaced.

When a `<topicref>` element or a specialization of a `<topicref>` element references a DITA resource, it defines a role for that resource. In some cases this role is straightforward, such as when a `<topicref>` element references a DITA topic (giving it the already known role of "topic"), or when a `<mapref>` element references a DITA map (giving it the role of "DITA map").

Unless otherwise instructed, a specialized `<topicref>` element that references a map supplies a role for the referenced content. This means that, in effect, the `@class` attribute of the referencing element cascades to top-level `<topicref>` elements in the referenced map. In situations where this should not happen - such as all elements from the mapgroup domain - the non-default behavior should be clearly specified.

For example, when a `<chapter>` element from the bookmap specialization references a map, it supplies a role of "chapter" for each top-level `<topicref>` element in the referenced map. When the `<chapter>` element references a branch in another map, it supplies a role of "chapter" for that branch. The `@class` attribute for `<chapter>` ("-map/topicref bookmap/chapter ") cascades to the top-level `<topicref>` element in the nested map, although it does not cascade any further.

Alternatively, the `<mapref>` element in the mapgroup domain is a convenience element; the top-level `<topicref>` elements in the map referenced by a `<mapref>` element *MUST NOT* be processed as if they are `<mapref>` elements. The `@class` attribute from the `<mapref>` element ("map/topicref mapgroup-d/mapref ") does not cascade to the referenced map.

In some cases, preserving the role of the referencing element might result in out-of-context content. For example, a `<chapter>` element that references a bookmap might pull in `<part>` elements that contain nested `<chapter>` elements. Treating the `<part>` element as a `<chapter>` will result in a chapter that nests other chapters, which is not valid in bookmap and might not be understandable by processors. The result is implementation specific; processors *MAY* choose to treat this as an error, issue a warning, or simply assign new roles to the problematic elements.

Example of cascading roles between maps

Consider the scenario of a `<chapter>` element that references a DITA map. This scenario could take several forms:

Referenced map contains a single top-level `<topicref>` element

The entire branch functions as if it were included in the bookmap; the top-level `<topicref>` element is processed as if it were the `<chapter>` element.

Referenced map contains multiple top-level `<topicref>` elements

Each top-level `<topicref>` element is processed as if it were a `<chapter>` element (the referencing element).

Referenced map contains a single `<appendix>` element

The `<appendix>` element is processed as it were a `<chapter>` element.

Referenced map contains a single `<part>` element, with nested `<chapter>` elements.

The `<part>` element is processed as it were a chapter element. Nested `<chapter>` elements might not be understandable by processors; applications *MAY* recover as described above.

`<chapter>` element references a single `<topicref>` element rather than a map

The referenced `<topicref>` element is processed as if it were a `<chapter>` element.

Context hooks and window metadata for user assistance

Context hook information specified in the `<resourceid>` element in the DITA map or in a DITA topic enables processors to generate the header, map, alias and other types of support files that are required to integrate the user assistance with the application. Some user assistance topics might need to be displayed in a specific window or viewport, and this windowing metadata can be defined in the DITA map within the `<ux-window>` element.

Context hook and windowing information is ignored if the processor does not support this metadata.

User interfaces for software application often are linked to user assistance (such as help systems and tool tips) through *context hooks*. Context hooks are identifiers that associate a part of the user interface with the location of a help topic. Context hooks can be direct links to URIs, but more they are indirect links (numeric context identifiers and context strings) that can be processed into external resource files. Context hooks can be direct links to URIs,

but more often they are indirect links (numeric context identifiers and context strings) that can be processed into external resource files. These external resource and mapping files are then used directly by context-sensitive help systems and other downstream tool applications.

Context hooks can define either one-to-one or one-to-many relationships between user interface controls and target help content.

The metadata that is available in `<resourceid>` and `<ux-window>` provides flexibility for content developers:

- You can overload maps and topics with all the metadata needed to support multiple target help systems. This supports single-sourcing of help content and help metadata.
- You can choose whether to add `<resourceid>` metadata to `<topicref>` elements, `<prolog>` elements, or both. Context-dependent metadata might be best kept with maps, while persistent, context-independent metadata might best stay with topics in `<prolog>` elements.

Context hook information is defined within DITA topics and DITA maps through attributes of the `<resourceid>` element:

@appid

Specifies an identifier that is used by an application to identify the topic.

@ux-context-string

Contains the value of a user-assistance context string that is used to identify the topic.

@ux-source-priority

(For `<resourceid>` elements within maps) Contains a value that indicates the precedence of context hooks in the map and context hooks in the topic. This makes it possible to avoid problems where context hooks defined in the DITA map potentially conflict with those defined in the topics; the values of the `@ux-source-priority` attribute indicate how potential conflicts should be resolved.

(For `<resourceid>` elements within topics) This usage is undefined and reserved for future use. Processors should ignore the `@ux-source-priority` attribute.

@ux-windowref

References the name of the window to be used to display the help topic. The window characteristics are separately defined in a `<ux-window>` element in the DITA map.

In some help systems, a topic might need to be displayed in a specifically sized or featured window. For example, a help topic might need to be displayed immediately adjacent to the user interface control that it supports in a window of a specific size that always remains on top, regardless of the focus within the operating system.

Windowing metadata can be defined in the DITA map within the `<ux-window>` element.

The `<ux-window>` element provides the `@top`, `@left`, `@height`, `@width`, `@on-top`, `@features`, `@relative`, and `@full-screen` attributes.

DITA addressing

DITA provides two addressing mechanisms. DITA addresses either are direct URI-based addresses, or they are indirect key-based addresses. Within DITA documents, individual elements are addressed by unique identifiers specified on the @id attribute. DITA defines two fragment-identifier syntaxes; one is the full fragment-identifier syntax, and the other is an abbreviated fragment-identifier syntax that can be used when addressing non-topic elements from within the same topic.

ID attribute

The @id attribute assigns an identifier to DITA elements so that the elements can be referenced.

The @id attribute is available for most elements. An element must have a valid value for the @id attribute before it can be referenced using a fragment identifier. The requirements for the @id attribute differ depending on whether it is used on a topic element, a map element, or an element within a topic or map.

All values for the @id attribute must be XML name tokens.

The @id attributes for topic and map elements are declared as XML attribute type ID; therefore, they must be unique with respect to other XML IDs within the XML document that contains the topic or map element. The @id attribute for most other elements within topics and maps are not declared to be XML IDs; this means that XML parsers do not require that the values of those attributes be unique. However, the DITA specification requires that all IDs be unique within the context of a topic. For this reason, tools might provide an additional layer of validation to flag violations of this rule.

Within documents that contain multiple topics, the values of the @id attribute for all non-topic elements that have the same nearest-ancestor-topic element should be unique with respect to each other. The values of the @id attribute for non-topic elements can be the same as non-topic elements with different nearest-ancestor-topic elements. Therefore, within a single DITA document that contains more than one topic, the values of the @id attribute of the non-topic elements need only to be unique within each topic.

Within a map document, the values of the @id attributes for all elements *SHOULD* be unique. When two elements within a map have the same value for the @id attribute, processors *MUST* resolve references to that ID to the first element with the given ID value in document order.

Element	XML attribute type for @id	Must be unique within	Required?
<map>	ID	document	No
<topic>	ID	document	Yes
sub-map (elements nested within a map)	NMTOKEN	document	Usually no, with some exceptions
sub-topic (elements nested within a topic)	NMTOKEN	individual topic	Usually no, with some exceptions

Figure 8: Summary of requirements for the @id attribute



Note: For all elements other than footnote (<fn>), the presence of a value for the @id attribute has no impact on processing. For <fn>, the presence or absence of a valid @id attribute affects how the element is processed. This is important for tools that automatically assign @id attributes to all elements.

DITA linking

DITA supports many different linking elements, but they all use the same set of attributes to describe relationships between content.

URI-based addressing

URI-based links are described by the following attributes.

@href

The `@href` attribute specifies the URI of the resource that is being addressed.

@format

The `@format` attribute identifies the format of the resource being addressed. For example, references to DITA topics are identified with `format="dita"`, whereas references to DITA maps use `format="ditamap"`. References to other types of content should use other values for this attribute. By default, references to non-XML content use the extension of the URI in the `@href` attribute as the effective format.

@scope

The `@scope` attribute describes the closeness of the relationship between the current document and the target resource. Resources in the same information unit are considered "local"; resources in the same system as the referencing content but not part of the same information unit are considered "peer"; and resources outside the system, such as Web pages, are considered "external".

@type

The `@type` attribute is used on cross-references to describe the target of the reference. Most commonly, the `@type` attribute names the element type being referenced when `format="dita"`.

These four attributes act as a unit, describing whatever link is established by the element that carries them.

The `@format` and `@scope` attributes are assigned default values based on the URI that is specified in the `@href` attribute. Thus they rarely need to be explicitly specified in most cases. However, they can be useful in many non-traditional linking scenarios or environments.

Indirect key-based addressing

DITA also supports indirect links and cross-references in which a DITA map assigns unique names, or keys, to the resources being referenced by the publication. This is done using `<topicref>` elements that specify the `@keys` attribute. Using the `@keyref` attribute, individual links, cross-references, and images then reference resources by their keys instead of their URIs. Links defined using `@keyref` thus allow context-specific linking behavior. That is, the links in a topic or map might resolve to one set of resources in one context, and a completely different set of resources in another, without the need for any modifications to the link markup.

When links are defined using `@keyref`, values for the four linking attributes described above are typically all specified (or given default values) on the key defining element.

URI-based (direct) addressing

Content reference and link relationships can be established from DITA elements by using URI references. DITA uses URI references in `@href`, `@conref`, and other attributes for all direct addressing of resources.

URI references address *resources* and (in some cases) subcomponents of those resources. In this context, a resource is a DITA document (map, topic, or DITA base document) or a non-DITA resource (for example, an image, a Web page, or a PDF document).

URI references that are URLs must conform to the rules for URLs and URIs. Windows paths that contains a backslash (\) are not valid URLs.

URIs and fragment identifiers

For DITA resources, fragment identifiers can be used with the URI to address individual elements. The fragment identifier is the part of the URI that starts with a number sign (#), for example, `#topicid/elementid`. URI references also can include a query component that is introduced with a question mark (?). DITA processors *MAY* ignore queries on URI references to DITA resources. URI references that address components in the same document *MAY* consist of just the fragment identifier.

For addressing DITA elements within maps and topics or individual topics within documents containing multiple topics, URI references must include the appropriate DITA-defined fragment identifier. URI references can be relative or absolute. A relative URI reference can consist of just a fragment identifier. Such a reference is a reference to the document that contains the reference.

Addressing non-DITA targets using a URI

DITA can use URI references to directly address non-DITA resources. Any fragment identifier used must conform to the fragment identifier requirements that are defined for the target media type or provided by processors.

Addressing elements within maps using a URI

When addressing elements within maps, URI references can include a fragment identifier that includes the ID of the map element, for example, `filename.ditamap#mapId` or `#mapId`. The same-topic, URI-reference fragment identifier of a period (.) must not be used in URI references to elements within maps.

Addressing topics using a URI

When addressing a DITA topic element, URI references may include a fragment identifier that includes the ID of the topic element (`filename.dita#topicId` or `#topicId`). When addressing the DITA topic element that contains the URI reference, the URI reference may include the same topic fragment identifier of "." (#.).

Topics always can be addressed by a URI reference whose fragment identifier consists of the topic ID. For the purposes of linking, a reference to a topic-containing document addresses the first topic within that document in document order. For the purposes of rendering, a reference to a topic-containing document addresses the root element of the document.

Consider the following examples:

- Given a document whose root element is a topic, a URI reference (with no fragment identifier) that addresses that document implicitly references the topic element.
- Given a `<dita>` document that contains multiple topics, for the purposes of linking, a URI reference that addresses the `<dita>` document implicitly references the first child topic.
- Given a `<dita>` document that contains multiple topics, for the purposes of rendering, a URI reference that addresses the `<dita>` document implicitly references all the topics that are contained by the `<dita>` element. This means that all the topics that are contained by the `<dita>` element are rendered in the result.

Addressing non-topic elements using a URI

When addressing a non-topic element within a DITA topic, a URI reference must use a fragment identifier that contains the ID of the ancestor topic element of the non-topic element being referenced, a slash ("/"), and the ID of the non-topic element (`filename.dita#topicId/elementId` or `#topicId/elementId`). When addressing a non-topic element within the topic that contains the URI reference, the URI reference can use an abbreviated fragment-identifier syntax that replaces the topic ID with "." (`#./elementId`).

This addressing model makes it possible to reliably address elements that have values for the `@id` attribute that are unique within a single DITA topic, but which might not be unique within a larger XML document that contains multiple DITA topics.

Examples: URI reference syntax

The following table shows the URI syntax for common use cases.

Use case	Sample syntax
Reference a table in a topic at a network location	"http://example.com/file.dita#topicID/tableID"
Reference a section in a topic on a local file system	"directory/file.dita#topicID/sectionID"
Reference a figure contained in the same XML document	"#topicID/figureID"
Reference a figure contained in the same topic of an XML document	"#./figureID"
Reference an element within a map	"http://example.com/map.ditamap#elementID" (and a value of "ditamap" for the <code>@format</code> attribute)
Reference a map element within the same map document	"#elementID" (and a value of "ditamap" for the <code>@format</code> attribute)
Reference an external Web site	"http://www.example.com", "http://www.example.com#somefragment" or any other valid URI
Reference an element within a local map	"filename.ditamap#elementid" (and a value of "ditamap" for the <code>@format</code> attribute)
Reference a local map	"filename.ditamap" (and a value of "ditamap" for the <code>@format</code> attribute)
Reference a local topic	Reference a local topic "filename.dita" or "path/filename.dita"
Reference a specific topic in a local document	"filename.dita#topicid" or "path/filename.dita#topicid"
Reference a specific topic in the same file	"#topicid"
Reference the same topic in the same XML document	"#."

Use case	Sample syntax
Reference a peer map for cross-deliverable linking	". . ./book-b/book-b.ditamap" (and a value of "ditamap" for the @format attribute, a value of "peer" for the @scope attribute, and a value for the @keyscope attribute)

Indirect key-based addressing

DITA keys provide an alternative to direct addressing. The key reference mechanism provides a layer of indirection so that resources (for example, URIs, metadata, or variable text strings) can be defined at the DITA map level instead of locally in each topic.

For information about using keys to define and reference controlled values, see [Subject scheme maps and their usage](#) on page 27.



Note: The material in this section of the DITA specification is exceptionally complex; it is targeted at implementers who build processors and other rendering applications.

Core concepts for working with keys

The concepts described below are critical for a full understanding of keys and key processing.

The use of the phrases "<map> element" or "<topicref> element" should be interpreted as "<map> element and any specialization of <map> element" or "<topicref> element or any specialization of <topicref> element."

Definitions related to keys

resource

For the purposes of keys and key resolution, one of the following:

- An object addressed by URI
- Metadata specified on a resource, such as a @scope or @format attribute
- Text or metadata located within a <topicmeta> element

key

A name for a resource. See [Using keys for addressing](#) on page 58 for more information.

key definition

A <topicref> element that binds one or more key names to zero or more resources.

key reference

An attribute that references a key, such as @keyref or @conkeyref.

key space

A list of key definitions that are used to resolve key references.

effective key definition

The definition for a key within a key space that is used to resolve references to that key. A key might have multiple definitions within a key space, but only one of those definitions is effective.

key scope

A map or section of a map that defines its own key space and serves as the resolution context for its key references.

Key definitions

A key definition binds one or more keys to zero or more resources. Resources can be:

- Any URI-addressed resource that is referenced directly by the @href attribute or indirectly by the @keyref attribute on the key definition. References to the key are considered references to the URI-addressed resource.
- (If the key definition contains a child <topicmeta> element) The child elements of the <topicmeta> element. The content of those elements can be used to populate the content of elements that reference the key.

If a key definition does not contain a <topicmeta> element and does not refer to a resource by @href or @keyref, it is nonetheless a valid key definition. References to the key definition are considered resolvable, but no linking or content transclusion occurs.

Key scopes

All key definitions and key references exist within a key scope. If the @keyscope attribute is never specified within the map hierarchy, all keys exist within a single, default key scope.

Additional key scopes are created when the @keyscope attribute is used. The @keyscope attribute specifies a name or names for the scope. Within a map hierarchy, key scopes are bounded by the following:

- The root map.
- The root element of submaps when the root elements of the submaps specify the @keyscope attribute
- Any <topicref> elements that specify the @keyscope attribute

Key spaces

The key space associated with a key scope is used to resolve all key references that occur immediately within that scope. Key references in child scopes are resolved using the key spaces that are associated with those child scopes.

A key scope is associated with exactly one key space. That key space contains all key definitions that are located directly within the scope; it may also contain definitions that exist in other scopes. Specifically, the key space associated with a key scope is comprised of the following key definitions, in order of precedence:

1. All key definitions from the key space associated with the parent key scope, if any.
2. Key definitions within the scope-defining element, including those defined in directly-addressed, locally-scoped submaps, but excluding those defined in child scopes. (Keys defined in child scopes cannot be addressed without qualifiers.)
3. The key definitions from child scopes, with each key prepended by the child scope name followed by a period. If a child scope has multiple names, the keys in that scope are addressable from the parent scope using any of the scope names as a prefix.



Note: Because of rules 1 and 3, the key space that is associated with a child scope includes the scope-qualified copies of its own keys that are inherited from the key space of the parent scope, as well as those from other "sibling" scopes.

Effective key definitions

A key space can contain many definitions for a given key, but only one definition is effective for the purpose of resolving key references.

When a key has a definition in the key space that is inherited from a parent scope, that definition is effective. Otherwise, a key definition is effective if it is first in a breadth-first traversal of the locally-scoped submaps

beneath the scope-defining element. Put another way, a key definition is effective if it is the first definition for that key name in the shallowest map that contains that key definition. This allows higher-level map authors to override keys defined in referenced submaps.



Note: A key definition that specifies more than one key name in its `@keys` attribute may be the effective definition for some of its keys but not for others.

Within a key scope, keys do not have to be defined before they are referenced. The key space is effective for the entire scope, so the order of key definitions and key references relative to one another is not significant. This has the following implications for processors:

- All key spaces for a root map must be determined before any key reference processing can be performed.
- Maps referenced solely by key reference have no bearing on key space contents.

For purposes of key definition precedence, the scope-qualified key definitions from a child scope are considered to occur at the location of the scope-defining element within the parent scope. See [Example: How key scopes affect key precedence](#) on page 77 for more information.

Key scopes

Key scopes enable map authors to specify different sets of key definitions for different map branches.

A key scope is defined by a `<map>` or `<topicref>` element that specifies the `@keyscope` attribute. The `@keyscope` attribute specifies the names of the scope, separated by spaces.

A key scope includes the following components:

- The scope-defining element
- The elements that are contained by the scope-defining element, minus the elements that are contained by child key scopes
- The elements that are referenced by the scope-defining element or its descendants, minus the elements that are contained by child key scopes

If the `@keyscope` attribute is specified on both a reference to a DITA map and the root element of the referenced map, only one scope is created; the submap does not create another level of scope hierarchy. The single key scope that results from this scenario has multiple names; its names are the union of the values of the `@keyscope` attribute on the map reference and the root element of the submap. This means that processors can resolve references to both the key scopes specified on the map reference and the key scopes specified on the root element of the submap.

The root element of a root map always defines a key scope, regardless of whether a `@keyscope` attribute is present. All key definitions and key references exist within a key scope, even if it is an unnamed, implicit key scope that is defined by the root element in the root map.

Each key scope has its own key space that is used to resolve the key references that occur within the scope. The key space that is associated with a key scope includes all of the key definitions within the key scope. This means that different key scopes can have different effective key definitions:

- A given key can be defined in one scope, but not another.
- A given key also can be defined differently in different key scopes.

Key references in each key scope are resolved using the effective key definition that is specified within its own key scope.

Example: Key scopes specified on both the map reference and the root element of the submap

Consider the following scenario:

```
<map>
  <mapref keyscope="A" href="installation.ditamap"/>
  <!-- ... -->
</map>
```

Figure 9: Root map

```
<map keyscope="B">
  <!-- ... -->
</map>
```

Figure 10: installation.ditamap

Only one key scope is created; it has key scope names of "A" and "B".

Using keys for addressing

For topic references, image references, and other link relationships, resources can be indirectly addressed by using the @keyref attribute. For content reference relationships, resources can be indirectly addressed by using the @conkeyref attribute.

Syntax

For references to topics, maps, and non-DITA resources, the value of the @keyref attribute is simply a key name (for example, keyref="topic-key").

For references to non-topic elements within topics, the value of the @keyref attribute is a key name, a slash ("/"), and the ID of the target element (for example, keyref="topic-key/some-element-id").

Example

For example, consider this topic in the document file.dita:

```
<topic id="topicid">
  <title>Example referenced topic</title>
  <body>
    <section id="section-01">Some content.</section>
  </body>
</topic>
```

and this key definition:

```
<map>
  <topicref keys="myexample"
    href="file.dita"
  />
</map>
```

A cross reference of the form xref="myexample/section-01" resolves to the <section> element in the topic. The key reference is equivalent to the URI reference xref="file.dita#topicid/section-01".

Addressing keys across scopes

When referencing key definitions that are defined in a different key scope, key names might need to be qualified with key scope names.

A root map might contain any number of key scopes; relationships between key scopes are discussed using the following terms:

child scope

A key scope that occurs directly within another key scope. For example, in the figure below, key scopes "A-1" and "A-2" are child scopes of key scope "A".

parent scope

A key scope that occurs one level above another key scope. For example, in the figure below, key scope "A" is a parent scope of key scopes "A-1" and "A-2".

ancestor scope

A key scope that occurs any level above another key scope. For example, in the figure below, key scopes "A" and "Root" are both ancestor scopes of key scopes "A-1" and "A-2"

descendant scope

A key scope that occurs any level below another key scope. For example, in the figure below, key scopes "A", "A-1", and "A-2" are all descendant scopes of the implicit, root key scope

sibling scope

A key scope that shares a common parent with another key scope. For example, in the figure below, key scopes "A" and "B" are sibling scopes; they both are children of the implicit, root key scope.

key scope hierarchy

A key scope and all of its descendant scopes.

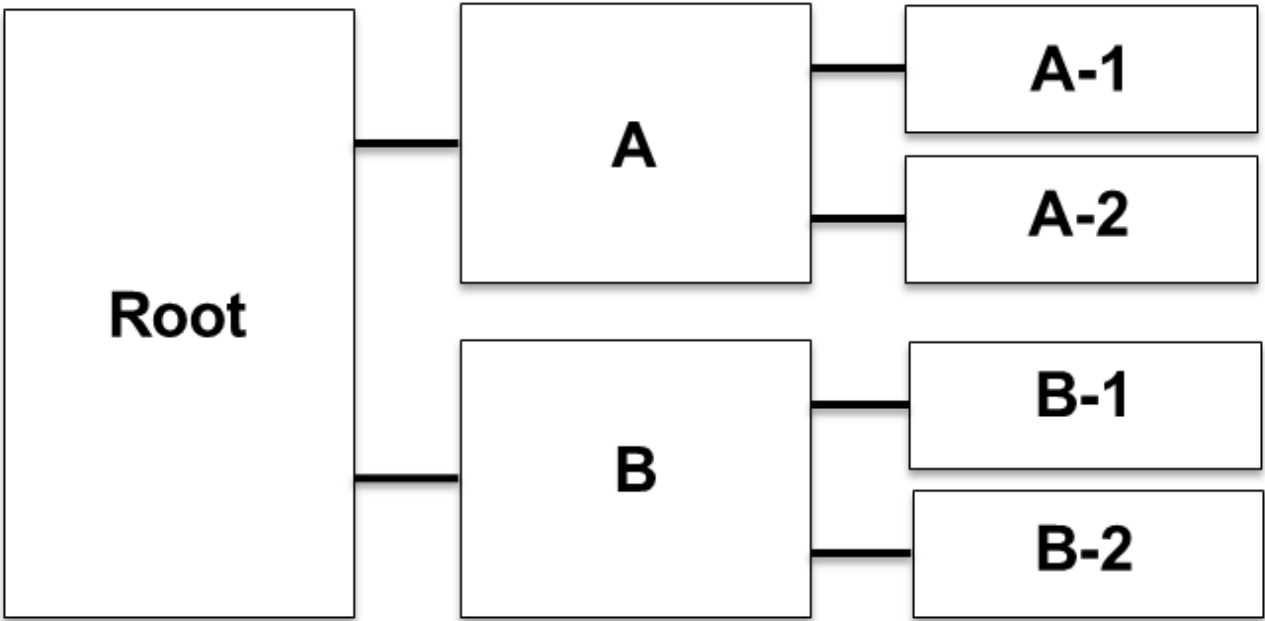


Figure 11: A key scope hierarchy

Keys that are defined in parent key scopes

The key space that is associated with a key scope also includes all key definitions from its parent key scope. If a key name is defined in both a key scope and its parent scope, the key definition in the parent scope takes precedence. This means that a key definition in a parent scope overrides all definitions for the same key name in all descendant scopes. This enables map authors to override the keys that are defined in submaps, regardless of whether the submaps define key scopes.

In certain complex cases, a scope-qualified key name (such as "scope.key") can override an unqualified key name from the parent scope. See [Example: How key scopes affect key precedence](#) on page 77.

Keys that are defined in child key scopes

The key space associated with a key scope does not include the *unqualified* key definitions from the child scopes. However, it does include scope-qualified keys from the child scopes. This enables sibling key scopes to have different key definitions for the same key name.

A *scope-qualified key name* is a key name, prepended by one or more key scope names and separated by periods. For example, to reference a key "keyName" defined in a child scope named "keyScope", specify `keyref="keyScope.keyName"`.

If a key scope has multiple names, its keys can be addressed from its parent scope using any of the scope names. For example, if a key scope is defined with `keyscope="a b c"`, and it contains a key name of "product", that key can be referenced from the parent scope by `keyref="a.product"`, `keyref="b.product"`, or `keyref="c.product"`

Because a child scope contributes its scope-qualified keys to its parent scope, and that parent scope contributes *its* scope-qualified keys to *its* parent scope, it is possible to address the keys in any descendant scope by using the scope-qualified key name. For example, consider a key scope named "ancestorScope" that has a child scope named "parentScope" which in turn has a child scope named "childScope". The scope "childScope" defines a key named "keyName". To reference the key "keyName" from scope "ancestorScope", specify the scope-qualified key name: `keyref="parentScope.childScope.keyName"`.

Keys that are defined in sibling key scopes

Because a parent key scope contains scope-qualified keys from all of its child scopes, and a child scope inherits all of the key definitions (including scope-qualified keys) from its parent scope, it is possible for a child scope to reference its own scope-qualified keys, as well as those defined by its sibling scopes.

For example, consider two sibling scopes, "scope1" and "scope2". Each scope defines the key "productName". References to "productName" in each scope resolve to the local definition. However, since each scope inherits the scope-qualified keys that are available in their parent scope, either scope can reference "scope1.productName" and "scope2.productName" to refer to the scope-specific definitions for that key.

Cross-deliverable addressing and linking

A map can use scoped keys to reference keys that are defined in a different root map. This cross-deliverable addressing can support the production of deliverables that contain working links to other deliverables.

When maps are referenced and the value of the `@scope` attribute is set to "peer", the implications are that the two maps are managed in tandem, and that the author of the referencing map might have access to the referenced map. Adding a key scope to the reference indicates that the peer map should be treated as a separate deliverable for the purposes of linking.

The keys that are defined by the peer map belong to any key scopes that are declared on the `<topicref>` element that references that map. Such keys can be referenced from content in the referencing map by using scope-qualified key names. However, processors handle references to keys that are defined in peer maps differently from how they handle references to keys that are defined in submaps.

DITA processors are not required to resolve key references to peer maps. However, if all resources are available in the same processing or management context, processors have the potential to resolve key references to peer maps. There might be performance, scale, and user interface challenges in implementing such systems, but the ability to resolve any given reference is ensured when the source files are physically accessible.

Note the inverse implication; if the peer map is not available, then it is impossible to resolve the key reference. Processors that resolve key references to peer maps should provide appropriate messages when a reference to a peer map cannot be resolved. Depending on how DITA resources are authored, managed, and processed, references to peer maps might not be resolvable at certain points in the content life cycle.

The peer map might specify `@keyscope` on its root element. In that case, the `@keyscope` on the peer map is ignored for the purpose of resolving scoped key references from the referencing map. This avoids the need for processors to have access to the peer map in order to determine whether a given key definition comes from the peer map.

Example: A root map that declares a peer map

Consider the DITA maps `map-a.ditamap` and `map-b.ditamap`. Map A designates Map B as a peer map by using the following markup:

```
<map>
  <title>Map A</title>
  <topicref
    scope="peer"
    format="ditamap"
    keyscope="map-b"
    href="../map-b/map-b.ditamap"
    processing-role="resource-only"
  />
  <!-- ... -->
</map>
```

In this example, `map-b.ditamap` is not a submap of Map A; it is a peer map.

Example: Key resolution in a peer map that contains a `@keyscope` attribute on the root element

Consider the map reference in map Map A:

```
<mapref
  keyscope="scope-b"
  scope="peer"
  href="map-b.ditamap"
/>
```

where `map-b.ditamap` contains the following markup:

```
<map keyscope="product-x">
  <!-- ... -->
</map>
```

From the context of Map A, key references of the form "scope-b.somekey" are resolved to keys that are defined in the global scope of map B, but key references of the form "product-x.somekey" are not. The presence of a `@keyscope` attribute on the `<map>` element in Map B has no effect. A key reference to the scope "scope-b.somekey" is equivalent to the unscoped reference "somekey" when processed in the context of Map B as the root map. In both cases, the presence of `@keyscope` on the root element of Map B has no effect; in the first case it is explicitly ignored, and in the second case the key reference is within the scope "product-x" and so does not need to be scope qualified.

Processing key references

Key references can resolve as links, as text, or as both. Within a map, they also can be used to create or supplement information on a topic reference. This topic covers information that is common to all key processing, regardless of how the key is used.

Processing of undefined keys

If both `@keyref` and `@href` attributes are specified on an element, the `@href` value *MUST* be used as a fallback address when the key name is undefined. If both `@conkeyref` and `@conref` attributes are specified on an element, the `@conref` value *MUST* be used as a fallback address when the key name is undefined.

Determining effective attributes on the key-referencing element

The attributes that are common to the key-defining element and the key-referencing element, other than the `@keys`, `@processing-role`, and `@id` attributes, are combined as for content references, including the special processing for the `@xml:lang`, `@dir`, and `@translate` attributes. There is no special processing associated with either the `@locktitle` or the `@lockmeta` attributes when attributes are combined.

Keys and conditional processing

The effective key definitions for a key space might be affected by conditional processing (filtering). Processors *SHOULD* perform conditional processing before determining the effective key definitions. However, processors might determine effective key definitions before filtering. Consequently, different processors might produce different effective bindings for the same map when there are key definitions that might be filtered out based on their filtering attributes.



Note: In order to retain backwards compatibility with DITA 1.0 and 1.1, the specification does not mandate a processing order for different DITA features. This makes it technically possible to determine an effective key definition, resolve references to that key definition, and then filter out the definition. However, the preferred approach is to take conditional processing into account when resolving keys, so that key definitions which are excluded by processing are not used in resolving key references.

Reusing a topic in multiple key scopes

If a topic that contains key references is reused in multiple key scopes within a given root map such that its references resolve differently in each use context, processors *MUST* produce multiple copies of the source topic in resolved output for each distinct set of effective key definitions that are referenced by the topic. In such cases, authors can use the `@copy-to` attribute to specify different source URIs for each reference to a topic.

Error conditions

If a referencing element contains a key reference with an undefined key, it is processed as if there were no key reference, and the value of the `@href` attribute is used as the reference. If the `@href` attribute is not specified, the element is not treated as a navigation link. If it is an error for the element to be empty, an implementation *MAY* give an error message; it also *MAY* recover from this error condition by leaving the key reference element empty.

Processing key references for navigation links and images

Keys can be used to create or redirect links and cross references. Keys also can be used to address resources such as images or videos. This topic explains how to evaluate key references on links and cross references to determine a link target.

When a key definition is bound to a resource that is addressed by the `@href` or `@keyref` attributes, and does not specify "none" for the `@linking` attribute, all references to that key definition become links to the bound resource.

When a key definition is not bound to a resource or specifies "none" for the `@linking` attribute, references to that key definition do not become links.

When a key definition has no `@href` value and no `@keyref` value, references to that key will not result in a link, even if they do contain an `@href` attribute of their own. If the key definition also does not contain a `<topicmeta>` subelement, empty elements that refer to the key (such as `<link keyref="a"/>` or `<xref keyref="a" href="fallback.dita"/>`) are ignored.

The `<object>` element has additional key-referencing attributes (`@archivekeyrefs`, `@classidkeyref`, `@codebasekeyref`, and `@datakeyref`). Key names in these attributes are resolved using the same processing that is described for the normal `@keyref` attribute.

Processing key references on `<topicref>` elements

While `<topicref>` elements are used to define keys, they also can reference keys that are defined elsewhere. This topic explains how to evaluate key references on `<topicref>` elements and its specializations.

For topic references that use the `@keyref` attribute, the effective value of the `<topicref>` element is determined in the following way:

Determining the effective resource

The effective resource bound to the `<topicref>` element is determined by resolving all intermediate key references. Each key reference is resolved either to a resource addressed directly by URI reference in an `@href` attribute, or to no resource. Processors *MAY* impose reasonable limits on the number of intermediate key references that they will resolve. Processors *SHOULD* support at least three levels of key references.



Note: This rule applies to all topic references, including those that define keys. The effective bound resource for a key definition that uses the `@keyref` attribute cannot be determined until the key space has been constructed.

Combining metadata

Content from a key-defining element cascades to the key-referencing element following the rules for combining metadata between maps and other maps and between maps and topics. The `@lockmeta` attribute is honored when metadata content is combined.

The combined attributes and content cascade from one map to another or from a map to a topic, but this is controlled by existing rules for cascading, which are not affected by the use of key references.

If, in addition to the `@keys` attribute, a key definition specifies a `@keyref` attribute that can be resolved after the key resolution context for the key definition has been determined, the resources bound to the referenced key definition take precedence.

Processing key references to generate text or link text

Key references can be used to pull text from the key definition. This topic explains how to generate text from a key definition, regardless of whether the key reference also results in a link.



Note: The processing described in this topic is unrelated to the `@conkeyref` attribute. In that case `@conkeyref` is used to determine the target of a `@conref` attribute, after which the normal `@conref` rules apply.

Empty elements that include a key reference with a defined key might get their effective content from the key definition. Empty elements are defined as elements that meet the following criteria:

- Have no text content, including white space

- Have no sub-elements
- Have no attributes that would be used as text content (such as @alt on the <image> element)

When an empty element as defined above references a key definition that has a child <topicmeta> element, content from that <topicmeta> element is used to determine the effective content of the referencing element. Effective content from the key definition becomes the element content, with the following exceptions:

- For empty <image> elements, effective content is used as alternate text, equivalent to creating an <alt> sub-element to hold that content.
- For empty <link> elements, effective content is used as link text, equivalent to creating a <linktext> sub-element to hold that content.
- For empty <link> and <xref> elements, a key definition can be used to provide a short description in addition to the normal effective content. If the key definition includes <shortdesc> inside of <topicmeta>, that <shortdesc> should be used to provide effective content for a <desc> sub-element.
- The <longdescref> and <longquoteref> elements are empty elements with no effective content. Key definitions are not used to set effective text for these elements.
- The <param> element does not have any effective content, so key definitions do not result in any effective content for <param> elements.
- The <indextermref> element is not completely defined, so determining effective content for this element is also left undefined.
- The <abbreviated-form> element is an empty element with special rules that determine its effective content.

Effective text content is determined using the following set of rules:

1. For the <abbreviated-form> element, see the rules described in [abbreviated-form](#)
2. For elements that also exist as a child of <topicmeta> in the key definition, effective content is taken from the first matching direct child of <topicmeta>. For example, given the following key definition, an empty <author> element with the attribute keyref="justMe" would result in the matching content "Just M. Name":

```
<keydef keys="justMe" href="http://www.example.com/my-profile" format="html"
scope="external">
  <topicmeta>
    <author>Just M. Name</author>
  </topicmeta>
</keydef>
```

3. For elements that do not allow the @href attribute, content is taken from the first <keyword> element inside of <keywords> inside of the <topicmeta>. For example, given the following key definition, empty <keyword>, <term>, and <dt> elements with the attribute keyref="nohref" would all result in the matching content "first":

```
<keydef keys="nohref">
  <topicmeta>
    <keywords><keyword>first</keyword><keyword>second</keyword><keyword>third</
keyword></keywords>
  </topicmeta>
</keydef>
```

4. For elements that do allow @href, elements from within <topicmeta> that are legal within the element using @keyref are considered matching text. For example, the <xref> element allows @href, and also allows <keyword> as a child. Using the code sample from the previous item, an empty <xref> with keyref="nohref" would use all three of these elements as text content; after processing, the result would be equivalent to:

```
<xref keyref="test"><keyword>first</keyword><keyword>second</keyword><keyword>third</
keyword></xref>
```


- Otherwise, if `<linktext>` is specified inside of `<topicmeta>`, the contents of `<linktext>` are used as the effective content.



Note: Because all elements that get effective content will eventually look for content in the `<linktext>` element, using `<linktext>` for effective content is a best practice for cases where all elements getting text from a key definition should result in the same value.

- Otherwise, if the element with the key reference results in a link, normal link text determination rules apply as they would for `<xref>` (for example, using the `<navtitle>` or falling back to the URI of the link target).

When the effective content for a key reference element results in invalid elements, those elements *SHOULD* be generalized to produce a valid result. For example, `<linktext>` in the key definition may use a domain specialization of `<keyword>` that is not valid in the key reference context, in which case the specialized element should be generalized to `<keyword>`. If the generalized content is also not valid, a text equivalent should be used instead. For example, `<linktext>` may include `<ph>` or a specialized `<ph>` in the key definition, but neither of those are valid as the effective content for a `<keyword>`. In that case, the text content of the `<ph>` should be used.

Examples of keys

This section of the specification contains examples and scenarios. They illustrate a wide variety of ways that keys can be used.

Examples: Key definition

The `<topicref>` element, and any specialization of `<topicref>` that allows the `@keys` attribute, can be used to define keys.

In the following example, a `<topicref>` element is used to define a key; the `<topicref>` element also contributes to the navigation structure.

```
<map>
  <!--... -->
  <topicref keys="apple-definition" href="apple-gloss-en-US.dita" />
  <!--... -->
</map>
```

The presence of the `@keys` attribute does not affect how the `<topicref>` element is processed.

In the following example, a `<keydef>` element is used to define a key.

```
<map>
  <!--... -->
  <keydef keys="apple-definition" href="apple-gloss-en-US.dita"/>
  <!--... -->
</map>
```

Because the `<keydef>` element sets the default value of the `@processing-role` attribute to "resource-only", the key definition does not contribute to the map navigation structure; it only serves as a key definition for the key name "apple-definition".

Examples: Key definitions for variable text

Key definitions can be used to store variable text, such as product names and user-interface labels. Depending on the key definition, the rendered output might have a link to a related resource.

In the following example, a "product-name" key is defined. The key definition contains a child `<keyword>` element nested within a `<keydef>` element.

```
<map>
  <keydef keys="product-name">
```

```

<topicmeta>
  <keywords>
    <keyword>Thing-O-Matic</keyword>
  </keywords>
</topicmeta>
</keydef>
</map>

```

A topic can reference the "product-name" key by using the following markup:

```

<topic id="topicid">
  <p><keyword keyref="product-name"/> is a product designed to ...</p>
</topic>

```

When processed, the output contains the text "Thing-O-Matic is a product designed to ...".

In the following example, the key definition contains both a reference to a resource and variable text.

```

<map>
  <keydef keys="product-name" href="thing-o-matic.dita">
    <topicmeta>
      <keywords>
        <keyword>Thing-O-Matic</keyword>
      </keywords>
    </topicmeta>
  </keydef>
</map>

```

When processed using the key reference from the first example, the output contains the "Thing-O-Matic is a product designed to ..." text. The phrase "Thing-O-Matic" also is a link to the `thing-o-matic.dita` topic.

Example: Scoped key definitions for variable text

Scoped key definitions can be used for variable text. This enables you to use the same DITA topic multiple times in a DITA map, and in each instance the variable text can resolve differently.

The Acme Tractor Company produces two models of tractor: X and Y. Their product manual contains sets of instructions for each model. While most maintenance procedures are different for each model, the instructions for changing the oil are identical for both model X and model Y. The company policies call for including the specific model number in each topic, so a generic topic that could be used for both models is not permitted.

1. The authoring team references the model information in the `changing-the-oil.dita` topic by using the following mark-up:

```
<keyword keyref="model"/>
```

2. The information architect examines the root map for the manual, and decides how to define key scopes. Originally, the map looked like the following:

```

<map>
  <!-- Model X: Maintenance procedures -->
  <topicref href="model-x-procedures.dita">
    <topicref href="model-x/replacing-a-tire.dita"/>
    <topicref href="model-x/adding-fluid.dita"/>
  </topicref>

  <!-- Model Y: Maintenance procedures -->
  <topicref href="model-y-procedures.dita">
    <topicref href="model-y/replacing-a-tire.dita"/>
    <topicref href="model-y/adding-fluid.dita"/>
  </topicref>
</map>

```

3. The information architect wraps each set of procedures in a `<topicgroup>` element and sets the `@keyscope` attribute.

```
<map>
  <!-- Model X: Maintenance procedures -->
  <topicgroup keyscope="model-x">
    <topicref href="model-x-procedures.dita">
      <topicref href="model-x/replacing-a-tire.dita"/>
      <topicref href="model-x/adding-fluid.dita"/>
    </topicref>
  </topicgroup>

  <!-- Model Y: Maintenance procedures -->
  <topicgroup keyscope="model-y">
    <topicref href="model-y-procedures.dita">
      <topicref href="model-y/replacing-a-tire.dita"/>
      <topicref href="model-y/adding-fluid.dita"/>
    </topicref>
  </topicgroup>
</map>
```

This defines the key scopes for each set of procedures.

4. The information architect then adds key definitions to each set of procedures, as well as a reference to the `changing-the-oil.dita` topic.

```
<map>
  <!-- Model X: Maintenance procedures -->
  <topicgroup keyscope="model-x">
    <keydef keys="model">
      <topicmeta>
        <linktext>X</linktext>
      </topicmeta>
    </keydef>
    <topicref href="model-x-procedures.dita">
      <topicref href="model-x/replacing-a-tire.dita"/>
      <topicref href="model-x/adding-fluid.dita"/>
      <topicref href="common/changing-the-oil.dita"/>
    </topicref>
  </topicgroup>

  <!-- Model Y: Maintenance procedures -->
  <topicgroup keyscope="model-y">
    <keydef keys="model">
      <topicmeta>
        <linktext>Y</linktext>
      </topicmeta>
    </keydef>
    <topicref href="model-y-procedures.dita">
      <topicref href="model-y/replacing-a-tire.dita"/>
      <topicref href="model-y/adding-fluid.dita"/>
      <topicref href="common/changing-the-oil.dita"/>
    </topicref>
  </topicgroup>
</map>
```

When the DITA map is processed, the `changing-the-oil.dita` topic is rendered twice. The `model` variable is rendered differently in each instance, using the text as specified in the scoped key definition. Without key scopes, the first key definition would win, and "model X" would be used in all topics.

Example: Duplicate key definitions within a single map

In this scenario, a DITA map contains duplicate key definitions. How a processor finds the effective key definition depends on document order and the effect of filtering applied to the key definitions.

In the following example, a map contains two definitions for the key "load-toner":

```
<map>
  <!--... -->
  <keydef keys="load-toner" href="model-1235-load-toner-proc.dita"/>
  <keydef keys="load-toner" href="model-4545-load-toner-proc.dita"
  />
  <!--... -->
</map>
```

In this example, only the first key definition (in document order) of the "load-toner" key is effective. All references to the key within the scope of the map resolve to the topic `model-1235-load-toner-proc.dita`.

In the following example, a map contains two definitions for the "file-chooser-dialog" key; each key definition specifies a different value for the `@platform` attribute.

```
<map>
  <!--... -->
  <keydef keys="file-chooser-dialog" href="file-chooser-osx.dita" platform="osx"/>
  <keydef keys="file-chooser-dialog" href="file-chooser-win7.dita" platform="windows7"/>
  <!--... -->
</map>
```

In this case, the effective key definition is determined not only by the order in which the definitions occur, but also by whether the active value of the platform condition is "osx" or "windows7". Both key definitions are *potentially* effective because they have distinct values for the conditional attribute. Note that if no active value is specified for the `@platform` attribute at processing time, then both of the key definitions are present and so the first one in document order is the effective definition.

If the DITAAVAL settings are defined so that both "osx" and "windows" values for the `@platform` attribute are excluded, then neither definition is effective and the key is undefined. That case can be avoided by specifying an unconditional key definition after any conditional key definitions, for example:

```
<map>
  <!--... -->
  <keydef keys="file-chooser-dialog" href="file-chooser-osx.dita" platform="osx"/>
  <keydef keys="file-chooser-dialog" href="file-chooser-win7.dita" platform="windows7"/>
  <keydef keys="file-chooser-dialog" href="file-chooser-generic.dita"/>
  <!--... -->
</map>
```

If the above map is processed with both "osx" and "windows" values for the `@platform` attribute excluded, then the effective key definition for "file-chooser-dialog" is the `file-chooser-generic.dita` resource.

Example: Duplicate key definitions across multiple maps

In this scenario, the root map contains references to two submaps, each of which defines the same key. The effective key definition depends upon the document order of the direct URI references to the maps.

In the following example, a root map contains a key definition for the key "toner-specs" and references to two submaps.

```
<map>
  <keydef keys="toner-specs" href="toner-type-a-specs.dita"/>
  <mapref href="submap-01.ditamap"/>
```

```
<mapref href="submap-02.ditamap"/>
</map>
```

The first submap, submap-01.ditamap, contains definitions for the keys "toner-specs" and "toner-handling":

```
<map>
  <keydef keys="toner-specs" href="toner-type-b-specs.dita"/>
  <keydef keys="toner-handling" href="toner-type-b-handling.dita"/>
</map>
```

The second submap, submap-02.ditamap, contains definitions for the keys "toner-specs", "toner-handling", and "toner-disposal":

```
<map>
  <keydef keys="toner-specs" href="toner-type-c-specs.dita"/>
  <keydef keys="toner-handling" href="toner-type-c-handling.dita"/>
  <keydef keys="toner-disposal" href="toner-type-c-disposal.dita"/>
</map>
```

For this example, the effective key definitions are listed in the following table.

Key	Bound resource
toner-specs	toner-type-a-specs.dita
toner-handling	toner-type-b-handling.dita
toner-disposal	toner-type-c-disposal.dita

The key definition for "toner-specs" in the root map is effective, because it is the first encountered in a breadth-first traversal of the root map. The key definition for "toner-handling" in submap-01.ditamap is effective, because submap-01 is included before submap-02 and so comes first in a breadth-first traversal of the submaps. The key definition for "toner-disposal" is effective because it is the only definition of the key.

Example: Key definition with key reference

When a key definition also specifies a key reference, the key reference must also be resolved in order to determine the effective resources bound to that key definition.

In the following example, a <topicref> element references the key "widget". The definition for "widget" in turn references the key "mainProduct".

```
<map>
  <topicref keyref="widget" id="example"/>
  <keydef keys="widget" href="widgetInfo.dita" scope="local" format="dita" rev="v1r2"
    keyref="mainProduct">
    <topicmeta><navtitle>Information about Widget</navtitle></topicmeta>
  </keydef>
  <keydef keys="mainProduct" href="http://example.com/productPage" scope="external"
    format="html"
    product="prodCode" audience="sysadmin">
    <topicmeta><navtitle>Generic product page</navtitle></topicmeta>
  </keydef>
</map>
```

For this example, the key reference to "widget" pulls resources from that key definition, which in turn pulls resources from "mainProduct". The metadata resources from "mainProduct" are combined with the resources already specified on the "widget" key definition, resulting in the addition of @product and @audience values. Along with the navigation title, the @href, @scope, and @format attributes on the "widget" key definition override

those on "mainProduct". Thus after key references are resolved, the original reference from `<topicref>` is equivalent to:

```
<topicref id="example"
  href="widgetInfo.dita" scope="local" format="dita" rev="v1r2"
  product="prodCode" audience="sysadmin">
  <topicmeta><navtitle>Information about Widget</navtitle></topicmeta>
</topicref>
```

Example: References to scoped keys

You can address scoped keys from outside the key scope in which the keys are defined.

```
<map xml:lang="en">
  <title>Examples of scoped key references</title>

  <!-- Key scope #1 -->
  <topicgroup keyscope="scope-1">
    <keydef keys="key-1" href="topic-1.dita"/>
    <topicref keyref="key-1"/>
    <topicref keyref="scope-1.key-1"/>
    <topicref keyref="scope-2.key-1"/>
  </topicgroup>

  <!-- Key scope #2 -->
  <topicgroup keyscope="scope-2">
    <keydef keys="key-1" href="topic-2.dita"/>
    <topicref keyref="key-1"/>
    <topicref keyref="scope-1.key-1"/>
    <topicref keyref="scope-2.key-1" />
  </topicgroup>

  <topicref keyref="key-1" />
  <topicref keyref="scope-1.key-1" />
  <topicref keyref="scope-2.key-1" />

</map>
```

For this example, the effective key definitions are listed in the following tables.

Table 2: Effective key definitions for scope-1

Key reference	Resource
key-1	topic-1.dita
scope-1.key-1	topic-1.dita
scope-2.key-1	topic-2.dita

Table 3: Effective key definitions for scope-2

Key reference	Resource
key-1	topic-2.dita
scope-1.key-1	topic-1.dita
scope-2.key-1	topic-2.dita

Table 4: Effective key definitions for the key scope associated with the root map

Key reference	Resource
key-1	Undefined
scope-1.key-1	topic-1.dita
scope-2.key-1	topic-2.dita

Example: Key definitions in nested key scopes

In this scenario, the root map contains nested key scopes, each of which contain duplicate key definitions. The effective key definition depends on key-scope precedence rules.

Consider the following DITA map:

```
<map>
  <title>Root map</title>
  <!-- Root scope -->
  <keydef keys="a"/>

  <!-- Key scope A -->
  <topicgroup keyscope="A">
    <keydef keys="b"/>

    <!-- Key scope A-1 -->
    <topicgroup keyscope="A-1">
      <keydef keys="c"/>
    </topicgroup>

    <!-- Key scope A-2 -->
    <topicgroup keyscope="A-2">
      <keydef keys="d"/>
    </topicgroup>
  </topicgroup>

  <!-- Key scope B -->
  <topicgroup keyscope="B">
    <keydef keys="a"/>
    <keydef keys="e"/>

    <!-- Key scope B-1 -->
    <topicgroup keyscope="B-1">
      <keydef keys="f"/>
    </topicgroup>

    <!-- Key scope B-2 -->
    <topicgroup keyscope="B-2">
      <keydef keys="g"/>
    </topicgroup>
  </topicgroup>
</map>
```

The key scopes in this map form a tree structure.

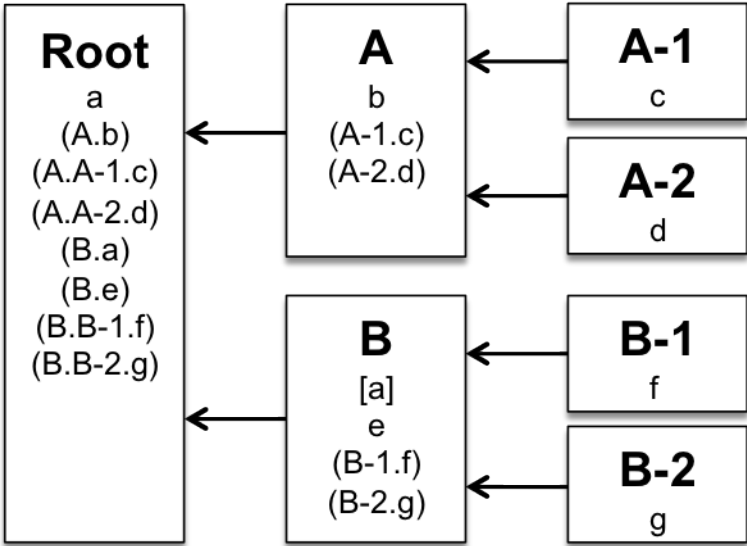


Figure 12: Graphical representation of the key scopes

Each box in the diagram represents a key scope; the name of the key scope is indicated in bold with upper-case letters. Below the name of the key scope, the key definitions that are present in the scope are listed. Different typographic conventions are used to indicate where the key definition occurs:

No styling

The key definition occurs in the immediate key scope and is not overridden by a key definition in a parent scope. For example, key "a" in the root map.

Parentheses

The key definition occurs in a child scope. For example, keys "A-1.c" and "A-2.d" in key scope A.

Brackets

The key definition occurs in the immediate key scope, but it is overridden by a key definition in an ancestor scope. For example, key "a" in key scope B.

Arrows points from child to parent scopes.

Assume that each key scope contains numerous key references. The following tables demonstrate how key references resolve in key scopes A-2 and B. The first column shows the value used in key references; the second column shows the resource to which the key resolves.

Table 5: Key scope A-2

Key reference	Resource to which the key resolves
a	"a", defined in the root map
d	"d", as defined in the immediate key scope
A-2.d	"d", as defined in the immediate key scope
c	Undefined
A-1.c	"A-1.c", as defined in key scope A-1. This key name is available because it exists in the parent scope, key scope A.

Key reference	Resource to which the key resolves
A.A-1.c	"A-1.c", as defined in key scope A-1. This key name is available because it exists in the root key scope.

Table 6: Key scope B

Key reference	Resource to which the key resolves
e	"e", defined in the immediate key scope
a	"a", as defined in the <i>root key scope</i> . (While a key definition for "a" exists in the immediate key scope, it is overridden by the key definition that occurs in the parent key scope.)
B.a	"a", as defined in the <i>immediate key scope</i> . Because the key reference uses the scope-qualified names, it resolves to the key "a" in scope B.
g	Undefined. The key "g" is defined only in key scope B-2, so no unqualified key named "g" is defined in scope B.
B-2.g	"g", as defined in key scope B-2.

Example: Link redirection

This scenario outlines how different authors can redirect links to a common topic by using key definitions. This could apply to `<xref>`, `<link>`, or any elements (such as `<keyword>` or `<term>`) that become navigation links.

A company wants to use a common DITA topic for information about recycling: `recycling.dita`. However, the topic contains a cross-reference to a topic that needs to be unique for each product line; each such topic contains product-specific URLs.

1. The editing team creates a `recycling.dita` topic that includes a cross-reference to the product-specific topic. The cross reference is implemented using a key reference:

```
<xref keyref="product-recycling-info" href="generic-recycling-info.dita"/>
```

The value of the `@href` attribute provides a fallback in the event that a product team forgets to include a key definition for "product-recycling-info".

2. Each product documentation group creates a unique key definition for "product-recycling-info". Each group authors the key definition in a DITA map, for example:

```
<map>
  <!-- ... -->
  <keydef keys="product-recycling-info" href="acme-server-recycling.dita"/>
  <!-- ... -->
</map>
```

Each team can use the `recycling.dita` topic, and the cross reference in the topic resolves differently for each team.

3. A year later, there is an acquisition. The newly-acquired team wants to reuse Acme's common material, but it needs to direct its users to an external Web site that lists the URLs, rather than a topic in the product documentation. Their key definition looks like the following:

```
<topicref keys="product-recycling-info"
          href="http://acme.example.com/server/recycling"
          scope="external" format="html"/>
```

When newly-acquired team uses the `recycling.dita` topic, it resolves to the external Web site; however for all other teams, the cross reference in the topic continues to resolves to their product-specific topic.

4. A new product team is formed, and the team forgets to include a key definition for "product-recycling-info" in one of their root maps. Because the cross reference in the `recycling.dita` topic contains a value for the `@href` attribute, the link falls back to `generic-recycling-info.dita`, thus avoiding a broken cross reference in the output.

Example: Link modification or removal

This scenario outlines how different authors can effectively remove or modify a `<link>` element in a shared topic.

A company wants to use a shared topic for information about customer support. For most products, the shared topic should include a link to a topic about extended warranties. But a small number of products do not offer extended warranties.

1. Team one creates the shared topic: `customer-support.dita`. The topic contains the following mark-up:

```
<related-links>
<link keyref="extended-warranties" href="common/extended-warranties.dita"/>
</related-links>
```

2. The teams that need the link to the topic about extended warranties can reference the `customer-support.dita` topic in their DITA maps. When processed, the related link in the topic resolves to the `common/extended-warranties.dita` topic.
3. The teams that do not want the related link to the topic about extended warranties can include a key definition in their DITA map that does not include an `@href` attribute, for example:

```
<map>
<!-- ... -->
<keydef keys="extended-warranties"/>
<!-- ... -->
</map>
```

When processed, the related link in the topic is not rendered.

4. Yet another team wants to simply have a paragraph about extended warranties printed. They define the key definition for "extended-warranties" as follows:

```
<map>
<!-- ... -->
<keydef keys="extended-warranties"/>
  <topicmeta>
    <linktext>This product does not offer extended warranties.</linktext>
  </topicmeta>
<!-- ... -->
</map>
```

When this team renders their content, there is no hyperlink in the output, just the text "This product does not offer extended warranties" statement.

Example: Links from <term> or <keyword> elements

The @keyref attribute enables authors to specify that references to keywords or terms in a DITA topic can be rendered as a link to an associated resource.

In this scenario, a company with well-developed glossary wants to ensure that instances of a term that is defined in the glossary always include a link to the glossary topic.

1. An information architect adds values for the @keys attribute to all the of the <topicref> elements that are in the DITA map for the glossary, for example:

```
<map>
  <title>Company-wide glossary</title>
  <topicref keys="term-1" href="term-1.dita"/>
  <topicref keys="term-2" href="term-2.dita"/>
  <topicref keys="term-3" href="term-3.dita"/>
  <topicref keys="term-4" href="term-4.dita"/>
</map>
```

2. When authors refer to a term in a topic, they use the following mark-up:

```
<term keyref="term-1"/>
```

When the <term> element is rendered, the content is provided by the <title> element of the glossary topic. The <term> element also is rendered as a link to the glossary topic.

Example: conref redirection

The @conkeyref attribute enables authors to share DITA topics that reuse content. It also enables map authors to specify different key definitions for common keys.

In this scenario, Acme produces content for a product that is also resold through a business partner. When the DITA content is published for the partner, several items must be different, including the following:

- Product names
- Standard notes that contain admonitions

Simply using the @conref attribute would not be possible for teams that use a component content management system where every DITA topic is addressed by a globally-unique identifier (GUID).

1. Authors reference the reusable content in their topics by using the @conkeyref attribute, for example:

```
<task id="reusable-product-content">
  <title><keyword conkeyref="reuse/product-name"/> prerequisites</title>
  <taskbody>
    <prereq><note conkeyref="reuse/warning-1"/></prereq>
    <!-- ... -->
  </taskbody>
</task>
```

2. Authors create two different topics; one topic contains elements appropriate for Acme, and the other topic contains elements appropriate for the partner. Note that each reuse topic must use the same element types (or compatible specializations) and values for the @id attribute. For example, the following reuse file is appropriate for use by Acme:

```
<topic id="acme-reuse">
  <title>Reuse topic for Acme</title>
  <body>
    <note id="warning-1">Admonitions for Acme</note>
    <p><keyword id="product-name">Acme product name</keyword></p>
    <!-- ... -->
  </body>
</topic>
```

```

</body>
</topic>

```

The following reuse file is appropriate for use by the OEM partner:

```

<topic id="oem-reuse">
  <title>Reuse topic for OEM partner</title>
  <body>
    <note id="warning-1">Admonitions for partner</note>
    <p><keyword id="product-name">OEM product name</keyword></p>
    <!-- ... -->
  </body>
</topic>

```

3. The two versions of the DITA maps each contain different key definitions for the key name "reuse". (This associates a key with the topic that contains the appropriate reusable elements.) For example:

```

<map>
  <!-- ... -->
  <keydef keys="reuse" href="acme-reuse.dita"/>
  <!-- ... -->
</map>

```

Figure 13: DITA map for Acme

```

<map>
  <!-- ... -->
  <keydef keys="reuse" href="oem-reuse.dita"/>
  <!-- ... -->
</map>

```

Figure 14: DITA map for OEM partner

When each of the DITA maps is published, the elements that are referenced by @conkeyref will use the reuse topic that is referenced by the <keydef> element in the map. The product names and warnings will be different in the output.

Example: Key scopes and omnibus publications

Key scopes enable you to create omnibus publications that include multiple submaps that define the same key names for common items, such as product names or common topic clusters.

In this scenario, a training organization wants to produce a deliverable that includes all of their training course materials. Each course manual uses common keys for standard parts of the course materials, including "prerequisites," "overview," "assessment", and "summary."

An information architect creates a root map that contains the following markup:

```

<map xml:lang="en">
  <title>Training courses</title>
  <mapref href="course-1.ditamap"/>
  <mapref href="course-2.ditamap"/>
  <mapref href="course-3.ditamap"/>
  <topicref href="omnibus-summary.dita"/>
</map>

```

Each of the submaps contain <topicref> elements that refer to resources using the @keyref attribute. Each submap uses common keys for standard parts of the course materials, including "prerequisites," "overview",

"assessment", and "summary", and their key definitions bind the key names to course-specific resources. For example:

```
<map xml:lang="en">
  <title>Training course #1</title>
  <mapref href="course-1/key-definitions.ditamap"/>
  <topicref keyref="prerequisites"/>
  <topicref keyref="overview"/>
  <topicref keyref="assessment"/>
  <topicref keyref="summary"/>
</map>
```

Without using key scopes, the effective key definitions for the common keys resolve to those found in `course-1.ditamap`. This is not the desired outcome. By adding key scopes to the submaps, however, the information architect can ensure that the key references in the submaps resolve to the course-specific key definitions.

```
<map xml:lang="en">
  <title>Training courses</title>
  <mapref href="course-1.ditamap" keyscope="course-1"/>
  <mapref href="course-2.ditamap" keyscope="course-2"/>
  <mapref href="course-3.ditamap" keyscope="course-3"/>
  <topicref href="omnibus-summary.dita"/>
</map>
```

The information architect does **not** set `keys="summary"` on the `<topicref>` element in the root map. Doing so would mean that all key references to "summary" in the submaps would resolve to `omnibus-summary.dita`, rather than the course-specific summary topics. This is because key definitions located in parent scopes override those located in child scopes.

Example: How key scopes affect key precedence

For purposes of key definition precedence, the scope-qualified key definitions from a child scope are considered to occur at the location of the scope-defining element within the parent scope.

Within a single key scope, key precedence is determined by which key definition comes first in the map, or by the depth of the submap that defines the key. This was true for all key definitions prior to DITA 1.3, because all key definitions were implicitly in the same key scope. Scope-qualified key names differ in that precedence is determined by the location where the key scope is defined.

This distinction is particularly important when key names or key scope names contain periods. While avoiding periods within these names will avoid this sort of issue, such names are legal so processors will need to handle them properly.

The following root map contains one submap and one key definition. The submap defines a key named "sample".

```
<map>
  <!-- The following mapref defines the key scope "scopeName" -->
  <mapref href="submap.ditamap" keyscope="scopeName"/>

  <!-- The following keydef defines the key "scopeName.sample" -->
  <keydef keys="scopeName.sample" href="losing-key.dita"/>

  <!-- Other content, key definitions, etc. -->
</map>
```

Figure 15: Root map

```
<map>
  <keydef keys="sample" href="winning-key.dita"/>
```

```
<!-- Other content, key definitions, etc. -->
</map>
```

Figure 16: Submap

When determining precedence, all keys from the key scope "scopeName" occur at the location of the scope-defining element -- in this case, the `<mapref>` element in the root map. Because the `<mapref>` comes first in the root map, the scope-qualified key name "scopeName.sample" that is pulled from `submap.ditamap` occurs before the definition of "scopeName.sample" in the root map. This means that in the context of the root map, the effective definition of "scopeName.sample" is the scope-qualified key definition that references `winning-key.dita`.

The following illustration shows a root map and several submaps. Each submap defines a new key scope, and each map defines a key. In order to aid understanding, this sample does not use valid DITA markup; instead, it shows the content of submaps inline where they are referenced.

```
<map> <!-- Start of the root map -->

  <mapref href="submapA.ditamap" keyscope="scopeA">
    <!-- Contents of submapA.ditamap begin here -->
    <mapref href="submapB.ditamap" keyscope="scopeB">
      <!-- Contents of submapB.ditamap: define key MYKEY -->
      <keydef keys="MYKEY" href="example-ONE.dita"/>
    </mapref>
    <keydef keys="scopeB.MYKEY" href="example-TWO.dita"/>
    <!-- END contents of submapA.ditamap -->
  </mapref>

  <mapref href="submapC.ditamap" keyscope="scopeA.scopeB">
    <!-- Contents of submapC.ditamap begin here -->
    <keydef keys="MYKEY" href="example-THREE.dita"/>
  </mapref>

  <keydef keys="scopeA.scopeB.MYKEY" href="example-FOUR.dita"/>
</map>
```

Figure 17: Complex map with multiple submaps and scopes

The sample map shows four key definitions. From the context of the root scope, all have key names of "scopeA.scopeB.MYKEY".

1. `submapB.ditamap` defines the key "MYKEY". The key scope "scopeB" is defined on the `<mapref>` to `submapB.ditamap`, so from the context of `submapA.ditamap`, the scope-qualified key name is "scopeB.MYKEY". The key scope "scopeA" is defined on the `<mapref>` to `submapA.ditamap`, so from the context of the root map, the scope-qualified key name is "scopeA.scopeB.MYKEY".
2. `submapA.ditamap` defines the key "scopeB.MYKEY". The key scope "scopeA" is defined on the `<mapref>` to `submapA.ditamap`, so from the context of the root map, the scope-qualified key name is "scopeA.scopeB.MYKEY".
3. `submapC.ditamap` defines the key "MYKEY". The key scope "scopeA.scopeB" is defined on the `<mapref>` to `submapC.ditamap`, so from the context of the root map, the scope-qualified key name is "scopeA.scopeB.MYKEY".
4. Finally, the root map defines the key "scopeA.scopeB.MYKEY".

Because scope-qualified key definitions are considered to occur at the location of the scope-defining element, the effective key definition is the one from `submapB.ditamap` (the definition that references `example-ONE.dita`).

Example: Keys and collaboration

Keys enable authors to collaborate and work with evolving content with a minimum of time spent reworking topic references.

In this scenario, authors collaborate on a publication that includes content for a product that is in the early stages of development. The company documentation is highly-structured and uses the same organization for all publications: "Introduction," "Example," and "Reference."

1. Author one creates a submap for the new product information. She knows the structure that the final content will have, but she does not want to create empty topics for information that is not yet available. She decides to initially author what content is available in a single topic. When more content is available, she'll create additional topics. Her DITA map looks like the following:

```
<map>
  <title>New product content</title>
  <topicref keys="1-overview 1-intro 1-example 1-reference" href="1-overview.dita"/>
</map>
```

2. Author two knows that he needs to add a `<topicref>` to the "Example" topic that will eventually be authored by author one. He references the not-yet-authored topic by key reference:

```
<topicref keyref="1-example"/>
```

His topic reference initially resolves to the `1-overview.dita` topic.

3. Author one finally gets the information that she was waiting on. She creates additional topics and modifies her DITA map as follows:

```
<map>
  <title>New product content</title>
  <topicref keys="1-overview" href="1-overview.dita">
    <topicref keys="1-intro" href="1-intro.dita"/>
    <topicref keys="1-example" href="1-example.dita"/>
    <topicref keys="1-reference" href="1-reference.dita"/>
  </topicref>
</map>
```

Without needing to make any changes to the content, author two's topic reference now resolves to the `1-example.dita` topic.

DITA processing

DITA processing is affected by a number of factors, including attributes that indicate the set of vocabulary and constraint modules on which a DITA document depends; navigation; linking; content reuse (using direct or indirect addressing); conditional processing; branch filtering; chunking; and more. In addition, translation of DITA content is expedited through the use of the `@dir`, `@translate`, and `@xml:lang` attributes, and the `<index-sort-as>` element.

Navigation

DITA includes markup that processors can use to generate reader navigation to or across DITA topics. Such navigation behaviors include table of contents (TOCs) and indexes.

Table of contents

Processors can generate a table of contents (TOC) based on the hierarchy of the elements in a DITA map. By default, each `<topicref>` element in a map represents a node in the TOC. These topic references define a navigation tree.

When a map contains a topic reference to a map (often called a map reference), processors should integrate the navigation tree of the referenced map with the navigation tree of the referencing map at the point of reference. In this way, a deliverable can be compiled from multiple DITA maps.



Note: If a `<topicref>` element that references a map contains child `<topicref>` elements, the processing behavior regarding the child `<topicref>` elements is undefined.

The effective navigation title is used for the value of the TOC node. A TOC node is generated for every `<topicref>` element that references a topic or specifies a navigation title, except in the following cases:

- The `@processing-role` attribute that is specified on the `<topicref>` element or an ancestor element is set to "resource-only".
- Conditional processing is used to filter out the node or an ancestor node.
- The `@print` attribute is specified on the `<topicref>` element or an ancestor element, and the current processing does not match the value set by the `@print` attribute. For example, `print="printonly"` and the output format is XHTML-based, or `print="no"` and the output format is PDF. (Note that the `@print` attribute is deprecated in DITA 1.3; it is replaced by the `@deliveryTarget` attribute.)
- There is no information from which a TOC entry can be constructed; there is no referenced resource or navigation title.
- The node is a `<topicgroup>` element, even if it specifies a navigation title.

To suppress a `<topicref>` element from appearing in the TOC, set the `@toc` attribute to "no". The value of the `@toc` attribute cascades to child `<topicref>` elements, so if `@toc` is set to "no" on a particular `<topicref>`, all children of the `<topicref>` element are also excluded from the TOC. If a child `<topicref>` overrides the cascading operation by specifying `toc="yes"`, then the node that specifies `toc="yes"` appears in the TOC (minus the intermediate nodes that set `@toc` to "no").

Indexes

An index can be generated from index entries that occur in topic bodies, topic prologs, or DITA maps.

The specialized indexing domain also provides elements to enable additional indexing function, such as "See" and "See also".

For more information, see [Indexing group elements](#).

Content reference (conref)

The DITA conref attributes provide mechanisms for reusing content. DITA content references support reuse scenarios that are difficult or impossible to implement using other XML-based inclusion mechanisms like XInclude and entities. Additionally, DITA content references have rules that help ensure that the results of content inclusion remain valid after resolution

Conref overview

The DITA @conref, @conkeyref, @conrefend, and @conaction attributes provide mechanisms for reusing content within DITA topics or maps. These mechanisms can be used both to pull and push content.

This topic uses the definitions of [referenced element](#) on page 0 and [referencing element](#) on page 0 as defined in [DITA terminology and notation](#) on page 5.

Pulling content to the referencing element

When the @conref or @conkeyref attribute is used alone, the referencing element acts as a placeholder for the referenced element, and the content of the referenced element is rendered in place of the referencing element.

The combination of the @conrefend attribute with either @conref or @conkeyref specifies a range of elements that is rendered in place of the referencing element. Although the start and end elements must be of the same type as the referencing element (or specialized from that element type), the elements inside the range can be any type.

Pushing content from the referencing element

The @conaction attribute reverses the direction of reuse from pull to push. With a push, the referencing element is rendered before, after, or in place of the referenced element. The location (before, after, or in place of) is determined by the value of the @conaction attribute. Because the @conaction and @conrefend attributes cannot both be used within the same referencing element, it is not possible to push a range of elements.

A fragment of DITA content, such as an XML document that contains only a single paragraph without a topic ancestor, does not contain enough information for the conref processor to be able to determine the validity of a reference to it. Consequently, the value of a conref must specify one of the following items:

- A referenced element within a DITA map
- A referenced element within a DITA topic
- An entire DITA map
- An entire DITA topic

Related information

[The conaction attribute](#)

[The conkeyref attribute](#)

[The conref attribute](#)

[The conrefend attribute](#)

Processing conrefs

When processing content references, DITA processors compare the restrictions of each context to ensure that the conrefed content is valid in its new context.

Except where allowed by weak constraints, a conref processor *MUST NOT* permit resolution of a reuse relationship that could be rendered invalid under the rules of either the reused or reusing content.



Note: The DITA `@conref` attribute is a transclusion mechanism similar to XInclude and to HyTime value references. DITA differs from these mechanisms, however, in that conref validity does not apply simply to the current content at the time of replacement, but to the possible content given the restrictions of both the referencing document type and the referenced document type.

When pulling content with the conref mechanism, if the referenced element is the same type as the referencing element, and the set of domains declared on the `@domains` attribute in the referenced topic or map instance is the same as or a subset of the domains declared in the referencing document, the element set allowed in the referenced element is guaranteed to be the same as, or a subset of, the element set allowed in the referencing element.

When pushing content with the conref mechanism, the domain checking algorithm is reversed. In this case, if the set of domains declared on the `@domains` attribute in the referencing topic or map instance is the same as or a subset of the domains declared in the referenced document, the element set allowed in the pushed content is guaranteed to be the same as, or a subset of, the element set allowed in the new location.

In both cases, processors resolving conrefs *SHOULD* tolerate specializations of valid elements and generalize elements in the pushed or pulled content fragment as needed for the resolving context.

Related information

[domains attribute rules and syntax](#) on page 122

The `@domains` attribute enables processors to determine whether two elements or two documents use compatible domains. The attribute is declared on the root element for each topic or map type. Each structural, domain, and constraint module defines its ancestry as a parenthesized sequence of space-separated module names; the effective value of the `@domains` attribute is composed of these parenthesized sequences.

Processing attributes when resolving conrefs

When resolving conrefs, processors need to combine the attributes that are specified on the referencing and referenced element.

The attribute specifications on the resolved element are drawn from both the referencing element and the referenced element, according to the following priority:

1. All attributes as specified on the referencing element, except for attributes set to "-dita-use-conref-target".
2. All attributes as specified on the referenced element except the `@id` attribute.
3. The `@xml:lang` attribute has special treatment as described in [The `xml:lang` attribute](#) on page 110.

The token `-dita-use-conref-target` is defined by the specification to enable easier use of `@conref` on elements with required attributes. The only time the resolved element would include an attribute whose specified value is `-dita-use-conref-target` is when the referenced element had that attribute specified with the `-dita-use-conref-target` value and the referencing element either had no specification for that attribute or had it also specified with the `-dita-use-conref-target` value. If the final resolved element (after the complete resolution of any conref chain, as explained below) has an attribute with the `-dita-use-conref-target` value, that element *MUST* be treated as equivalent to having that attribute unspecified.

A given attribute value on the resolved element comes in its entirety from either the referencing element or the referenced element; the attribute values of the referencing and referenced elements for a given attribute are never additive, even if the property (such as `@audience`) takes a list of values.

If the referenced element has a `@conref` attribute specified, the above rules should be applied recursively with the resolved element from one referencing/referenced combination becoming one of the two elements participating in the next referencing/referenced combination. The result should preserve without generalization all elements that are valid in the originating context, even if they are not valid in an intermediate context.

For example, if topic A and topic C allow highlighting, and topic B does not, then a content reference chain of topic A-to-topic B-to-topic C should preserve any highlighting elements in the referenced content. The result, however it is achieved, must be equivalent to the result of resolving the conref pairs recursively starting from the original referencing element in topic A.

Related information

[Using the `-dita-use-conref-target` value](#)

Processing xrefs and conrefs within a conref

When referenced content contains a content reference or cross reference, the effective target of the reference depends on the form of address that is used in the referenced content. It also might depend on the map context, especially when key scopes are present.

Direct URI reference (but not a same-topic fragment identifier)

When the address is a direct URI reference of any form other than a same-topic fragment identifier, processors *MUST* resolve it relative to the source document that contains the original URI reference.

Same-topic fragment identifier

When the address is a same-topic fragment identifier, processors *MUST* resolve it relative to the location of the content reference (referencing context).

Key reference

When the address is a key reference, processors *MUST* resolve it relative to the location of the content reference (referencing context).

When resolving key references or same-topic fragment identifiers, the phrase *location of the content reference* means the final resolved context. For example, in a case where content references are chained (topic A pulls from topic B, which in turn pulls a reference from topic C), the reference is resolved relative to the topic that is rendered. When topic B is rendered, the reference is resolved relative to the content reference in topic B; when topic A is rendered, the reference is resolved relative to topic A. If content is pushed from topic A to topic B to topic C, then the same-topic fragment identifier is resolved in the context of topic C.

The implication is that a content reference or cross reference can resolve to different targets in different use contexts. This is because a URI reference that contains a same-topic fragment identifier is resolved in the context of the topic that contains the content reference, and a key reference is resolved in the context of the key scope that is in effect for each use of the topic that contains the content reference.



Note: In the case of same-topic fragment identifiers, it is the responsibility of the author of the content reference to ensure that any element IDs that are specified in same-topic fragment identifiers in the referenced content will also be available in the referencing topic at resolution time.

Example: Resolving conrefs to elements that contain cross references

Consider the following paragraphs in `paras-01.dita` that are intended to be used by reference from other topics:

```
<topic id="paras-01"><title>Reusable paragraphs</title>
  <body>
    <p id="p1">See <xref href="#paras-01/p5"/>.</p>
    <p id="p2">See <xref href="topic-02.dita#topic02/fig-01"/>.</p>
    <p id="p3">See <xref href="#./p5"/>.</p>
```

```

    <p id="p4">See <xref keyref="task-remove-cover"/>.</p>
    <p id="p5">Paragraph 5 in paras-01.</p>
  </body>
</topic>

```

The paragraphs are used by content reference from other topics, including the using-topic-01.dita topic:

```

<topic id="using-topic-01"><title>Using topic one</title>
  <body>
    <p id="A" conref="paras-01.dita#paras-01/p1"/>
    <p id="B" conref="paras-01.dita#paras-01/p2"/>
    <p id="C" conref="paras-01.dita#paras-01/p3"/>
    <p id="D" conref="paras-01.dita#paras-01/p4"/>
    <p id="p5">Paragraph 5 in using-topic-01</p>
  </body>
</topic>

```

Following resolution of the content references and processing of the <xref> elements in the referenced paragraphs, the rendered cross references in using-topic-01.dita are shown in the following table.

Paragraph	Value of @id attribute on conrefed paragraph	<xref> within conrefed paragraph	Resolution
A	p1	<xref href="#paras-01/p5"/>	The cross reference in paragraph p1 is a direct URI reference that does not contain a same-topic fragment identifier. It can be resolved only to paragraph p5 in paras-01.dita, which contains the content "Paragraph 5 in paras-01".
B	p2	<xref href="topic-02.dita#topic02/fig-01"/>	The cross reference in paragraph p2 is a direct URI reference. It can be resolved only to the element with id="fig-01" in topic-02.dita.
C	p3	<xref href="#./p5"/>	The cross reference in paragraph p3 is a direct URI reference that contains a same-topic fragment identifier. Because the URI reference contains a same-topic fragment identifier, the reference is resolved in the context of the referencing topic (using-topic-01.dita). If using-topic-01.dita did not contain an element with id="p5", then the conref to paragraph p3 would result in a link resolution failure.
D	p4	<xref keyref="task-remove-cover"/>	The cross reference in paragraph p4 is a key reference. It is resolved to whatever resource is bound to the key name "task-remove-cover" in the applicable map context.

Example: Resolving conrefs to elements that contain key-based cross references

Consider the following map, which uses the topics from the previous example:

```
<map>
  <topicgroup keyscope="product-1">
    <topicref keys="task-remove-cover" href="prod-1-task-remove-cover.dita"/>
    <topicref href="using-topic-01.dita"/>
  </topicgroup>
  <topicgroup keyscope="product-2">
    <topicref keys="task-remove-cover" href="prod-2-task-remove-cover.dita"/>
    <topicref href="using-topic-01.dita"/>
  </topicgroup>
</map>
```

The map establishes two key scopes: "product-1" and "product-2". Within the map branches, the key name "task-remove-cover" is bound to a different topic. The topic `using-topic-01.dita`, which includes a conref to a paragraph that includes a cross reference to the key name "task-remove-cover", is also referenced in each branch. When each branch is rendered, the target of the cross reference is different.

In the first branch with the key scope set to "product-1", the cross reference from paragraph p4 is resolved to `prod-1-task-remove-cover.dita`. In the second branch with the key scope set to "product-2", the cross reference from paragraph p4 is resolved to `prod-2-task-remove-cover.dita`.

Conditional processing (profiling)

Conditional processing, also known as profiling, is the filtering or flagging of information based on processing-time criteria.

DITA defines attributes that can be used to enable filtering and flagging individual elements. The `@audience`, `@deliveryTarget`, `@otherprops`, `@platform`, and `@props` attributes (along with specializations of `@props`) allow conditions to be assigned to elements so that the elements can be included, excluded, or flagged during processing. The `@rev` flagging attribute allows values to be assigned to elements so that special formatting can be applied to those elements during processing. A conditional-processing profile specifies which elements to include, exclude, or flag. DITA defines a document type called DITAVAL for creating conditional-processing profiles.

Processors *SHOULD* be able to perform filtering and flagging using the attributes listed above. The `@props` attribute can be specialized to create new attributes, and processors *SHOULD* be able to perform conditional processing on specializations of `@props`.

Although metadata elements exist with similar names, such as the `<audience>` element, processors are not required to perform conditional processing using metadata elements.

Conditional processing values and groups

Conditional processing attributes classify elements with metadata. The metadata is specified using space-delimited string values or grouped values.

For example, the string values within `@product` in `<p product="basic deluxe">` indicate that the paragraph applies to the "basic" product and to the "deluxe" product.

Groups organize classification metadata into subcategories. Groups can be used within `@audience`, `@product`, `@platform`, or `@otherprops`. The following rules apply:

- Groups consist of a name immediately followed by a parenthetical group of one or more space-delimited string values. For example, "groupName (valueOne valueTwo)".
- Groups cannot be nested.
- If two groups with the same name are found in a single attribute, they should be treated as if all values are specified in the same group. The following values for the @otherprops attribute are equivalent:

```
otherprops="groupA(a b) groupA(c) groupZ (APPNAME) "
otherprops="groupA(a b c) groupZ (APPNAME) "
```

- If both grouped values and ungrouped values are found in a single attribute, the ungrouped values belong to an implicit group; the name of that group matches the name of the attribute. Therefore, the following values for the @product attribute are equivalent:

```
product="a database(dbA dbB) b appserver(mySERVER) c"
product="product(a b c) database(dbA dbB) appserver(mySERVER) "
```

Setting a conditional processing attribute to an empty value, such as `product=""`, is equivalent to omitting that attribute from the element. An empty group within an attribute is equivalent to omitting that group from the attribute. For example, `<ph product="database() A">` is equivalent to `<ph product="A">`. Combining both rules into one example, `<ph product="operatingSystem() ">` is equivalent to `<ph>`.

If two groups with the same name exist on different attributes, a rule specified for that group will evaluate the same way on each attribute. For example, if the group "sampleGroup" is specified within both @platform and @otherprops, a DITAVAL rule for `sampleGroup="value"` will evaluate the same in each attribute. If there is a need to distinguish between similarly named groups on different attributes, the best practice is to use more specific group names such as `platformGroupname` and `productGroupname`. Alternatively, DITAVAL rules can be specified based on the attribute name rather than the group name.

If the same group name is used in different attributes, a complex scenario could be created where different defaults are specified for different attributes, with no rule set for a group or individual value. In this case a value might end up evaluating differently in the different attributes. For example, a DITAVAL can set a default of "exclude" for @platform and a default of "flag" for @product. If no rules are specified for group `oddgroup()`, or for the value `oddgroup="edgecase"`, the attribute `platform="oddgroup(edgecase) "` will evaluate to "exclude" while `product="oddgroup(edgecase) "` will resolve to "flag". See [DITAVAL elements](#) for information on how to change default behaviors in DITAVAL provile.



Note: While the grouped values reuse the generalized attribute syntax found in [Attribute generalization](#) on page 130, the @audience, @product, @platform, and @otherprops attributes cannot be specialized or generalized.

Filtering

At processing time, a conditional processing profile can be used to specify profiling attribute values that determine whether an element with those values is included or excluded.

By default, values in conditional processing attributes that are not defined in a DITAVAL profile evaluate to "include". For example, if the value `audience="novice"` is used on a paragraph, but this value is not defined in a DITAVAL profile, the attribute evaluates to "include".

However, the DITAVAL profile can change this default to "exclude", so that any value not explicitly defined in the DITAVAL profile will evaluate to "exclude". The DITAVAL profile also can be used to change the default for a single attribute; for example, it can declare that values in the @platform attribute default to "exclude", while those in the @product attribute default to include. See [DITAVAL elements](#) for information on how to set up a DITAVAL profile and how to change default behaviors.

When deciding whether to include or exclude a particular element, a processor should evaluate each attribute independently:

1. For each attribute:
 - If the attribute value does not contain any groups, then if any token in the attribute value evaluates to "include", the element evaluates to "include"; otherwise it evaluates to "exclude". In other words, the attribute evaluates to "exclude" only when **all** the values in that attribute evaluate to "exclude".
 - If the attribute value does include groups, evaluate as follows, treating ungrouped tokens together as a group:
 1. For each group (including the group of ungrouped tokens), if any token inside the group evaluates to "include", the group evaluates to "include"; otherwise it evaluates to "exclude". In other words, a group evaluates to "exclude" only when every token in that group evaluates to "exclude".
 2. If any group within an attribute evaluates to "exclude", that attribute evaluates to "exclude"; otherwise it evaluates to "include".
2. If **any single attribute** evaluates to exclude, the element is excluded.

For example, if a paragraph applies to three products and the publisher has chosen to exclude all of them, the processor should exclude the paragraph. This is true even if the paragraph applies to an audience or platform that is not excluded. But if the paragraph applies to an additional product that has not been excluded, then its content is still relevant for the intended output and should be preserved.

Similarly, with groups, a step might apply to one application server and two database applications:

```
<steps>
  <step><cmd>Common step</cmd></step>
  <step product="appserver(mySERVER) database(ABC dbOtherName)">
    <cmd>Do something special for databases ABC or OtherName when installing on mySERVER</
  cmd>
  </step>
  <!-- additional steps -->
</steps>
```

If a publisher decides to exclude the application server mySERVER, then the appserver() group evaluates to exclude. This can be done by setting product="mySERVER" to exclude *or* by setting appserver="mySERVER" to exclude. This means the step should be excluded, regardless of how the values "ABC" or "dbOtherName" evaluate. If a rule is specified for both product="mySERVER" and appserver="mySERVER", the rule for the more specific group name "appserver" takes precedence.

Similarly, if both "ABC" and "dbOtherName" evaluate to exclude, then the database() group evaluates to exclude and the element should be excluded regardless of how the "mySERVER" value is set.

In more advanced usage, a DITAVAL can be used to specify a rule for a group name. For example, an author could create a DITAVAL rule that sets product="database" to "exclude". This is equivalent to setting a default of "exclude" for any individual value in a database() group; it also excludes the more common usage of "database" as a single value within the @product attribute. Thus when "myDB" appears in a database() group within the @product attribute, the full precedence for determining whether to include or exclude the value is as follows:

1. Check for a DITAVAL rule for database="myDB"
2. Check for a DITAVAL rule for product="myDB"
3. Check for a DITAVAL rule for product="database" (default for the database group)
4. Check for a DITAVAL rule for "product" (default for the @product attribute)
5. Check for a default rule for all conditions (default of include or exclude for all attributes)
6. Otherwise, evaluate to "include"

Flagging

Flagging is the application of text, images, or styling during rendering. This can highlight the fact that content applies to a specific audience or operating system, for example; it also can draw a reader's attention to content that has been marked as being revised.

At processing time, a conditional processing profile can be used to specify profiling attribute values that determine whether an element with those values is flagged.

When deciding whether to flag a particular element, a processor should evaluate each value. Wherever an attribute value that has been set as flagged appears (for example, `audience="administrator"`), the processor should add the flag. When multiple flags apply to a single element, multiple flags should be rendered, typically in the order that they are encountered.

When the same element evaluates as both flagged and included, the element should be flagged and included. When the same element evaluates as both flagged and filtered (for example, flagged because of a value for the `@audience` attribute and filtered because of a value for the `@product` attribute value), the element should be filtered.

Conditional processing to generate multiple deliverable types

By default, the content of most elements is included in all output media. Within maps and topics, elements can specify the delivery targets to which they apply.

Within maps, topic references can use the `@deliveryTarget` attribute to indicate the delivery targets to which they apply. The map or topic references can still use the deprecated `@print` attribute to indicate whether they are applicable to print deliverables.

Within topics, most elements can use the `@deliveryTarget` attribute to indicate the delivery targets to which they apply.

If a referenced topic should be excluded from all output formats, set the `@processing-role` attribute to "resource-only" instead of using the `@deliveryTarget` or `@print` attribute. Content within that topic can still be referenced for display in other locations.

`@deliveryTarget` attribute

`@deliveryTarget`

The intended delivery target of the content, for example "html", "pdf", or "epub". This attribute is a replacement for the now deprecated `@print` attribute.

The `@deliveryTarget` attribute is specialized from the `@props` attribute. It is defined in the `deliveryTargetAttDomain`, which is integrated into all OASIS-provided document-type shells. If this domain is not integrated into a given document-type shell, the `@deliveryTarget` attribute will not be available.

The `@deliveryTarget` attribute is processed the same way as any other conditional processing attribute. For example, the element `<topicref deliveryTarget="html5 epub" href="example.dita"/>` uses two values for `@deliveryTarget`. A conditional processing profile can then set rules for `@deliveryTarget` that determine whether the topic `example.dita` is included or excluded when the map is rendered as HTML5 or EPUB.

`@print` attribute



Note: Beginning with DITA 1.3, the `@print` attribute is deprecated. Its functionality is superseded by that of the `@deliveryTarget` conditional processing attribute described above.

The @print attribute supports the following enumerated values, each of which control the way that print-oriented processors handle the inclusion or exclusion of topics or groups of topics.

Value	Example	Print-oriented processing	Non-print-oriented processing
Unspecified (default)	<code><topicref href="foo.dita"></code>	Topics are included in output.	Topics are included in output.
yes	<code><topicref href="foo.dita" print="yes"></code>	Topics are included in output.	Topics are included in output.
printonly	<code><topicref href="foo.dita" print="printonly"></code>	Topics are included in output.	Topics are excluded from the output.
no	<code><topicref href="foo.dita" print="no"></code>	Topics are excluded from the output.	Topics are included in output.
-dita-use-conref-target	<code><topicref conref="#foo-topic" print="-dita-use-conref-target"></code>	Topics derive a value for the @print attribute from the @print attribute of the referenced element. See Using the -dita-use-conref-target value for more information.	Topics derive a value for the @print attribute from the @print attribute of the referenced element. See Using the -dita-use-conref-target value for more information.

If a value for the @print is not specified explicitly in a map element, but is specified in a map that references the map element, the @print value cascades to the referenced map.

Examples of conditional processing

This section provides examples that illustrate the ways that conditional processing attributes can be set and used.

Example: Setting conditional processing values and groups

Elements may classify elements with conditional processing attributes using individual values or using groups.

Example: Simple product values

In the following example, the first configuration option applies only to the "extendedprod" product, while the second option applies to both "extendedprod" and to "baseprod". The entire <p> element containing the list applies to an audience of "administrator".

```
<p audience="administrator">Set the configuration options:
  <ul>
    <li product="extendedprod">Set foo to bar</li>
    <li product="basicprod extendedprod">Set your blink rate</li>
    <li>Do some other stuff</li>
    <li>Do a special thing for Linux</li>
  </ul>
</p>
```

Example: Grouped values on an attribute

The following example indicates that a step applies to one application server and two databases. Specifically, this step only applies when it is taken on the server "mySERVER"; likewise, it only applies when used with the databases "ABC" or "dbOtherName".

```
<steps>
  <step><cmd>Common step</cmd></step>
  <step product="appserver(mySERVER) database(ABC dbOtherName)">
    <cmd>Do something special for databases ABC or OtherName when installing on
mySERVER</cmd>
  </step>
  <!-- additional steps -->
</steps>
```

Example: Filtering and flagging topic content

A publisher might want to flag information that applies to administrators, and to exclude information that applies to the extended product.

Consider the following DITA source fragment and conditional processing profile:

```
<p audience="administrator">Set the configuration options:
  <ul>
    <li product="extendedprod">Set foo to bar</li>
    <li product="basicprod extendedprod">Set your blink rate</li>
    <li>Do some other stuff</li>
    <li>Do a special thing for Linux</li>
  </ul>
</p>
```

Figure 18: DITA source fragment

```
<val>
  <prop att="audience" val="administrator" action="flag">
    <startflag><alt-text>ADMIN</alt-text></startflag>
  </prop>
  <prop att="product" val="extendedprod" action="exclude"/>
</val>
```

Figure 19: DITAVAL profile

When the content is rendered, the paragraph is flagged, and the first list item is excluded (since it applies to extendedprod). The second list item is still included; even though it does apply to extendedprod, it also applies to basicprod, which was not excluded.

The result should look something like the following:

ADMIN Set the configuration options:

- Set your blink rate
- Do some other stuff
- Do a special thing for Linux

Branch filtering

The branch filtering mechanism enables map authors to set filtering conditions for specific branches of a map. This makes it possible for multiple conditional-processing profiles to be applied within a single publication.

Without the branch filtering mechanism, the conditions specified in a DITAVAL document are applied globally. With branch filtering, the `<ditavalref>` element specifies a DITAVAL document that can be applied to a subset of content; the location of the `<ditavalref>` element determines the content to which filtering conditions are applied. The filtering conditions then are used to filter the map branch itself (map elements used to create the branch), as well as the local maps or topics that are referenced by that branch.

The `<ditavalref>` element also provides the ability to process a single branch of content multiple times, applying unique conditions to each instance of the branch.

Overview of branch filtering

Maps or map branches can be filtered by adding a `<ditavalref>` element that specifies the DITAVAL document to use for that map or map branch.

The `<ditavalref>` element is designed to manage conditional processing for the following use cases.

1. A map branch needs to be filtered using conditions that do not apply to the rest of the publication. For example, a root map might contain content that is written for both novice and expert users. However, the authors want to add a section that targets only novice users. Using branch filtering, a map branch can be filtered so that it only includes content germane to a novice audience, while the content of the rest of the map remains appropriate for multiple audiences.
2. A map branch needs to be presented differently for different audiences. For example, a set of software documentation might contain installation instructions that differ between operating systems. In that case, the map branch with the installation instructions needs to be filtered multiple times with distinct sets of conditions, while the rest of the map remains common to all operating systems.

Filtering rules often are specified globally, outside of the content. When global conditions set a property value to "exclude", that setting overrides any other settings for the same property that are specified at a branch level. Global conditions that set a conditional property to "include" or "flag" do not override branch-level conditions that set the same property to "exclude".

Using `<ditavalref>` elements, it is possible to specify one set of conditions for a branch and another set of conditions for a subset of the branch. As with global conditions, properties set to "exclude" for a map branch override any other settings for the same property specified for a subset of the branch. Branch conditions that set a conditional property to "include" or "flag" do not override conditions on a subset of the branch that explicitly set the same property to "exclude".

In addition to filtering, applications *MAY* support flagging at the branch level based on conditions that are specified in referenced DITAVAL documents.

Branch filtering: Single condition set for a branch

Using a single `<ditavalref>` element as a child of a map or map branch indicates that the map or map branch must be conditionally processed using the rules specified in the referenced DITAVAL document.

The following rules outline how the filtering conditions that are specified in DITAVAL document are applied:

`<ditavalref>` element as a direct child of a map

The filtering conditions are applied to the entire map.

<ditavalref> element within a map branch

The filtering conditions are used to process the entire branch, including the parent element that contains the <ditavalref> element.

<ditavalref> element within a <topicref> reference to a local map

The filtering conditions are applied to the submap.

<ditavalref> element within a <topicref> reference to peer map

The reference conditions are **not** applied to the peer map.

Branch filtering: Multiple condition sets for a branch

Using multiple <ditavalref> elements as the children of a map or map branch indicates that the map or map branch must be conditionally processed using the rules that are specified in the referenced DITAVAL documents.

When multiple <ditavalref> elements occur as children of the same element, the rules in the referenced DITAVAL documents are processed independently. This effectively requires a processor to maintain one copy of the branch for each <ditavalref>, so that each copy can be filtered using different conditions.



Note: In most cases, it is possible to create a valid, fully-resolved view of a map with branches copied to reflect the different <ditavalref> conditions. However, this might not be the case when multiple <ditavalref> elements occur as direct children of a root map. In this case, it is possible that the map could be filtered in a manner that results in two or more distinct versions of the <title> or metadata. How this is handled is processor dependent. For example, when a root map has three <ditavalref> elements as children of <map>, a conversion to EPUB could produce an EPUB with three versions of the content, or it could produce three distinct EPUB documents.

When a processor maintains multiple copies of a branch for different condition sets, it has to manage situations where a single resource with a single key name results in two distinct resources. Key names must be modified in order to allow references to a specific filtered copy of the resource; without renaming, key references could only be used to refer to a single filtered copy of the resource, chosen by the processor. See [Branch filtering: Impact on resource and key names](#) on page 92 for details on how to manage resource names and key names.

Branch filtering: Impact on resource and key names

When map branches are cloned by a processor in order to support multiple condition sets, processors must manage conflicting resource and key names. The ditavalref domain includes metadata elements that authors can use to specify how resource and key names are renamed.



Note: While the processing controls that are described here are intended primarily for use with map branches that specify multiple condition sets, they also can be used with map branches that include only a single <ditavalref> element.

When a map branch uses multiple condition sets, processors must create multiple effective copies of the branch to support the different conditions. This results in potential conflicts for resource names, key names, and key scopes. Metadata elements inside of the <ditavalref> element are available to provide control over these values, so that keys, key scopes, and URIs can be individually referenced within a branch.

For example, the following map branch specifies two DITAVAL documents:

```
<topicref href="productFeatures.dita" keys="features" keyscope="prodFeatures">
  <ditavalref href="novice.ditaval"/>
  <ditavalref href="admin.ditaval"/>
  <topicref href="newFeature.dita" keys="newThing"/>
</topicref>
```

In this case, the processor has two effective copies of `productFeatures.dita` and `newFeature.dita`. One copy of each topic is filtered using the conditions specified in `novice.ditaval`, and the other copy is filtered using the conditions specified in `admin.ditaval`. If an author has referenced a topic using `keyref="features"` or `keyref="prodFeatures.features"`, there is no way for a processor to distinguish which filtered copy is the intended target.

Metadata elements in the DITAVAL reference domain

Metadata within the `<ditavalref>` element makes it possible to control changes to resource names and key scope names, so that each distinct filtered copy can be referenced in a predictable manner.

`<dvrResourcePrefix>`

Enables a map author to specify a prefix that is added to the start of resource names for each resource in the branch.

`<dvrResourceSuffix>`

Enables a map author to specify a suffix that is added to the end of resource names (before any extension) for each resource in the branch.

`<dvrKeyscopePrefix>`

Enables a map author to specify a prefix that is added to the start of key scope names for each key scope in the branch. If no key scope is specified for the branch, this can be used to establish a new key scope, optionally combined with a value specified in `<dvrKeyscopeSuffix>`.

`<dvrKeyscopeSuffix>`

Enables a map author to specify a suffix that is added to the end of key scope names for each key scope in the branch.

For example, the previous code sample can be modified as follows to create predictable resource names and key scopes for the copy of the branch that is filtered using the conditions that are specified in `admin.ditaval`.

```
<topicref href="productFeatures.dita" keys="features" keyscope="prodFeatures">
  <ditavalref href="novice.ditaval"/>
  <ditavalref href="admin.ditaval">
    <ditavalmeta>
      <dvrResourcePrefix>admin-</dvrResourcePrefix>
      <dvrKeyscopePrefix>adminscope-</dvrKeyscopePrefix>
    </ditavalmeta>
  </ditavalref>
  <topicref href="newFeature.dita" keys="newThing"/>
</topicref>
```

The novice branch does not use any renaming, which allows it to be treated as the default copy of the branch. As a result, when the topics are filtered using the conditions that are specified in `novice.ditaval`, the resource names and key names are unmodified, so that references to the original resource name and key name will resolve to topics in the novice copy of the branch. This has the following effect on topics that are filtered using the conditions specified in `admin.ditaval`:

- The prefix `admin-` is added to the beginning of each resource name in the admin branch.
 - The resource `productFeatures.dita` becomes `admin-productFeatures.dita`
 - The resource `newFeature.dita` becomes `admin-newFeature.dita`
- The prefix `adminscope-` is added to the existing key scope "prodFeatures".
 - The attribute value `keyref="adminscope-prodFeatures.features"` refers explicitly to the admin copy of `productFeatures.dita`

- The attribute `keyref="adminscoop-prodFeatures.newThing"` refers explicitly to the admin copy of `newFeature.dita`



Note: In general, the best way to reference a topic that will be modified based on branch filtering is to use a key rather than a URI. Key scopes and key names (including those modified based on the elements above) must be calculated by processors before other processing. This means that in the example above, a key reference to `adminscoop-prodFeatures.features` will always refer explicitly to the instance of `productFeatures.dita` filtered against the conditions in `admin.ditaval`, regardless of whether a processor has performed the filtering yet. References that use the URI `productFeatures.dita` or `admin-productFeatures.dita` could resolve differently (or fail to resolve), as discussed in [Branch filtering: Implications of processing order](#) on page 95.

Renaming based on multiple `<ditavalref>` elements

It is possible for a branch with `<ditavalref>` already in effect to specify an additional `<ditavalref>`, where each `<ditavalref>` includes renaming metadata. When renaming, metadata on the `<ditavalref>` nested more deeply within the branch appears closer to the original resource or key name. For example:

```
<topicref href="branchParent.dita">
  <ditavalref href="parent.ditaval">
    <ditavalmeta>
      <dvrResourcePrefix>parentPrefix-</dvrResourcePrefix>
    </ditavalmeta>
  </ditavalref>
  <!-- additional topics or layers of nesting -->
  <topicref href="branchChild.dita">
    <ditavalref href="child.ditaval">
      <ditavalmeta>
        <dvrResourcePrefix>childPrefix-</dvrResourcePrefix>
      </ditavalmeta>
    </ditavalref>
  </topicref>
</topicref>
```

In this situation, the resource `branchChild.dita` is given a prefix based on both the reference to `parent.ditaval` and the reference to `child.ditaval`. The value `"childPrefix-"` is specified in the `<ditavalref>` that is nested more deeply within the branch, so it appears closer to the original resource name. The resource `branchChild.dita` would result in `parentPrefix-childPrefix-branchChild.dita`. Suffixes (if specified) would be added in a similar manner, resulting in a name like `branchChild-childSuffix-parentSuffix.dita`. Note that the hyphens are part of the specified prefix; they are not added automatically.

Handling resource name conflicts

It is an error if `<ditavalref>`-driven branch cloning results in multiple copies of a topic that have the same resolved name. Processors *SHOULD* report an error in such cases. Processors *MAY* recover by using an alternate naming scheme for the conflicting topics.

In rare cases, a single topic might appear in different branches that set different conditions, yet still produce the same result. For example, a topic might appear in both the admin and novice copies of a branch but not contain content that is tailored to either audience; in that case, the filtered copies would match. A processor *MAY* consider this form of equivalence when determining if two references to the same resource should be reported as an error.

Branch filtering: Implications of processing order

Because the branch filtering process can result in new or renamed keys, key scopes, or URIs, the full effects of the branch filtering process *MUST* be calculated by processors before they construct the effective map and key scope structure.



Note: The @keyref attribute and related attributes are explicitly disallowed on <ditavalref>. This prevents any confusion resulting from a @keyref that resolves to additional key- or resource-renaming metadata.

In general, the DITA specification refrains from mandating a processing order; thus publication results can vary slightly depending on the order in which processes are run. With branch filtering, processors are not required to apply filter conditions specified outside of the map and filter conditions specified with <ditavalref> at the same time in a publishing process.

For example, a processor might use the following processing order:

1. Apply externally-specified filter conditions to maps
2. Apply filtering based on <ditavalref> elements

Because externally-specified "exclude" conditions always take precedence over branch-specific conditions, content excluded based on external conditions will always be removed, regardless of the order in which processors evaluate conditions.

Processors should consider the following points when determining a processing order:

- If links are generated based on the map hierarchy, those links should be created using the renamed keys and URIs that result from evaluating the <ditavalref> filter conditions, to ensure that the links are consistent within the modified branches. For example, sequential links based on a map hierarchy should remain within the appropriate modified branch.
- If conrefs are resolved in topics before the <ditavalref> filtering conditions are evaluated, content that applies to multiple audiences can be brought in and (later in the process) selectively filtered by branch. For example, if a set of installation steps is pulled in with conref (from outside the branch), it might contain information that is later filtered by platform based on <ditavalref>. This results in copies of the steps that are specific to each operating system. If conref is processed after the <ditavalref>, content might be pulled in that has not been appropriately filtered for the new context.
- The same scenario applies to conref values that push content into the branch.
 - Pushing content into a branch before resolving the <ditavalref> conditions allows content for multiple conditions to be pushed and then filtered by branch based on the <ditavalref> conditions.
 - If the branch using <ditavalref> pushes content elsewhere, resolving <ditavalref> first could result in multiple copies of the content to be pushed (one for each branch), resulting in multiple potentially conflicting copies pushed to the new destination.

Examples of branch filtering

The branch filtering examples illustrate the processing expectations for various scenarios that involve `<ditavalref>` elements. Processing examples use either before and after sample markup or expanded syntax that shows the equivalent markup without the `<ditavalref>` elements.

Example: Single `<ditavalref>` on a branch

A single `<ditavalref>` element can be used to supply filtering conditions for a branch.

Consider the following DITA map and the DITAVAL file that is referenced from the `<ditavalref>` element:

```
<map>
  <topicref href="intro.dita"/>
  <topicref href="install.dita">
    <ditavalref href="novice.ditaval"/>
    <topicref href="do-stuff.dita"/>
    <topicref href="advanced-stuff.dita" audience="admin"/>
    <!-- more topics -->
  </topicref>
  <!-- Several chapters worth of other material -->
</map>
```

Figure 20: input.ditamap:

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

Figure 21: Contents of novice.ditaval

When this content is published, the following processing occurs:

- The first topic (`intro.dita`) does not use any of the conditions that are specified in `novice.ditaval`. It is published normally, potentially using other DITAVAL conditions that are specified externally.
- The second topic (`install.dita`) is filtered using any external conditions as well as the conditions that are specified in `novice.ditaval`.
- The third topic (`do-stuff.dita`) is filtered using any external conditions as well as the conditions that are specified in `novice.ditaval`.
- The fourth topic (`advanced-stuff.dita`) is removed from the map entirely, because it is filtered out with the conditions that are specified for the branch.

In this example, no resources are renamed based on the `<ditavalref>` processing.



Note: In cases where the original resource names map directly to names or anchors in a deliverable, the absence of renaming ensures that external links to those topics are stable regardless of whether a DITAVAL document is used.

Example: Multiple `<ditavalref>` elements on a branch

Multiple `<ditavalref>` elements can be used on a single map branch to create multiple distinct copies of the branch.

Consider the following DITA map that contains a branch with three peer `<ditavalref>` elements. Because topics in the branch are filtered in three different ways, processors are effectively required to handle three copies of the entire branch. Sub-elements within the `<ditavalref>` elements are used to control how new resource names are constructed for two copies of the branch; one copy (based on the conditions in `win.ditaval`) is left with the original file names.

```
<map>
  <topicref href="intro.dita"/>
  <!-- Beginning of installing branch -->
  <topicref href="install.dita">
    <ditavalref href="win.ditaval"/>
    <ditavalref href="mac.ditaval">
      <ditavalmeta>
        <dvrResourceSuffix>-apple</dvrResourceSuffix>
      </ditavalmeta>
    </ditavalref>
    <ditavalref href="linux.ditaval">
      <ditavalmeta>
        <dvrResourceSuffix>-linux</dvrResourceSuffix>
      </ditavalmeta>
    </ditavalref>
    <topicref href="do-stuff.dita">
      <topicref href="mac-specific-stuff.dita" platform="mac"/>
    </topicref>
  <!-- End of installing branch -->
  <topicref href="cleanup.dita"/>
</topicref>
</map>
```

Figure 22: `input.ditamap`

```
<val>
  <prop att="platform" val="win" action="include"/>
  <prop att="platform" action="exclude"/>
</val>
```

Figure 23: Contents of `win.ditaval`

```
<val>
  <prop att="platform" val="mac" action="include"/>
  <prop att="platform" action="exclude"/>
</val>
```

Figure 24: Contents of `mac.ditaval`

```
<val>
  <prop att="platform" val="linux" action="include"/>
  <prop att="platform" action="exclude"/>
</val>
```

Figure 25: Contents of `linux.ditaval`

When a processor evaluates this markup, it results in three copies of the installing branch. The following processing takes place:

- The first topic (`intro.dita`) is published normally, potentially using any other DITAVAL conditions that are specified externally.
- The installing branch appears three times, once for each DITAVAL document. The branches are created as follows:
 - The first branch uses the first DITAVAL document (`win.ditaval`). Resources use their original names as specified in the map. The `mac-specific-stuff.dita` topic is removed. The resulting branch, with indenting to show the hierarchy, matches the original without the `mac` topic:

```
install.dita
  do-stuff.dita
    ...more topics and nested branches...
  cleanup.dita
```

- The second branch uses the second DITAVAL document (`mac.ditaval`). Resources are renamed based on the `<dvrResourceSuffix>` element. The `mac-specific-stuff.dita` topic is included. The resulting branch, with indenting to show the hierarchy, is as follows:

```
install-apple.dita
  do-stuff-apple.dita
    mac-specific-stuff-apple.dita
    ...more topics and nested branches...
  cleanup-apple.dita
```

- The third branch uses the last DITAVAL document (`linux.ditaval`). Resources are renamed based on the `<dvrResourceSuffix>` element. The `mac-specific-stuff.dita` topic is removed. The resulting branch, with indenting to show the hierarchy, is as follows:

```
install-linux.dita
  do-stuff-linux.dita
    ...more topics and nested branches...
  cleanup-linux.dita
```

The example used three DITAVAL documents to avoid triple maintenance of the installing branch in a map; the following map is functionally equivalent, but it requires parallel maintenance of each branch.

```
<map>
  <topicref href="intro.dita"/>
  <!-- Windows installing branch -->
  <topicref href="install.dita">
    <ditavalref href="win.ditaval"/>
    <topicref href="do-stuff.dita">
      <!-- more topics and nested branches -->
    </topicref>
    <topicref href="cleanup.dita"/>
  </topicref>
  <!-- Mac installing branch -->
  <topicref href="install.dita">
    <ditavalref href="mac.ditaval">
      <ditavalmeta><dvrResourceSuffix>-apple</dvrResourceSuffix></ditavalmeta>
    </ditavalref>
    <topicref href="do-stuff.dita">
      <topicref href="mac-specific-stuff.dita" platform="mac"/>
      <!-- more topics and nested branches -->
    </topicref>
    <topicref href="cleanup.dita"/>
  </topicref>
  <!-- Linux installing branch -->
  <topicref href="install.dita">
    <ditavalref href="linux.ditaval">
      <ditavalmeta><dvrResourceSuffix>-linux</dvrResourceSuffix></ditavalmeta>
    </ditavalref>
    <topicref href="do-stuff.dita">
```

```

    <!-- more topics and nested branches -->
  </topicref>
  <topicref href="cleanup.dita"/>
</topicref>
<!-- Several chapters worth of other material -->
</map>

```

Figure 26: input.ditamap

Example: Single `<ditavalref>` as a child of `<map>`

Using a `<ditavalref>` element as a direct child of the `<map>` element is equivalent to setting global filtering conditions for the map.

The following map is equivalent to processing all the contents of the map with the conditions in the `novice.ditaval` document. If additional conditions are provided externally (for example, as a parameter to the publishing process), those conditions take precedence.

```

<map>
  <title>Sample map</title>
  <ditavalref href="novice.ditaval"/>
  <!-- lots of content -->
</map>

```

Example: Single `<ditavalref>` in a reference to a map

Using a `<ditavalref>` element in a reference to a map is equivalent to setting filtering conditions for the referenced map.

In the following example, `other.ditamap` is referenced by a root map. The `<ditavalref>` element indicates that all of the content in `other.ditamap` should be filtered using the conditions specified in the `some.ditaval` document.

```

<topicref href="parent.dita">
  <topicref href="other.ditamap" format="ditamap">
    <ditavalref href="some.ditaval"/>
  </topicref>
</topicref>

```

Figure 27: Map fragment

```

<map>
  <topicref href="nestedTopic1.dita">
    <topicref href="nestedTopic2.dita"/>
  </topicref>
  <topicref href="nestedTopic3.dita"/>
</map>

```

Figure 28: Contents of `other.ditamap`

This markup is functionally equivalent to applying the conditions in `some.ditaval` to the topics that are referenced in the nested map. For the purposes of filtering, it could be rewritten in the following way. The extra

`<topicgroup>` container is used here to ensure filtering is not applied to `parent.dita`, as it would not be in the original example:

```
<topicref href="parent.dita">
  <topicgroup>
    <ditavalref href="some.ditaval"/>
    <topicref href="nestedTopic1.dita">
      <topicref href="nestedTopic2.dita"/>
    </topicref>
    <topicref href="nestedTopic3.dita"/>
  </topicgroup>
</topicref>
```

For the purposes of filtering, this map also could be rewritten as follows.

```
<topicref href="parent.dita">
  <topicref href="nestedTopic1.dita">
    <ditavalref href="some.ditaval"/>
  </topicref>
  <topicref href="nestedTopic2.dita"/>
</topicref>
<topicref href="nestedTopic3.dita">
  <ditavalref href="some.ditaval"/>
</topicref>
</topicref>
```

Filtering based on the `<ditavalref>` element applies to the containing element and its children, so in each case, the files `nestedTopic1.dita`, `nestedTopic2.dita`, and `nestedTopic3.dita` are filtered against the conditions specified in `some.ditaval`. In each version, `parent.dita` is not a parent for the `<ditavalref>`, so it is not filtered.

Example: Multiple `<ditavalref>` elements as children of `<map>` in a root map

Using multiple instances of the `<ditavalref>` element as direct children of the `<map>` element in a root map is equivalent to setting multiple sets of global filtering conditions for the root map.



Note: Unlike most other examples of branch filtering, this example cannot be rewritten using a single valid map with alternate markup that avoids having multiple `<ditavalref>` elements as children of the same grouping element.

Processing the following root map is equivalent to processing all the contents of the map with the conditions in the `mac.ditaval` file and again with the `linux.ditaval` file. If additional conditions are provided externally (for example, as a parameter to the publishing process), those global conditions take precedence.

```
<map>
  <title>Setting up my product
  on <keyword platform="mac">Mac</keyword><keyword platform="linux">Linux</keyword></title>
  <topicmeta>
    <othermeta platform="mac" name="ProductID" content="1234M"/>
    <othermeta platform="linux" name="ProductID" content="1234L"/>
  </topicmeta>
  <ditavalref href="mac.ditaval"/>
  <ditavalref href="linux.ditaval"/>
```

```
<!-- lots of content, including relationship tables -->
</map>
```

Figure 29: input.ditamap

```
<val>
  <prop att="platform" val="mac" action="include"/>
  <prop att="platform" val="linux" action="exclude"/>
</val>
```

Figure 30: Contents of mac.ditaval

```
<val>
  <prop att="platform" val="mac" action="exclude"/>
  <prop att="platform" val="linux" action="include"/>
</val>
```

Figure 31: Contents of linux.ditaval

Because the title and metadata each contain filterable content, processing using the conditions that are referenced by the `<ditavalref>` element results in two variants of the title and common metadata. While this cannot be expressed using valid DITA markup, it is conceptually similar to something like the following.

```
<!-- The following wrapperElement is not a real DITA element.
  It is used here purely as an example to illustrate one possible
  way of picturing the conditions. -->
<wrapperElement>
  <map>
    <title>Setting up my product on <keyword platform="mac">Mac</keyword></title>
    <topicmeta>
      <othermeta platform="mac" name="ProductID" content="1234M"/>
    </topicmeta>
    <ditavalref href="mac.ditaval"/>
    <!-- lots of content, including relationship tables -->
  </map>
  <map>
    <title>Setting up my product on <keyword platform="linux">Linux</keyword></title>
    <topicmeta>
      <othermeta platform="linux" name="ProductID" content="1234L"/>
    </topicmeta>
    <ditavalref href="linux.ditaval"/>
    <!-- lots of content, including relationship tables -->
  </map>
</wrapperElement>
```

How this map is rendered is implementation dependent. If this root map is rendered as a PDF, possible renditions might include the following:

- Two PDFs, with one using the conditions from `mac.ditaval` and another using the conditions from `linux.ditaval`
- One PDF, with a title page that includes each filtered variant of the title and product ID, followed by Mac-specific and Linux-specific renderings of the content as chapters in the PDF
- One PDF, with the first set of filter conditions used to set book level titles and metadata, followed by content filtered with those conditions, followed by content filtered with conditions from the remaining `<ditavalref>` element.

Example: Multiple `<ditavalref>` elements in a reference to a map

Using multiple instances of the `<ditavalref>` element in a reference to a map is equivalent to referencing that map multiple times, with each reference nesting one of the `<ditavalref>` elements.

In the following example, `other.ditamap` is referenced by a root map. The `<ditavalref>` elements provide conflicting sets of filter conditions.

```
<topicref href="parent.dita">
  <topicref href="other.ditamap" format="ditamap">
    <ditavalref href="audienceA.ditaval"/>
    <ditavalref href="audienceB.ditaval"/>
    <ditavalref href="audienceC.ditaval"/>
  </topicref>
</topicref>
```

Figure 32: Map fragment

This markup is functionally equivalent to referencing `other.ditamap` three times, with each reference including a single `<ditavalref>` elements. The fragment could be rewritten as:

```
<topicref href="parent.dita">
  <topicref href="other.ditamap" format="ditamap">
    <ditavalref href="audienceA.ditaval"/>
  </topicref>
  <topicref href="other.ditamap" format="ditamap">
    <ditavalref href="audienceB.ditaval"/>
  </topicref>
  <topicref href="other.ditamap" format="ditamap">
    <ditavalref href="audienceC.ditaval"/>
  </topicref>
</topicref>
```

Figure 33: Map fragment

Example: `<ditavalref>` within a branch that already uses `<ditavalref>`

When a branch is filtered because a `<ditavalref>` element is present, another `<ditavalref>` deeper within that branch can supply additional conditions for a subset of the branch.

In the following map fragment, a set of operating system conditions applies to installation instructions. Within that common branch, a subset of content applies to different audiences.

```
<topicref href="install.dita">
  <ditavalref href="linux.ditaval"/>
  <ditavalref href="mac.ditaval">
    <ditavalmeta>
      <dvrResourceSuffix>-mac</dvrResourceSuffix>
    </ditavalmeta>
  </ditavalref>
  <ditavalref href="win.ditaval">
    <ditavalmeta>
      <dvrResourceSuffix>-win</dvrResourceSuffix>
    </ditavalmeta>
  </ditavalref>
  <topicref href="perform-install.dita">
    <!-- other topics-->
  </topicref>
  <!-- Begin configuration sub-branch -->
  <topicref href="configure.dita">
```

```

<ditavalref href="novice.ditaval">
  <ditavalmeta>
    <dvrResourceSuffix>-novice</dvrResourceSuffix>
  </ditavalmeta>
</ditavalref>
<ditavalref href="advanced.ditaval">
  <ditavalmeta>
    <dvrResourceSuffix>-admin</dvrResourceSuffix>
  </ditavalmeta>
</ditavalref>
<!-- Other config topics -->
</topicref>
<!-- End configuration sub-branch -->
</topicref>

```

In this case, the effective map contains three copies of the complete branch. The branches are filtered by operating system. Because topics in the branch are filtered in different ways, processors are effectively required to handle three copies of the entire branch. The map author uses the `<dvrResourceSuffix>` elements to control naming for each copy. The Linux branch does not specify a `<dvrResourceSuffix>` element, because it is the default copy of the branch; this allows documents such as `install.dita` to retain their original names.

Within each operating system instance, the configuration sub-branch is repeated; it is filtered once for novice users and then again for advanced users. As a result, there are actually six instances of the configuration sub-branch. Additional `<dvrResourceSuffix>` elements are used to control naming for each instance.

1. The first instance is filtered using the conditions in `linux.ditaval` and `novice.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-novice.dita`. There is no renaming based on `linux.ditaval`, and the `<ditavalref>` that references `novice.ditaval` adds the suffix `-novice`.
2. The second instance is filtered using the conditions in `linux.ditaval` and `advanced.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-admin.dita`. There is no renaming based on `linux.ditaval`, and the `<ditavalref>` that references `advanced.ditaval` adds the suffix `-admin`.
3. The third instance is filtered using the conditions in `mac.ditaval` and `novice.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-novice-mac.dita`. The `<ditavalref>` that references `novice.ditaval` adds the suffix `-novice`, resulting in `configure-novice.dita`, and then the `<ditavalref>` that references `mac.ditaval` adds the additional suffix `-mac`.
4. The fourth instance is filtered using the conditions in `mac.ditaval` and `advanced.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-admin-mac.dita`. The `<ditavalref>` that references `admin.ditaval` adds the suffix `-admin`, resulting in `configure-admin.dita`, and then the `<ditavalref>` that references `mac.ditaval` adds the additional suffix `-mac`.
5. The fifth instance is filtered using the conditions in `win.ditaval` and `novice.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-novice-win.dita`. The `<ditavalref>` that references `novice.ditaval` adds the suffix `-novice`, resulting in `configure-novice.dita`, and then the `<ditavalref>` that references `win.ditaval` adds the additional suffix `-win`.
6. The sixth instance is filtered using the conditions in `win.ditaval` and `advanced.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-admin-win.dita`. The `<ditavalref>` that references `admin.ditaval` adds the suffix `-admin`, resulting in `configure-admin.dita`, and then the `<ditavalref>` that references `win.ditaval` adds the additional suffix `-win`.

Example: <ditavalref> error conditions

It is an error condition when multiple, non-equivalent copies of the same file are created with the same resource name.

The following map fragment contains several error conditions that result in name clashes:

```
<topicref href="a.dita" keys="a">
  <ditavalref href="one.ditaval"/>
  <ditavalref href="two.ditaval"/>
  <topicref href="b.dita" keys="b"/>
</topicref>
<topicref href="a.dita"/>
<topicref href="c.dita" keys="c">
  <ditavalref href="one.ditaval">
    <ditavalmeta>
      <dvrResourceSuffix>-token</dvrResourceSuffix>
    </ditavalmeta>
  </ditavalref>
  <ditavalref href="two.ditaval">
    <ditavalmeta>
      <dvrResourceSuffix>-token</dvrResourceSuffix>
    </ditavalmeta>
  </ditavalref>
</topicref>
```

In this sample, the effective map that results from evaluating the filter conditions has several clashes. In some cases the same document must be processed with conflicting conditions, using the same URI. In addition, because no key scope is added or modified, keys in the branch are duplicated in such a way that only one version is available for use. When the branches are evaluated to create distinct copies, the filtered branches result in the following equivalent map:

```
<topicref href="a.dita" keys="a"> <!-- a.dita to be filtered by one.ditaval -->
  <topicref href="b.dita" keys="b"/> <!-- b.dita to be filtered by one.ditaval -->
</topicref>
<topicref href="a.dita" keys="a"> <!-- a.dita to be filtered by two.ditaval; key "a"
ignored -->
  <topicref href="b.dita" keys="b"/> <!-- b.dita to be filtered by two.ditaval; key "b"
ignored -->
</topicref>
<topicref href="a.dita"/>
<topicref href="c-token.dita" keys="c">
  <!-- c-token.ditaval to be filtered by one.ditaval -->
</topicref>
<topicref href="c-token.dita" keys="c">
  <!-- c-token.ditaval to be filtered by two.ditaval, key "c" ignored -->
</topicref>
```

The equivalent map highlights several problems with the original source:

- The key names "a" and "b" are present in a branch that will be duplicated. No key scope is introduced for either version of the branch, meaning that the keys will be duplicated. Because there can only be one effective key definition for "a" or "b", it only is possible to reference one version of the topic using keys.
- The key name "c" is present on another branch that will be duplicated, resulting in the same problem.
- The file `c.dita` is filtered with two sets of conditions, each of which explicitly maps the filtered resource to `c-token.dita`. This is an error condition that should be reported by processors.
- In situations where resource names map directly to output file names, such as an HTML5 rendering that creates files based on the original resource name, the following name conflicts also occur. In this case a processor would need to report an error, use an alternate naming scheme, or both:

1. a.dita generates a.html using three alternate set of conditions. One version uses one.ditaval, one version uses two.ditaval, and the third version uses no filtering.
2. b.dita generates b.html using two alternate set of conditions. One version uses one.ditaval, and the other version uses two.ditaval.

Chunking

Content can be chunked (divided or merged into new output documents) in different ways for the purposes of delivering content and navigation. For example, content best authored as a set of separate topics might need to be delivered as a single Web page. A map author can use the @chunk attribute to split up multi-topic documents into component topics or to combine multiple topics into a single document as part of output processing.

The @chunk attribute is commonly used for the following use cases.

Reuse of a nested topic

A content provider creates a set of topics as a single document. Another user wants to incorporate only one of the nested topics from the document. The new user can reference the nested topic from a DITA map, using the @chunk attribute to specify that the topic should be produced in its own document.

Identification of a set of topics as a unit

A curriculum developer wants to compose a lesson for a SCORM LMS (Learning Management System) from a set of topics without constraining reuse of those topics. The LMS can save and restore the learner's progress through the lesson if the lesson is identified as a referenceable unit. The curriculum developer defines the collection of topics with a DITA map, using the @chunk attribute to identify the learning module as a unit before generating the SCORM manifest.

Using the @chunk attribute

The specification defines tokens for the @chunk attribute that cover the most common chunking scenarios. These tokens may be used to override whatever default chunking behavior is set by a processor. Chunking is necessarily format specific, with chunked output required for some and not supported for other rendered formats. Chunking is also implementation specific with some implementations supporting some, but not all, chunking methods, or adding new methods to the standard ones described in this specification.

The value of the @chunk attribute consists of one or more space delimited tokens. Tokens are defined in three categories: for selecting topics, for setting chunking policies, and for defining how the chunk values impact rendering. It is an error to use two tokens from the same category on a single <topicref> element.

Selecting topics

When addressing a document that contains multiple topics, these values determine which topics are selected. These values are ignored when the element on which they are specified does not reference a topic. The defined values are:

- **select-topic:** Selects an individual topic without any ancestors, descendents, or peers from within the same document.
- **select-document:** Selects all topics in the target document.
- **select-branch:** Selects the target topic together with its descendent topics.

Policies for splitting or combining documents

The chunking policy tokens determine how source topics are chunked to produce different output chunks, for output formats where that makes sense. When specified on a `<map>` element, the policy becomes the default policy for all topic references. When specified on a topic reference, the policy applies only to that `<topicref>` and not to any descendant `<topicref>` elements.

- **by-topic:** A separate output chunk is produced for each of the selected topics. In particular, topics that are part of multi-topic documents are processed as though they are the root topics within a separate XML document.
- **by-document:** A single output chunk is produced for the referenced topic or topics, as though the selected topics were all children of the same document.

Rendering the selection

The following tokens affect how the chunk values impact rendering of the map or topics.

- **to-content:** The selection should be rendered as a new chunk of content.
 - When specified on a `<topicref>`, this means that the topics selected by this `<topicref>` and its children will be rendered as a single chunk of content.
 - When specified on the `<map>` element, this indicates that the contents of all topics referenced by the map are to be rendered as a single document.
 - When specified on a `<topicref>` that contains a title but no target, this indicates that processors *MUST* generate a title-only topic in the rendered result, along with any topics referenced by child `<topicref>` elements of this `<topicref>`. The rendition address of the generated topic is determined as defined for the `@copy-to` attribute. If the `@copy-to` attribute is not specified and the `<topicref>` has no `@id` attribute, the address of the generated topic is not required to be predictable or consistent across rendition instances.

For cross references to `<topicref>` elements, if the value of the `@chunk` attribute is "to-content" or is unspecified, the cross reference is treated as a reference to the target topic. If the reference is to a `<topicref>` with no target, it is treated as a reference to the generated title-only topic.

- **to-navigation (DEPRECATED):** The "to-navigation" token is deprecated in DITA 1.3. In earlier releases, the "to-navigation" token indicates that a new chunk of navigation should be used to render the current selection (such as an individual Table of Contents or related links). When specified on the `<map>` element, this token indicates that the map should be presented as a single navigation chunk. If a cross reference is made to a `<topicref>` that has a title but no target, and the `@chunk` value of that `<topicref>` is set to "to-navigation", the resulting cross reference is treated as a reference to the rendered navigation document (such as an entry in the table of contents).

Some tokens or combinations of tokens might not be appropriate for all output types. When unsupported or conflicting tokens are encountered during output processing, processors *SHOULD* produce warning or error messages. Recovery from such conflicts or other errors is implementation dependent.

There is no default value for the `@chunk` attribute on most elements and the `@chunk` attribute does not cascade from container elements, meaning that the `@chunk` value on one `<topicref>` is not passed to its children. A default by-xxx policy for an entire map may be established by setting the `@chunk` attribute on the `<map>` element, which will apply to any `<topicref>` that does not specify its own by-xxx policy.

When no `@chunk` attribute values are specified or defaulted, chunking behavior is implementation dependent. When variations of this sort are not desired, a default for a specific map can be established by including a `@chunk` attribute value on the `<map>` element.

When chunk processing results in new documents, the resource name or identifier for the new document (if relevant) is determined as follows:

1. If an entire map is used to generate a single chunk (by placing to-content on the `<map>` element), the resource name *SHOULD* be taken from the resource name of the map.

2. If the @copy-to attribute is specified, the resource name *MUST* taken from the @copy-to attribute.
3. If the @copy-to attribute is not specified and one or more keys are specified on the <topicref>, the resource name *SHOULD* be constructed using one of the keys.
4. If @copy-to and @keys are not specified and the by-topic policy is in effect, the resource name *SHOULD* be taken from the @id attribute of the topic.
5. If @copy-to and @keys are not specified and the by-document policy is in effect, the resource name *SHOULD* be taken from the resource name of the referenced document.

When following these steps results in resource name clashes, processors *MAY* recover by generating alternate resource identifiers. For example, when two chunked topics use the same @id attribute, a processor could recover by combining the original resource name with the @id value instead of using only the @id value.

Implementation-specific tokens and future considerations

Implementers *MAY* define their own custom, implementation-specific tokens. To avoid name conflicts between implementations or with future additions to the standard, implementation-specific tokens *SHOULD* consist of a prefix that gives the name or an abbreviation for the implementation followed by a colon followed by the token or method name.

For example: "acme:level2" could be a token for the Acme DITA Toolkit that requests the "level2" chunking method.

Chunking examples

The following examples cover many common chunking scenarios, such as splitting one document into many rendered objects or merging many documents into one rendered object.

In the examples below, an extension of ".xxxx" is used in place of the actual extensions that will vary by output format. For example, when the output format is HTML, the extension may actually be ".html", but this is not required.

The examples below assume the existence of the following files:

- parent1.dita, parent2.dita, etc., each containing a single topic with id P1, P2, etc.
- child1.dita, child2.dita, etc., each containing a single topic with id C1, C2, etc.
- grandchild1.dita, grandchild2.dita, etc., each containing a single topic with id GC1, GC2, etc.
- nested1.dita, nested2.dita, etc., each containing two topics: parent topics with id N1, N2, etc., and child topics with ids N1a, N2a, etc.
- ditabase.dita, with the following contents:

```
<dita xml:lang="en-us">
  <topic id="X">
    <title>Topic X</title><body><p>content</p></body>
  </topic>
  <topic id="Y">
    <title>Topic Y</title><body><p>content</p></body>
    <topic id="Y1">
      <title>Topic Y1</title><body><p>content</p></body>
      <topic id="Y1a">
        <title>Topic Y1a</title><body><p>content</p></body>
      </topic>
    </topic>
    <topic id="Y2">
      <title>Topic Y2</title><body><p>content</p></body>
    </topic>
  </topic>
  <topic id="Z">
    <title>Topic Z</title><body><p>content</p></body>
  </topic>
</dita>
```

```

    <topic id="Z1">
      <title>Topic Z1</title><body><p>content</p></body>
    </topic>
  </dita>

```

1. The following map causes the entire map to generate a single output chunk.

```

<map chunk="to-content">
  <topicref href="parent1.dita">
    <topicref href="child1.dita"/>
    <topicref href="child2.dita"/>
  </topicref>
</map>

```

2. The following map will generate a separate chunk for every topic in every document referenced by the map. In this case, it will result in the topics P1 . xxxx, N1 . xxxx, and N1a . xxxx.

```

<map chunk="by-topic">
  <topicref href="parent1.dita">
    <topicref href="nested1.dita"/>
  </topicref>
</map>

```

3. The following map will generate two chunks: parent1 . xxxx will contain only topic P1, while child1 . xxxx will contain topic C1, with topics GC1 and GC2 nested within C1.

```

<map>
  <topicref href="parent1.dita">
    <topicref href="child1.dita" chunk="to-content">
      <topicref href="grandchild1.dita"/>
      <topicref href="grandchild2.dita"/>
    </topicref>
  </topicref>
</map>

```

4. The following map breaks down portions of database . dita into three chunks. The first chunk Y . xxxx will contain only the single topic Y. The second chunk Y1 . xxxx will contain the topic Y1 along with its child Y1a. The final chunk Y2 . xxxx will contain only the topic Y2. For navigation purposes, the chunks for Y1 and Y2 are still nested within the chunk for Y.

```

<map>
  <topicref href="database.dita#Y" copy-to="Y.dita"
    chunk="to-content select-topic">
    <topicref href="database.dita#Y1" copy-to="Y1.dita"
      chunk="to-content select-branch"/>
    <topicref href="database.dita#Y2" copy-to="Y2.dita"
      chunk="to-content select-topic"/>
  </topicref>
</map>

```

5. The following map will produce a single output chunk named parent1 . xxxx, containing topic P1, with topic Y1 nested within P1, but without topic Y1a.

```

<map chunk="by-document">
  <topicref href="parent1.dita" chunk="to-content">
    <topicref href="database.dita#Y1"
      chunk="select-topic"/>
  </topicref>
</map>

```

6. The following map will produce a single output chunk, `parent1.xxxx`, containing topic P1, topic Y1 nested within P1, and topic Y1a nested within Y1.

```
<map chunk="by-document">
  <topicref href="parent1.dita" chunk="to-content">
    <topicref href="database.dita#Y1"
      chunk="select-branch"/>
  </topicref>
</map>
```

7. The following map will produce a single output chunk, `P1.xxxx`. The topic P1 will be the root topic, and topics X, Y, and Z (together with their descendents) will be nested within topic P1.

```
<map chunk="by-topic">
  <topicref href="parent1.dita" chunk="to-content">
    <topicref href="database.dita#Y1"
      chunk="select-document"/>
  </topicref>
</map>
```

8. The following map will produce a single output chunk named `parentchunk.xxxx` containing topic P1 at the root. Topic N1 will be nested within P1, and N1a will be nested within N1.

```
<map chunk="by-document">
  <topicref href="parent1.dita" chunk="to-content" copy-to="parentchunk.dita">
    <topicref href="nested1.dita" chunk="select-branch"/>
  </topicref>
</map>
```

9. The following map will produce two output chunks. The first chunk named `parentchunk.xxxx` will contain the topics P1, C1, C3, and GC3. The "to-content" token on the reference to `child2.dita` causes that branch to begin a new chunk named `child2chunk.xxxx`, which will contain topics C2 and GC2.

```
<map chunk="by-document">
  <topicref href="parent1.dita"
    chunk="to-content" copy-to="parentchunk.dita">
    <topicref href="child1.dita" chunk="select-branch"/>
    <topicref href="child2.dita"
      chunk="to-content select-branch"
      copy-to="child2chunk.dita">
      <topicref href="grandchild2.dita"/>
    </topicref>
    <topicref href="child3.dita">
      <topicref href="grandchild3.dita"
        chunk="select-branch"/>
    </topicref>
  </topicref>
</map>
```

10. The following map produces a single chunk named `nestedchunk.xxxx`, which contains topic N1 with no topics nested within.

```
<map>
  <topicref href="nested1.dita#N1"
    copy-to="nestedchunk.dita"
    chunk="to-content select-topic"/>
</map>
```

11. In DITA 1.3, the "to-navigation" chunk is deprecated. In earlier releases, the following map produced two navigation chunks, one for P1, C1, and the other topic references nested under `parent1.dita`, and a second for P2, C2, and the other topic references nested under `parent2.dita`.

```
<map>
  <topicref href="parent1.dita"
```

```

        navtitle="How to set up a web server"
        chunk="to-navigation">
<topicref href="child1.dita"
        chunk="select-branch"/>
<!-- ... -->
</topicref>
<topicref href="parent2.dita"
        navtitle="How to ensure database security"
        chunk="to-navigation">
<topicref href="child2.dita"
        chunk="select-branch"/>
<!-- ... -->
</topicref>
<!-- ... -->
</map>

```

Translation and localization

DITA has features that facilitate preparing content for translation and working with multilingual content, including the `@xml:lang` attribute, the `@dir` attribute, and the `@translate` attribute. In addition, the `<sort-as>` and `<index-sort-as>` elements provide support for sorting in languages in which the correct sorting of an element requires text that is different from the base content of the element.

The `@xml:lang` attribute

The `@xml:lang` attribute specifies the language and (optional) locale of the element content. The `@xml:lang` attribute applies to all attributes and content of the element where it is specified, unless it is overridden with `@xml:lang` on another element within that content.

The `@xml:lang` attribute *SHOULD* be explicitly set on the root element of each map and topic.

Setting the `@xml:lang` attribute in the DITA source ensures that processors handle content in a language- and locale-appropriate way. If the `@xml:lang` attribute is not set, processors assume a default value which might not be appropriate for the DITA content. When the `@xml:lang` attribute is specified for a document, DITA processors *MUST* use the specified value to determine the language of the document.

Setting the `@xml:lang` attribute in the source language document facilitates the translation process; it enables translation tools (or translators) to simply change the value of the existing `@xml:lang` attribute to the value of the target language. Some translation tools support changing the value of an existing `@xml:lang` attribute, but they do not support adding new markup to the document that is being translated. Therefore, if source language content does not set the `@xml:lang` attribute, it might be difficult or impossible for the translator to add the `@xml:lang` attribute to the translated document.

If the root element of a map or a top-level topic has no value for the `@xml:lang` attribute, a processor *SHOULD* assume a default value. The default value of the processor can be either fixed, configurable, or derived from the content itself, such as the `@xml:lang` attribute on the root map.

The `@xml:lang` attribute is described in the [XML Recommendation](#). Note that the recommended style for the `@xml:lang` attribute is lowercase language and (optional) uppercase, separated by a hyphen, for example, "en-US" or "sp-SP" or "fr". According to RFC 5646, *Tags for Identifying Languages*, language codes are case insensitive.

Recommended use in topics

For a DITA topic that contains a single language, set the `@xml:lang` attribute on the highest-level element that contains content.

When a DITA topic contains more than one language, set the `@xml:lang` attribute on the highest-level element to specify the *primary language and locale* that applies to the topic. If part of a topic is written in a different language, authors should ensure that the part is enclosed in an element with the `@xml:lang` attribute set appropriately. This method of overriding the default document language applies to both block and inline elements that use the alternate language. Processors *SHOULD* style each element in a way that is appropriate for its language as identified by the `@xml:lang` attribute.

Recommended use in maps

The `@xml:lang` attribute can be specified on the `<map>` element. The `@xml:lang` attribute cascades within the map in the same way that it cascades within a topic. However, since the `@xml:lang` attribute is an inherent property of the XML document, the value of the `@xml:lang` attribute does not cascade from one map to another or from a map to a topic; the value of the `@xml:lang` attribute that is specified in a map does not override `@xml:lang` values that are specified in other maps or in topics.

The primary language for the map *SHOULD* be set on the `<map>` element. The specified language remains in effect for all child `<topicref>` elements, unless a child specifies a different value for the `@xml:lang` attribute.

When no `@xml:lang` value is supplied locally or on an ancestor, a processor-determined default value is assumed.

Recommended use with the `@conref` or `@conkeyref` attribute

When a `@conref` or `@conkeyref` attribute is used to include content from one element into another, the processor *MUST* use the effective value of the `@xml:lang` attribute from the referenced element, that is, the element that contains the content. If the referenced element does not have an explicit value for the `@xml:lang` attribute, the processor *SHOULD* default to using the same value that is used for topics that do not set the `@xml:lang` attribute.

This behavior is shown in the following example, where the value of the `@xml:lang` attribute of the included note is obtained from its parent `<section>` element that sets the `@xml:lang` attribute to "fr". When the `installingAcme.dita` topic is processed, the `<note>` element with the `@id` attribute set to "mynote" has an effective value for the `@xml:lang` attribute of "fr".

```
<?xml version="1.0"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "task.dtd">
<task xml:lang="en" id="install_acme">
  <title>Installing Acme</title>
  <shortdesc>Step-by-step details about how to install Acme.</shortdesc>
  <taskbody>
    <prereq>
      <p>Special notes when installing Acme in France:</p>
      <note id="mynote" conref="warningsAcme.dita#topic_warnings/frenchwarnings"/>
    </prereq>
  </taskbody>
</task>
```

Figure 34: `installingAcme.dita`

```
<?xml version="1.0"?>
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="topic_warnings">
  <title>Warnings</title>
  <body>
    <section id="qqwwee" xml:lang="fr">
      <title>French warnings</title>
      <p>These are our French warnings.</p>
      <note id="frenchwarnings">Note in French!</note>
    </section>
    <section xml:lang="en">
      <title>English warnings</title>
      <p>These are our English warnings.</p>
    </section>
  </body>
</topic>
```



```
<note id="englishwarnings">Note in English!</note>
</section>
</body>
</topic>
```

Figure 35: warningsAcme.dita

The @dir attribute

The @dir attribute provides instructions to processors about how *bi-directional text* should be rendered.

Bi-directional text is text that contains text in both text directionalities, right-to-left (RTL) and left-to-right (LTR). For example, languages such as Arabic, Hebrew, Farsi, Urdu, and Yiddish have text written from right-to-left; however, numerics and embedded sections of Western language text are written from left to right. Some multilingual documents also contain a mixture of text segments in two directions.

DITA contains the following attributes that have an effect on bi-directional text processing:

@xml:lang

Identifies the language and locale, and so can be used to identify text that requires bi-directional rendering.

@dir

Identifies or overrides the text directionality. It can be set to "ltr", "rtl", "lro", or "rlo"

In general, properly-written mixed text does not need any special markers; the Unicode bidirectional algorithm positions the punctuation correctly for a given language. The processor is responsible for displaying the text properly. However, some rendering systems might need directions for displaying bidirectional text, such as Arabic, properly. For example, Apache FOP might not render Arabic properly unless the left-to-right and right-to-left indicators are used.

The use of the @dir attribute and the Unicode algorithm is explained in the article [Specifying the direction of text and tables: the dir attribute \(http://www.w3.org/TR/html4/struct/dirlang.html#h-8.2\)](http://www.w3.org/TR/html4/struct/dirlang.html#h-8.2). This article contains several examples of how to use the @dir attribute set to either "ltr" or "rtl". There is no example of setting the @dir attribute to either "lro" or "rlo", although it can be inferred from the example that uses the <bdo> element, a now-deprecated W3C mechanism for overriding the entire Unicode bidirectional algorithm.

Recommended usage

The @dir attribute, together with the @xml:lang attribute, is essential for rendering table columns and definition lists in the proper order.

In general text, the Unicode Bidirectional algorithm, as specified by the @xml:lang attribute together with the @dir attribute, provides for various levels of bidirectionality:

- Directionality is either explicitly specified via the @xml:lang attribute in combination with the @dir attribute on the highest level element (topic or derived peer for topics, map for ditamaps) or assumed by the processing application. If used, the @dir attribute *SHOULD* be specified on the highest level element in the topic or document element of the map.
- When embedding a right-to-left text run inside a left-to-right text run (or vice-versa), the default direction might provide incorrect results based on the rendering mechanism, especially if the embedded text run includes punctuation that is located at one end of the embedded text run. Unicode defines spaces and punctuation as having neutral directionality and defines directionality for these neutral characters when they appear between characters having a strong directionality (most characters that are not spaces or punctuation). While the default direction is often sufficient to determine the correct directionality of the language, sometimes it renders the characters incorrectly (for example, a question mark at the end of a Hebrew question might appear at the beginning of the question instead of at the end or a parenthesis might render incorrectly).

To control this behavior, the `@dir` attribute is set to "ltr" or "rtl" as needed, to ensure that the desired direction is applied to the characters that have neutral bidirectionality. The "ltr" and "rtl" values override only the neutral characters (for example, spaces and punctuation), not all Unicode characters.



Note: Problems with Unicode rendering can be caused by the rendering mechanism. The problems are not due to the XML markup itself.

- Sometimes you might want to override the default directionality for strongly bidirectional characters. Overrides are done using the "lro" and "rlo" values, which overrides the Unicode Bidirectional algorithm. This override forces a direction on the contents of the element. These override attributes give the author a brute force way of setting the directionality independent of the Unicode Bidirectional algorithm. The gentler "ltr" and "rtl" values have a less radical effect, only affecting punctuation and other so-called neutral characters.

For most authoring needs, the "ltr" and "rtl" values are sufficient. Use the override values only when you cannot achieve the desired effect using the the "ltr" and "rtl" values.

Processing expectations

Applications that process DITA documents, whether at the authoring, translation, publishing, or any other stage, *SHOULD* fully support the Unicode bidirectional algorithm to correctly implement the script and directionality for each language that is used in the document.

Applications *SHOULD* ensure that the root element in every topic document and the root element in the root map has values for the `@dir` and `@xml:lang` attributes.

Processing documents with different values of the `@domains` attribute

When DITA elements are copied from one document to another, processors need to determine the validity of the copied elements. This copying might occur as the result of a content reference (conref) or key reference (keyref), or it might occur in the context of an author editing a DITA document.

A processor can examine the value of the `@domains` attribute and compare the set of modules listed to the set of modules for which it provides direct support. It then can take appropriate action if it does not provide support for a given module, for example, issuing a warning before applying fallback processing.

Documents might have incompatible constraints applied; see [Weak and strong constraints](#) on page 133 for more information about constraint compatibility checking.

When copying content from one DITA document to another, processors *SHOULD* determine if the data being copied (the copy source) requires modules that are not required by the document into which the data is to be copied (the copy target). Such a copy operation is always safe if the copy source requires a subset of the modules that are required by the copy target. Such a copy is unsafe if the copy source requires modules that are not required by the copy target.

When a copy operation is unsafe, processors *MAY* compare the copy source to the copy target to determine if the copy source satisfies the constraints of the copy target. If the copy source meets the copy target constraints, the copy operation can proceed. Processors *SHOULD* issue a warning that the copy was allowed but the constraints are not compatible. If the copy source does not meet the constraints of the copy target, processors *MAY* apply generalization until the generalized result either satisfies the copy target constraints or no further generalization can be performed. If the copy operation can be performed following generalization, the processor *SHOULD* issue a warning that the constraints are not compatible and generalization had to be performed in order to complete the copy operation.

Sorting

Processors can be configured to sort elements. Typical processing includes sorting glossary entries, lists of parameters or reference entries in custom navigation structures, and tables based on the contents of cells in specific columns or rows.

Each element to be sorted must have some inherent text on which it will be sorted. This text is the *base sort phrase* for the element. For elements that have titles, the base sort phrase usually is the content of the `<title>` element. For elements that do not have titles, the base sort phrase might be literal content in the DITA source, or it might be generated or constructed based on the semantics of the element involved; for example, it could be constructed from various attribute or metadata values. Processors that perform sorting *SHOULD* explicitly document how the base sort phrase is determined for a given element.

The `<sort-as>` element can be used to specify an effective sort phrase when the base sort phrase is not appropriate for sorting. For index terms, the `<index-sort-as>` element can be used to specify the effective sort phrase for an index entry.

The details of sorting and grouping are implementation specific. Processors might provide different mechanisms for defining or configuring collation and grouping details. Even where the `<sort-as>` element is specified, two processors might produce different sorted and grouped results because they might use different collation and grouping rules. For example, one processor might be configured to sort English terms before non-English terms, while another might be configured to sort them after. The grouping and sorting of content is subject to local editorial rules.

When a `<sort-as>` element is specified, processors that sort the containing element *MUST* construct the effective sort phrase by prepending the content of the `<sort-as>` element to the base sort phrase. This ensures that two items with the same `<sort-as>` element but different base sort phrases will sort in the appropriate order.

For example, if a processor uses the content of the `<title>` element as the base sort phrase, and the title of a topic is "24 Hour Support Hotline" and the value of the `<sort-as>` element is "twenty-four hour", then the effective sort phrase would be "twenty-four hour24 Hour Support Hotline".

Configuration, specialization, generalization, and constraints

The extension facilities of DITA allow existing vocabulary and constraint modules to be combined to create specific DITA document types. Vocabulary modules also can be specialized to meet requirements that are not satisfied by existing markup.

Overview of DITA extension facilities

DITA provides three extension facilities: configuration, constraint, and specialization. In addition, generalization augments specialization.

Configuration

Configuration enables the definition of DITA document types that include only the vocabulary modules that are required for a given set of documents. There is no need to modify the vocabulary modules. Configurations are implemented as document type shells.

Specialization

Specialization enables the creation of new element types in a way that preserves the ability to interchange those new element types with conforming DITA applications. Specializations are implemented as vocabulary modules, which are integrated into document-type shells.

Specializations are implemented as sets of vocabulary modules, each of which declares the markup and entities that are unique to a specialization. The separation of the vocabulary and its declarations into modules makes it easy to extend existing modules, because new modules can be added without affecting existing document types. It also makes it easy to assemble elements from different sources into a single document-type shell and to reuse specific parts of the specialization hierarchy in more than one document-type shell.

Generalization

Generalization is the process of reversing a specialization. It converts specialized elements or attributes into the original types from which they were derived.

Constraint

Constraint enables the restriction of content models and attribute lists for individual elements. There is no need to modify the vocabulary modules. Constraints are implemented as constraint modules, which are integrated into document-type shells.

Configuration

Configuration enables the definition of DITA document types that include only the vocabulary modules that are required for a given set of documents. There is no need to modify the vocabulary modules. Configurations are implemented as document-type shells.

Overview of document-type shells

A document type shell is an XML grammar file that specifies the elements and attributes that are allowed in a DITA document. The document type shell integrates structural modules, domain modules, and constraint modules. In addition, a document type shell specifies whether and how topics can nest.

A DITA document must either have an associated document-type definition or all required attributes must be made explicit in the document instances. Most DITA documents have an associated document-type shell. DITA documents that reference a document-type shell can be validated using standard XML processors. Such validation enables processors to read the XML grammar files and determine default values for the @domains and @class attributes.

The following figure illustrates the relationship between a DTD-based DITA document, its document-type shell, and the various vocabulary modules that it uses. A similar structure applies to DITA documents that use other XML grammars.

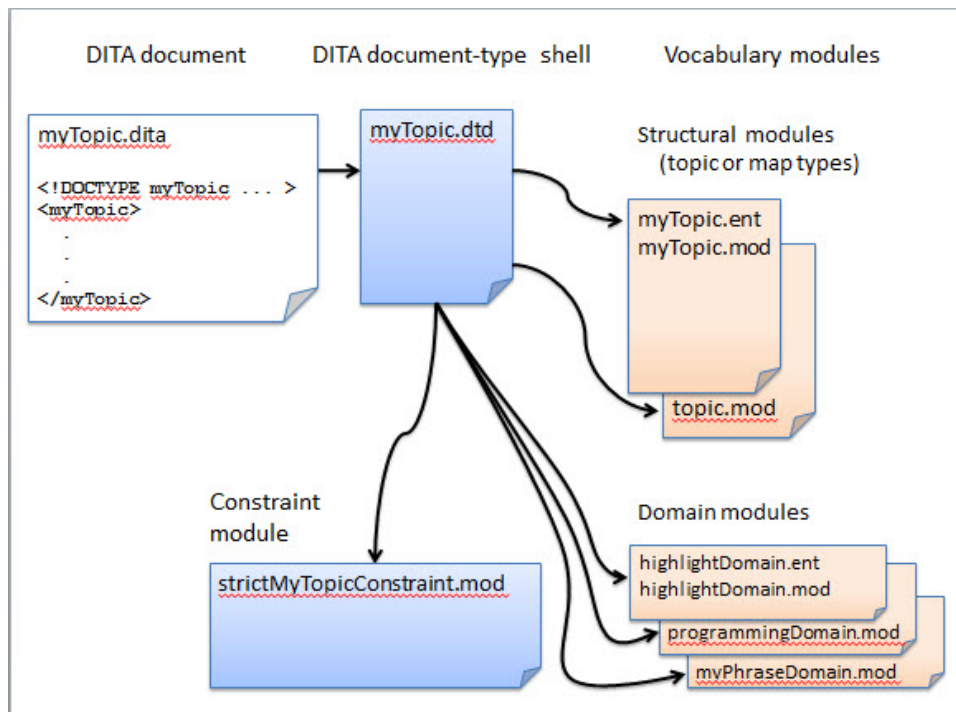


Figure 36: Document type shell

The DITA specification contains a starter set of document-type shells. These document type shells are commented and can be used as templates for creating custom document-type shells. While the OASIS-provided document-type shells can be used without any modification, creating custom document-type shells is a best practice. If the document-type shells need to be modified in the future, for example, to include a specialization or integrate a constraint, the existing DITA documents will not need to be modified to reference a new document-type shell.

Rules for document-type shells

This topic collects the rules that concern DITA document-type shells.

- While the DITA specification only defines coding requirements for DTD, RELAX NG, and XML Schema documents, conforming DITA documents *MAY* use other document-type constraint languages, such as Schematron.
- With two exceptions, a document-type shell *MUST NOT* directly define element or attribute types; it only includes and configures vocabulary and constraint modules. The exceptions to this rule are the following:
 - The ditabase document-type shell directly defines the `<dita>` element.
 - RNG- and XML Schema-based shells directly specify values for the `@domains` attribute; these values reflect the details of the domains and structural types that are integrated by the document-type shell.
- Document type shells that are not provided by OASIS *MUST* have a unique public identifier, if public identifiers are used.
- Document type shells that are not provided by OASIS *MUST NOT* indicate OASIS as the owner; the public identifier or URN for such document-type shells *SHOULD* reflect the owner or creator of the document-type shell.

For example, if example.com creates a copy of the document type shell for topic, an appropriate public identifier would be "-//example.com//DTD DITA Topic//EN", where "example.com" is the owner identifier component of the public identifier. An appropriate URN would be "urn:example.com:names:dita:rng:topic.rng".

Equivalence of document-type shells

Two distinct DITA document types that are taken from different tools or environments might be functionally equivalent.

A DITA document type is defined by the following:

- The set of modules that are declared in the `@domains` attribute on the root element of the document
- The values of the `@class` attributes of all the elements in the document
- Rules for topic nesting

Two document-type shells define the same DITA document type if they integrate identical vocabulary modules, constraint modules, and rules for topic nesting. For example, a document type shell that is an unmodified copy of the OASIS-provided document-type shell for topic defines the same DITA document type as the original document-type shell. However, the new document-type shell has the following differences:

- It is a distinct file that is stored in a different location.
- It has a distinct system identifier.
- If it has a public identifier, the public identifier is unique.



Note: The public or system identifier that is associated with a given document-type shell is not, by itself, necessarily distinguishing. This is because two different people or groups might use the same modules and constraints to assemble equivalent document type shells, while giving them different names or public identifiers.

Conformance of document-type shells

DITA documents typically are governed by a conforming DITA document-type shell. However, the conformance of a DITA document is a function of the document instance, not its governing grammar. Conforming DITA documents are not required to use a conforming document-type shell.

Conforming DITA documents are not required to have any governing document type declaration or schema. There might be compelling or practical reasons to use non-conforming document-type shells. For example, a document might use a document-type shell that does not conform to the DITA requirements for shells in order to meet the needs of a specific application or tool. Such a non-conforming document-type shell still might enable the creation of conforming DITA content.

Specialization

The specialization feature of DITA allows for the creation of new element types and attributes that are explicitly and formally derived from existing types. This facilitates interchange of conforming DITA content and ensures a minimum level of common processing for all DITA content. It also allows specialization-aware processors to add specialization-specific processing to existing base processing.

Overview of specialization

Specialization allows information architects to define new kinds of information (new structural types or new domains of information), while reusing as much of existing design and code as possible, and minimizing or eliminating the costs of interchange, migration, and maintenance.

Specialization modules enable information architects to create new element types and attributes. These new element types and attributes are derived from existing element types and attributes.

In traditional XML applications, all semantics for a given element instance are bound to the element type, such as `<para>` for a paragraph or `<title>` for a title. The XML specification provides no built-in mechanism for relating two element types to say "element type B is a subtype of element type A".

In contrast, the DITA specialization mechanism provides a standard mechanism for defining that an element type or attribute is derived from an ancestor type. This means that a specialized type inherits the semantics and default processing behavior from its ancestor type. Additional processing behavior optionally can be associated with the specialized descendant type.

For example, the `<section>` element type is part of the DITA base or core. It represents an organizational division in a topic. Within the task information type (itself a specialization of `<topic>`), the `<section>` element type is further specialized to other element types (such as `<prereq>` and `<context>`) that provide more precise semantics about the type of organizational division that they represent. The specialized element types inherit both semantic meaning and default processing from the ancestor elements.

There are two types of DITA specializations:

Structural specialization

Structural specializations are developed from either topic or map types. Structural specializations enable information architect to add new document types to DITA. The structures defined in the new document types either directly use or inherit from elements found in other document types. For example; concept, task, and reference are specialized from topic, whereas bookmap is specialized from map.

Domain specialization

Domain specializations are developed from elements defined with `topic` or `map`, or from the `@props` or `@base` attributes. They define markup for a specific information domain or subject area. Domain specialization can be added to document-type shells.

Each type of specialization module represents an “is a” hierarchy, in object-oriented terms, with each structural type or domain being a subclass of its parent. For example, a specialization of `task` is still a `task`, and a specialization of the user interface domain is still part of the user interface domain. A given domain can be used with any `map` or `topic` type. In addition, specific structural types might require the use of specific domains.

Use specialization when you need a new structural type or domain. Specialization is appropriate in the following circumstances:

- You need to create markup to represent new semantics (meaningful categories of information). This might enable you to have increased consistency or descriptiveness in your content model.
- You have specific needs for output processing and formatting that cannot be addressed using the current content model.

Do not use specialization to simply eliminate element types from specific content models. Use constraint modules to restrict content models and attribute lists without changing semantics.

Modularization

Modularization is at the core of DITA design and implementation. It enables reuse and extension of the DITA specialization hierarchy.

The DITA XML grammar files are a set of module files that declare the markup and entities that are required for each specialization. The document-type shell then integrates the modules that are needed for a particular authoring and publishing context.

Because all the pieces are modular, the task of developing a new information type or domain is easy. An information architect can start with existing base types (`topic` or `map`) -- or with an existing specialization if it comes close to matching their business requirements -- and only develop an extension that adds the extra semantics or functionality that is required. A specialization reuses elements from ancestor modules, but it only needs to declare the elements and attributes that are unique to the specialization. This saves considerable time and effort; it also reduces error, enforces consistency, and makes interoperability possible.

Because all the pieces are modular, it is easy to reuse different modules in different contexts. For example, a company that produces machines can use the `task requirements` and `hazard statements` domains, while a company that produces software can use the `software`, `user interface`, and `programming` domains. A company that produces health information for consumers can avoid using any of the standard domains, and instead develop a new domain that contains the elements necessary for capturing and tracking the comments made by medical professionals who review their information for accuracy and completeness.

Because all the pieces are modular, new modules can be created and put into use without affecting existing document-type shells. For example, a marketing division of a company can develop a new specialization for message campaigns and have their content authors begin using that specialization, without affecting any of the other information types that they have in place.

Vocabulary modules

A DITA element type or attribute is declared in exactly one vocabulary module.

The following terminology is used to refer to DITA vocabulary modules:

structural module

A vocabulary module that defines a top-level map or topic type. Structural modules also can define specializations of, or reuse elements from, domain or other structural modules. When this happens, the structural module becomes dependent.

element domain module

A vocabulary module that defines one or more specialized element types that can be integrated with maps or topics.

attribute domain module

A vocabulary module that defines exactly one specialization of either the `@base` or `@props` attribute.

For structural types, the module name is typically the same as the root element. For example, "task" is the name of the structural vocabulary module whose root element is `<task>`.

For element domain modules, the module name is typically a name that reflects the subject domain to which the domain applies, such as "highlight" or "software". Domain modules often have an associated short name, such as "hi-d" for the highlighting domain or "sw-d" for the software domain.

The name (or short name) of an element domain module is used to identify the module in `@class` and `@domains` attribute values. While module names need not be globally unique, module names must be unique within the scope of a given specialization hierarchy. The short name must be a valid XML name token.

Structural modules based on topic *MAY* define additional topic types that are then allowed to occur as subordinate topics within the top-level topic. However, such subordinate topic types *MAY NOT* be used as the root elements of conforming DITA documents. For example, a top-level topic type might require the use of subordinate topic types that would only ever be meaningful in the context of their containing type and thus would never be candidates for standalone authoring or aggregation using maps. In that case, the subordinate topic type can be declared in the module for the top-level topic type that uses it. However, in most cases, potential subordinate topics should be defined in their own vocabulary modules.

Domain elements intended for use in topics *MUST* ultimately be specialized from elements that are defined in the topic module. Domain elements intended for use in maps *MUST* ultimately be specialized from elements defined by or used in the map module. Maps share some element types with topics but no map-specific elements can be used within topics.

Specialization rules for element types

There are certain rules that apply to element type specializations.

A specialized element type has the following characteristics:

- A properly-formed `@class` attribute that specifies the specialization hierarchy of the element
- A content model that is the same or less inclusive than that of the element from which it was specialized
- A set of attributes that are the same or a subset of those of the element from which it was specialized
- Values or value ranges of attributes that are the same or a subset of those of the element from which it was specialized

DITA elements are never in a namespace. Only the `@DITAArchVersion` attribute is in a DITA-defined namespace. All other attributes, except for those defined by the XML standard, are in no namespace.

This limitation is imposed by the details of the `@class` attribute syntax, which makes it impractical to have namespace-qualified names for either vocabulary modules or individual element types or attributes. Elements included as descendants of the DITA `<foreign>` element type can be in any namespace.



Note: Domain modules that are intended for wide use should define element type names that are unlikely to conflict with names used in other domains, for example, by using a domain-specific prefix on all names.

Specialization rules for attributes

There are certain rules that apply to attribute specializations.

A specialized attribute has the following characteristics:

- It is specialized from `@props` or `@base`.
- It is declared as a global attribute. Attribute specializations cannot be limited to specific element types.
- It does not have values or value ranges that are more extensive than those of the attribute from which it was specialized.
- Its values must be alphanumeric space-delimited values. In generalized form, the values must conform to the rules for attribute generalization.

@class attribute rules and syntax

The specialization hierarchy of each DITA element is declared as the value of the `@class` attribute. The `@class` attribute provides a mapping from the current name of the element to its more general equivalents, but it also can provide a mapping from the current name to more specialized equivalents. All specialization-aware processing can be defined in terms of `@class` attribute values.

The `@class` attribute tells a processor what general classes of elements the current element belongs to. DITA scopes elements by module type (for example topic type, domain type, or map type) instead of document type, which lets document type developers combine multiple module types in a single document without complicating transformation logic.

The sequence of values in the `@class` attribute is important because it tells processors which value is the most general and which is most specific. This sequence is what enables both specialization aware processing and generalization.

Syntax

Values for the `@class` attribute have the following syntax requirements:

- An initial "-" or "+" character followed by one or more spaces. Use "-" for element types that are defined in structural vocabulary modules, and use "+" for element types that are defined in domain modules.
- A sequence of one or more tokens of the form "*modulename/typename*", with each token separated by one or more spaces, where *modulename* is the short name of the vocabulary module and *typename* is the element type name. Tokens are ordered left to right from most general to most specialized.

These tokens provide a mapping for every structural type or domain in the ancestry of the specialized element. The specialization hierarchy for a given element type must reflect any intermediate modules between the base type and the specialization type, even those in which no element renaming occurs.

- At least one trailing space character (" "). The trailing space ensures that string matches on the tokens can always include a leading and trailing space in order to reliably match full tokens.

Rules

When the `@class` attribute is declared in an XML grammar, it *MUST* be declared with a default value. In order to support generalization round-tripping (generalizing specialized content into a generic form and then returning it to the specialized form) the default value *MUST NOT* be fixed. This allows a generalization process to overwrite the default values that are defined by a general document type with specialized values taken from the document being generalized.

A vocabulary module *MUST NOT* change the `@class` attribute for elements that it does not specialize, but simply reuses by reference from more generic levels. For example, if `<task>`, `<bctask>`, and `<quitask>` use the `<p>` element without specializing it, they *MUST NOT* declare mappings for it.

Authors *SHOULD NOT* modify the `@class` attribute.

Example: DTD declaration for `@class` attribute for the `<step>` element

The following code sample lists the DTD declaration for the `@class` attribute for the `<step>` element:

```
<!ATTLIST step          class  CDATA "- topic/li task/step ">
```

This indicates that the `<step>` element is specialized from the `` element in a generic topic. It also indicates explicitly that the `<step>` element is available in a task topic; this enables round-trip migration between upper level and lower level types without the loss of information.

Example: Element with `@class` attribute made explicit

The following code sample shows the value of the `@class` attribute for the `<wintitle>` element:

```
<wintitle class="+ topic/keyword ui-d/wintitle ">A specialized keyword</wintitle>
```

The `@class` attribute and its value is generally not surfaced in authored DITA topics, although it might be made explicit as part of a processing operation.

Example: `@class` attribute with intermediate value

The following code sample shows the value of a `@class` attribute for an element in the `guitask` module, which is specialized from `<task>`. The element is specialized from `<keyword>` in the base topic vocabulary, rather than from an element in the task module:

```
<windowname class="- topic/keyword task/keyword guitask/windowname ">...</windowname>
```

The intermediate values are necessary so that generalizing and specializing transformations can map the values simply and accurately. For example, if `task/keyword` was missing as a value, and a user decided to generalize this `guitask` up to a task topic, then the transformation would have to guess whether to map to `keyword` (appropriate if `task` is more general than `guitask`, which it is) or leave it as `windowname` (appropriate if `task` were more specialized, which it isn't). By always providing mappings for more general values, processors can then apply the simple rule that missing mappings must by default be to more specialized values than the one we are generalizing to, which means the last value in the list is appropriate. For example, when generalizing `<guitask>` to `<task>`, if a `<p>` element has no target value for `<task>`, we can safely assume that `<p>` does not specialize from `<task>` and should not be generalized.

`@domains` attribute rules and syntax

The `@domains` attribute enables processors to determine whether two elements or two documents use compatible domains. The attribute is declared on the root element for each topic or map type. Each structural, domain, and constraint module defines its ancestry as a parenthesized sequence of space-separated module names; the effective value of the `@domains` attribute is composed of these parenthesized sequences.

Document type shells collect the values that are provided by each module to construct the effective value of the `@domains` attribute. Processors can examine the collected values when content from one document is used in another, in order to determine whether the content is compatible.

For example, when an author pastes content from one topic into another topic within an XML editor, the application can use the `@domains` attribute to determine if the two topics use compatible domains. If not, copied content from the first topic might need to be generalized before it can be placed in the other topic.

The @domains attribute serves the same function when an element uses the @conref attribute to reference a more specialized version of the element. For example, a <note> element in a concept topic conrefs a <hazardstatement> element in a reference document. If the hazard statement domain is not available in the concept topic, the <hazardstatement> element is generalized to a <note> element when the content reference is resolved.

Syntax and rules

Each domain and constraint module *MUST* provide a value for use by the @domains attribute. Each structural vocabulary module *SHOULD* provide a value for use by the @domains attribute, and it *MUST* do so when it has a dependency on elements from any module that is not part of its specialization ancestry.

Values provided for the @domains attribute values are specified from root module (map or topic) to the provided module.

structural modules

The value of the @domains attribute includes each module in the specialization ancestry:

```
'(', topic-or-map, (' ', module)+, ')'
```

For example, consider the <glossentry> specialization, in which the topic type is specialized to the concept type, and the concept type is specialized to glossentry. The structural module contribution to the value of the @domains attribute for the glossentry structural module is (topic concept glossentry).

structural modules with dependencies

Structural modules can directly reference or specialize elements from modules that are outside of their specialization ancestry. They also can define specialized elements that reference specialized attributes. In these cases the structural module has a dependency on the non-ancestor module, and the structural module contribution to the value of the @domains attribute *MUST* include the names of each dependent, non-ancestor module.

Dependencies are included in the value of the @domains attribute following the name of the structural module with the dependency on the non-ancestor module. Domain or attribute modules are appended to the name of the structural module with the dependency on the non-ancestor module, or to previous dependencies, separated by "+". Dependencies on structural specialization modules are appended to the name of the structural module with the dependency on the non-ancestor module, or to previous dependencies, separated by "++". The syntax is the same as for other structural modules, except that added modules can include these dependencies:

```
'(', topic-or-map, (' ', module-plus-optional-dependency-list)+, ')'
```

When the structural module is included in a document-type shell, all dependency modules also are included along with their own @domains values.

For example, the cppAPIRef structural module is specialized from reference, which is specialized from topic. The cppAPIRef module has a dependency on the cpp-d element domain and on the compilerTypeAtt-d attribute domain. The dependencies are listed after the name of cppAPIRef:

```
(topic reference cppAPIRef+cpp-d+compilerTypeAtt-d)
```

Similarly, a codeChecklist structural module is specialized from reference, which is specialized from topic. The codeChecklist module has a dependency on the pr-d domain and on the task structural specialization.

Again, the dependencies are listed after the name of `codeChecklist`. The `pr-d` domain and the task module each contribute their own values, so taken together these modules contribute the following values:

```
(topic reference codeChecklist+pr-d++task) (topic pr-d) (topic task)
```

element domains

The value includes the structural type ancestry and, if applicable, the domain module ancestry from which the domain is specialized:

```
'(', topic-or-map, (' ', domain-module)+, ')'
```

For example, the highlighting domain (specialized from `topic`) supplies the following value: `(topic hi-d)`. A CPP domain that is specialized from the programming domain, which in turn is specialized from `topic`, supplies the following value: `(topic pr-d cpp-d)`.

structural constraint modules

The value includes the structural type ancestry followed by the name of the constraint domain:

```
'(', inheritance-hierarchy qualifierTagname-c, ')'
```

where:

- *inheritance-hierarchy* is the specialization hierarchy, for example, `topic task`.
- *qualifier* is a string that is specific to the constraints module and characterizes it, for example, "strict" or "requiredTitle" or "myCompany-".
- *Tagname* is the element type name with an initial capital, for example, "Taskbody" or "Topic".
- The literal "-c" indicates that the name is the name of a constraint.

For example, the `strictTaskbody` constraint applies to the task module, which is specialized from `topic`, resulting in the following value: `(topic task strictTaskbody-c)`.

Optionally, a domains contribution can indicate a strong constraint by preceding the domains contribution with the letter "s". For example, `s(topic task strictTaskbody-c)` indicates a strong constraint.

domain constraint modules

The value includes the specialization ancestry followed by the name of the constraint domain:

```
'(', inheritance-hierarchy qualifierdomainDomain-c ')'
```

where:

- *inheritance-hierarchy* is the specialization hierarchy, for example, `topic hi-d`.
- *qualifier* is a string that is specific to the constraints module and characterizes it, for example, "noSyntaxDiagram" or "myCompany-".
- *domain* is the name of the domain to which the constraints apply, for example, "Highlighting" or "Programming".
- The literal "-c" indicates that the name is the name of a constraint.

For example, a domain constraint module that restricts the highlighting domain includes a value like the following: `(topic hi-d basic-HighlightingDomain-c)`

attribute domains

The value uses an "a" before the initial parenthesis to indicate an attribute domain. Within the parenthesis, the value includes the attribute specialization hierarchy, starting with @props or @base:

```
'a(', props-or-base, (' ', attname)+, ')'
```

For example, the @mySelectAttribute specialized from @props results in the following value: a(props mySelectAttribute)

Example: Task with multiple domains

In this example, a document-type shell integrates the task structural module and the following domain modules:

Domain	Domain short name
User interface	ui-d
Software	sw-d
Programming	pr-d

The value of the @domains attribute includes one value from each module; the effective value is the following:

```
domains="(topic task) (topic ui-d) (topic sw-d) (topic pr-d)"
```

If the document-type shell also used a specialization of the programming domain that describes C++ programming (with a short name of "cpp-d"), the new C++ programming domain would add an additional value to the @domains attribute:

```
domains="(topic task) (topic ui-d) (topic sw-d) (topic pr-d) (topic pr-d cpp-d)"
```

Note that the value for the @domains attribute is not authored; Instead, the value is defaulted based on the modules that are included in the document type shell.

Related information

[Processing conrefs](#) on page 82

When processing content references, DITA processors compare the restrictions of each context to ensure that the conrefed content is valid in its new context.

Specializing to include non-DITA content

You can extend DITA to incorporate standard vocabularies for non-textual content, such as MathML and SVG, as markup within DITA documents. This is done by specializing the <foreign> or <unknown> elements.

There are three methods of incorporating foreign content into DITA.

- A domain specialization of the <foreign> or <unknown> element. This is the usual implementation.
- A structural specialization using the <foreign> or <unknown> element. This affords more control over the content.
- Directly embedding the non-DITA content within <foreign> or <unknown> elements. If the non-DITA content has interoperability or vocabulary naming issues such as those that are addressed by specialization in DITA, they must be addressed by means that are appropriate to the non-DITA content.

The <foreign> or <unknown> elements should not be used to include textual content or metadata in DITA documents, except where such content acts as an example or display, rather than as the primary content of a topic.



Note: Beginning with DITA 1.3, both MathML and SVG are domains shipped with the OASIS grammars; they serve as working examples of `<foreign>` specializations.

Example: Including SVG markup within a specialization of `<foreign>`

The following code sample shows how SVG markup can be included within the `<svgcontainer>` element, which is part of the SVG domain and a specialization of the `<foreign>` element.

```
<p>This is an ellipse:
  <svg-container>
    <svg:svg width="100%" height="100%" version="1.1"
    xmlns="http://www.w3.org/2000/svg">

    <ellipse cx="300" cy="150" rx="200" ry="80"
    style="fill:rgb(200,100,50);
    stroke:rgb(0,0,100);stroke-width:2"/>

    </svg:svg>
  </svg-container>.
</p>
```

Example: Creating an element domain specialization for SVG

The following code sample, which is from the `svgDomain.ent` file, shows the domain declaration for the SVG domain.

```
<!-- ===== -->
<!--           SVG DOMAIN ENTITIES           -->
<!-- ===== -->

<!-- SVG elements must be prefixed, otherwise they conflict with
      existing DITA elements (e.g., <desc> and <title>).
-->
<!ENTITY % NS.prefix "INCLUDE" >
<!ENTITY % SVG.prefix "svg" >

<!ENTITY % svg-d-foreign
      "svg-container
      "
>

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

Note that the SVG-specific `%SVG.prefix;` parameter entity is declared. This establishes the default namespace prefix to be used for the SVG content embedded with this domain. The namespace can be overridden in a document-type shell by declaring the parameter entity before the reference to the `svgDomain.ent` file. Other foreign domains may need similar entities when required by the new vocabulary.

For more information, see the `svgDomain.mod` file that is shipped with the OASIS DITA distributions. For an example of including the SVG domain in a document type shell, see `task.dtd`.

Sharing elements across specializations

Specialization enables easy reuse of elements from ancestor specializations. However, it is also possible to reuse elements from non-ancestor specializations, as long as the dependency is properly declared in order to prevent invalid generalization or conref processing.

A structural specialization can incorporate elements from unrelated domains or other structural specializations by referencing them in the content model of a specialized element. The elements included in this manner must be specialized from ancestor content that is valid in the new context. If the reusing and reused specializations share common ancestry, the reused elements must be valid in the reusing context at every level they share in common.

Although a well-designed structural specialization hierarchy with controlled use of domains is still the primary means of sharing and reusing elements in DITA, the ability to also share elements declared elsewhere in the hierarchy allows for situations where relevant markup comes from multiple sources and would otherwise be developed redundantly.

Example: A specialization of `<concept>` reuses an element from the task module

A specialized concept topic could declare a specialized `<process>` section that contains the `<steps>` element that is defined in the task module. This is possible because of the following factors:

- The `<steps>` element is specialized from ``.
- The `<process>` element is specialized from `<section>`, and the content model of `<section>` includes ``.

The `<steps>` element in `<process>` always can be generalized back to `` in `<section>`.

Example: A specialization of `<reference>` reuses an element from the programming domain

A specialized reference topic could declare a specialized list (`<apilist>`) in which each `<apilistitem>` contains an `<apiname>` element that is borrowed from the programming domain.

Generalization

Generalization is the process of reversing a specialization. It converts specialized elements or attributes into the original types from which they were derived.

Overview of generalization

Specialized content can be generalized to any ancestor type. The generalization process can preserve information about the former level of specialization to allow round-tripping between specialized and unspecialized forms of the same content.

All DITA documents contain a mix of markup from at least one structural type and zero or more domains. When generalizing the document, any individual structural type or domain can be left as-is, or it can be generalized to any of its ancestors. If the document will be edited or processed in generalized form, it might be necessary to have a document-type shell that includes all non-generalized modules from the original document-type shell.

Generalization serves several purposes:

- It can be used to migrate content. For example, if a specialization is unsuccessful or is no longer needed, the content can be generalized back to a less specialized form.

- It can be used for temporary round-tripping. For example, if content is shared with a process that is not specialization aware, it can be temporarily generalized for that process and then returned to specialized form.
- It can allow reuse of specialized content in an environment that does not support the specialization. Similar to round-tripping, content can be generalized for sharing, without the need to re-specialize.

When generalizing for migration, the `@class` attribute and `@domains` attribute should be absent from the generalized instance document, so that the default values in the document-type shell are used. When generalizing for round-tripping, the `@class` attribute and `@domains` attribute *SHOULD* retain the original specialized values in the generalized instance document.

Note that when using constraints, a document instance can always be converted from a constrained document type to an unconstrained document type merely by switching the binding of the document instance to the less restricted document type shell (which would also have a different `@domains` attribute declaration). No renaming of elements is needed to remove constraints.

Element generalization

Elements are generalized by examining the `@class` attribute. When a generalization process detects that an element belongs to one of the modules that is being generalized, the element is renamed to a more general form.

For example, the `<step>` element has a `@class` attribute value of `"- topic/li task/step "`. If the task module is being generalized, the `<step>` element is renamed to its more general form from the topic module: ``.

For specific concerns when generalizing structural types with dependencies on non-ancestor modules, see [Generalization with cross-specialization dependencies](#) on page 131.

While the tag name of a given element is normally the same as the type name of the last token in the `@class` value, this is not required. For example, if a generalization process has already run on the element, the `@class` attribute could contain tokens from two or more modules based on the original specialization. In that case, the element name could already match the first token or an intermediate token in the `@class` attribute. A second generalization process could end up renaming the element again or could leave it alone, depending on the target module or document type.

Generalization and conref

To determine compatibility between a document instance and a target document type when resolving a conref reference, a generalization processor can use the `@domains` and `@class` attributes for the document instance and the `@domains` attribute for the target document type to determine how to rename elements in the resolved instance. For each element type, a generalization processor:

- Iterates over the `@class` attribute from specific to general, inspecting the vocabulary modules.
- Identifies the first vocabulary module that is both present in each document type, with a compatible set of constraints for that vocabulary module. If such a module is not found, the instance can only be generalized to a less constrained document type.

Processor expectations when generalizing elements

Generalization processors convert elements from one or more modules into their less specialized form. The list of modules can be supplied to a generalization processor, or it can be inferred based on knowledge of a target document-type shell.

The person or application initiating a generalization process can supply the source and target modules for each generalization, for example, "generalize from reference to topic". Multiple target modules can be specified, for example, "generalize from reference to topic and from ui-d to topic". When the source and target modules are not

supplied, the generalization process is assumed to be from all structural types to the base (topic or map), and no generalization is performed for domains.

The person or application initiating a generalization process also can supply the target document-type shell. When the target document-type shell is not supplied, the generalized document will not contain a reference to a document-type shell.

A generalization processor *SHOULD* be able to handle cases where it is given:

- Only source modules for generalization (in which case the designated source types are generalized to topic or map)
- Only target modules for generalization (in which case all descendants of each target are generalized to that target)
- Both (in which case only the specified descendants of each target are generalized to that target)

For each structural type instance, the generalization processor checks whether the structural type instance is a candidate for generalization, or whether it has domains that are candidates for generalization. It is important to be selective about which structural type instances to process; if the process simply generalizes every element based on its @class attribute values, an instruction to generalize "reference" to "topic" could leave a specialization of reference with an invalid content model, since any elements it reuses from "reference" would have been renamed to topic-level equivalents.

The @class attribute for the root element of the structural type is checked before generalizing structural types:

	Source module unspecified	Source module specified
Target module unspecified	Generalize this structural type to its base ancestor	Check whether the root element of the topic type matches a specified source module; generalize to its base ancestor if it does, otherwise ignore the structural type instance unless it has domains to generalize.
Target module specified	Check whether the @class attribute contains the target module. If it does contain the target, rename the element to the value associated with the target module. Otherwise, ignore the element.	It is an error if the root element matches a specified source but its @class attribute does not contain the target. If the root element matches a specified source module and its @class attribute does contain the target module, generalize to the target module. Otherwise, ignore the structural type instance unless it has domains to generalize.

The @domains attribute for the root element of the structural type is checked before generalizing domains:

	Source module unspecified	Source module specified
Target module unspecified	Do not generalize domain specializations in this structural type.	Check whether the @domains attribute lists the specified domain; proceed with generalization if it does, otherwise ignore the structural type instance unless it is itself a candidate for generalization.
Target module specified	Check whether the @domains attribute contains the target module. If it does, generalize to the target module. Otherwise, skip the structural type instance unless it is itself a candidate for generalization.	It is an error if the @domains attribute matches a specified source but the domain value string does not contain the target. If the @domains attribute matches a specified source module and the domain value string does contain the target module, generalize to the target module. Otherwise, ignore the structural type instance unless it is itself a candidate for generalization.

For each element in a candidate structural type instance:

	Source module unspecified	Source module specified
Target module unspecified	If the @class attribute starts with "-" (part of a structural type), rename the element to its base ancestor equivalent. Otherwise ignore it.	Check whether the last value of the @class attribute matches a specified source; generalize to its base ancestor if it does, otherwise ignore the element.
Target module specified	Check whether the @class attribute contains the target module; rename the element to the value associated with the target module if it does contain the target, otherwise ignore the element.	It is an error if the last value in the @class attribute matches a specified source but the previous values do not include the target. If the last value in the @class attribute matches a specified source module and the previous values do include the target module, rename the element to the value associated with the target module. Otherwise, ignore the element.

When renaming elements during round-trip generalization, the generalization processor *SHOULD* preserve the values of all attributes. When renaming elements during one-way or migration generalization, the process *SHOULD* preserve the values of all attributes except the @class and @domains attribute, both of which should be supplied by the target document type.

Attribute generalization

DITA provides a syntax to generalize attributes that have been specialized from the @props or @base attribute. Specialization-aware processors *SHOULD* process both the specialized and generalized forms of an attribute as equivalent in their values.

When a specialized attribute is generalized to an ancestor attribute, the value of the ancestor attribute consists of the name of the specialized attribute followed by its specialized value in parentheses. For example, if @jobrole is an attribute specialized from @person, which in turn is specialized from @props:

- jobrole="programmer" can be generalized to person="jobrole (programmer) " or to props="jobrole (programmer) "
- props="jobrole (programmer) " can be respecialized to person="jobrole (programmer) " or to jobrole="programmer"

In this example, processors performing generalization and respecialization can use the @domains attribute to determine the ancestry of the specialized @jobrole attribute, and therefore the validity of the specialized @person attribute as an intermediate target for generalization.

If more than one attribute is generalized, the value of each is separately represented in this way in the value of the ancestor attribute.

Generalized attributes are typically not expected to be authored or edited directly. They are used by processors to preserve the values of the specialized attributes during the time or in the circumstances in which the document is in a generalized form.



Note: The @audience, @platform, @product, and @otherprops attributes allow grouped values that reuse the generalized syntax described here; however, these attributes are not specialized or specializeable. For these attributes, it may be typical to author or edit the grouped values directly.

A single element *MUST NOT* contain both generalized and specialized values for the same attribute. For example, the following <p> element provides two values for the @jobrole attribute, one in a generalized syntax and the other in a specialized syntax:

```
<p person="jobrole (programmer) " jobrole="admin">
  <!-- ... -->
</p>
```

This is an error condition, since it means the document has been only partially generalized, or that the document has been generalized and then edited using a specialized document type.

Generalization with cross-specialization dependencies

Dependencies across specializations limit generalization targets to those that either preserve the dependency or eliminate them. Some generalization targets will not be valid and should be detected before generalization occurs.

When a structural specialization has a dependency on a domain specialization, then the domain cannot be generalized without also generalizing the reusing structural specialization.

For example, a structural specialization `codeConcept` might incorporate and require the `<codeblock>` element from the programming domain. A generalization process that turns programming domain elements back into topic elements would convert `<codeblock>` to `<pre>`, making a document that uses `codeConcept` invalid. However, `codeConcept` could be generalized to concept or topic, without generalizing programming domain elements, as long as the target document type includes the programming domain.

When a structural specialization has a dependency on another structural specialization, then both must be generalized together to a common ancestor.

For example, if the task elements in checklist were generalized without also generalizing checklist elements, then the checklist content models that referenced task elements would be broken. And if the checklist elements were generalized to topic without also generalizing the task elements, then the task elements would be out of place, since they cannot be validly present in topic. However, checklist and task can be generalized together to any ancestor they have in common: in this case topic.

When possible, generalizing processes *SHOULD* detect invalid generalization target combinations and report them as errors.

Constraints

Constraint modules define additional constraints for vocabulary modules in order to restrict content models or attribute lists for specific element types, remove certain extension elements from an integrated domain module, or replace base element types with domain-provided, extension element types.

Overview of constraints

Constraint modules enable information architects to restrict the content models or attributes of OASIS-defined DITA grammars. A constraint is a simplification of an XML grammar such that any instance that conforms to the constrained grammar also will conform to the original grammar.

A constraint module can perform the following functions:

Restrict the content model for an element

Constraint modules can modify content models by removing optional elements, making optional elements required, or requiring unordered elements to occur in a specific sequence. Constraint modules cannot make required elements optional or change the order of element occurrence for ordered elements.

For example, a constraint for `<topic>` can require `<shortdesc>`, can remove `<abstract>`, and can require that the first child of `<body>` be `<p>`. A constraint cannot allow `<shortdesc>` to follow `<prolog>`, because the content model for `<topic>` requires that `<shortdesc>` precedes `<prolog>`.

Restrict the attributes that are available on an element

Constraint modules can restrict the attributes that are available on an element. They also can limit the set of permissible values for an attribute.

For example, a constraint for `<note>` can limit the set of allowed values for the `@type` attribute to "note" and "tip". It also can omit the `@othertype` attribute, since it is needed only when the value of the `@type` attribute is "other".

Restrict the elements that are available in a domain

Constraint modules can restrict the set of extension elements that are provided in a domain. They also can restrict the content models for the extension elements.

For example, a constraint on the programming domain can reduce the list of included extension elements to `<codeph>` and `<codeblock>`.

Replace base elements with domain extensions

Constraint modules can replace base element types with the domain-provided extension elements.

For example, a constraint module can replace the `<ph>` element with the domain-provided elements, making `<ph>` unavailable.

Constraint rules

There are certain rules that apply to the design and implementation of constraints.

Contribution to the `@domains` attribute

Each constraint that is integrated into a DITA document type *MUST* be declared in the `@domains` attribute for each structural type that is integrated into the document type. For DTDs, the contribution for the `@domains` attribute is specified in the constraint module file; for XSD and RELAX NG, the contribution to the `@domains` attribute is specified directly in the document type shell.

Content model

The content model for a constrained element must be at least as restrictive as the unconstrained content model for the element.

The content model and attributes of an element can be constrained by only one constraint module. If two constraint modules exist that constrain the content model or attributes for a specific element, those two modules must be replaced with a new constraint module that reflects the aggregation of the two original constraint modules.

Domain constraints

When a domain module is integrated into a document-type shell, the base domain element can be omitted from the domain extension group or parameter entity. In such a case, there is no separate constraint declaration, because the content model is configured directly in the document-type shell.

A domain module can be constrained by only one constraint module. This means that all restrictions for the extension elements that are defined in the domain must be contained within that one constraint module.

Structural constraints

Each constraint module may constrain elements from only one vocabulary module. For example, a single constraint module that constrains `<refsyn>` from `reference.mod` and constrains `<context>` from `task.mod` is not allowed. This rule maintains granularity of reuse at the module level.

Constraint modules that restrict different elements from within the same vocabulary module can be combined with one another. Such combinations of constraints on a single vocabulary module have no meaningful order or precedence.

Constraints, processing, and interoperability

Because constraints can make optional elements required, documents that use the same vocabulary modules might have incompatible constraints. Thus the use of constraints can affect the ability for content from one topic or map to be used in another topic or map.

A constraint does not change basic or inherited element semantics. The constrained instances remain valid instances of the unconstrained element type, and the element type retains the same semantics and `@class` attribute declaration. Thus, a constraint never creates a new case to which content processing might need to react.

For example, a document type constrained to require the `<shortdesc>` element allows a subset of the possible instances of the unconstrained document type with an optional `<shortdesc>` element. Thus, the content processing for topic still works when `<topic>` is constrained to require a short description.

A constrained document type allows only a subset of the possible instances of the unconstrained document type. Thus, for a processor to determine whether a document instance is compatible with another document type, the document instance *MUST* declare any constraints on the document type.

For example, an unconstrained task is compatible with an unconstrained topic, because the `<task>` element can be generalized to `<topic>`. However, if the topic is constrained to require the `<shortdesc>` element, a document type with an unconstrained task is not compatible with the constrained document type, because some instances of the task might not have a `<shortdesc>` element. However, if the task document type also has been constrained to require the `<shortdesc>` element, it is compatible with the constrained topic document type.

Weak and strong constraints

Constraints can be classified into two categories: Weak and strong. This classification determines whether processors enforce strict compatibility during `@conref` or `@conkeyref` resolution.

Strong constraints

Constraints for which processors enforce strict compatibility during `@conref` or `@conkeyref` resolution.

Weak constraints

Constraints for which a processor does not enforce strict compatibility during `@conref` or `@conkeyref` resolution.

By default, constraints are weak unless they are explicitly designated as strong.

Any constraint declaration can designate a constraint as strong. A constraint can be designated as strong by prefixing the letter "s" to the domains attribute contribution, for example, "s(topic task strictTaskbody-c)". Processors also can be configured to treat all constraints as strong.

The following behavior is expected of processors:

- Processors *MAY* perform constraint compatibility checking.
- If processors perform constraint compatibility checking, they *SHOULD* enforce strict compatibility for strong constraints.
- Processors *MAY* have an option for configuring whether all constraints are treated as strong constraints.

Conref compatibility with constraints

To determine compatibility between two document instances, a conref processor checks the `@domains` attribute to confirm whether the referencing document has a superset of the vocabulary modules in the referenced document. If one or both of the document instances are constrained, the conref processor checks to confirm the compatibility of the constraints.

Conref processors take into account whether constraints are specified as strong. For strong constraints, the following rules apply:

Conref pull

For each vocabulary module used by both document types, the module in the document type that contains the referencing element must be less (or equally) constrained than the same module in the document type that contains the referenced element. For example, if each document type uses the highlighting domain module, that module must be less (or equally) constrained in the document type that contains the referencing element.

Conref push

For each vocabulary module used by both document types, the module in the document type that contains the referencing element must be more (or equally) constrained than the same module in the document type that contains the referenced element. For example, if each document type uses the highlighting domain module, that module must be more (or equally) constrained in the document type that contains the referencing element.

Example: Conref pull and constraint compatibility

The following table contains scenarios where conref pull occurs between constrained and unconstrained document instances. It assumes that the processor is **not** configured to treat all constraints as strong constraints.

Values of @domains attribute in document type that contains the referencing element	Values of @domains attribute in document type that contains the referenced element	Resolution	Comments
(topic)	(topic shortdescReq-c)	Allowed	The content model of the referenced topic is more constrained than the referencing topic.
s(topic shortdescReq-c)	(topic)	Prevented	The constraint is specified as a strong constraint, and the content model of the referenced topic is less constrained than the referencing topic.
(topic shortdescReq-c)	(topic)	Allowed	Although the content model of referenced topic is less constrained than the referencing topic, this is a weak constraint and so permitted.
(topic task) (topic hi-d) (topic hi-d basicHighlightingDomain-c)	(topic simpleSection-c) (topic task) (topic task simpleStep-c)	Allowed	The referenced topic has a subset of the vocabulary modules that are integrated into the document-type shell for the referencing topic. Both document types

Values of @domains attribute in document type that contains the referencing element	Values of @domains attribute in document type that contains the referenced element	Resolution	Comments
			integrate constraints, but for modules used in both document types, the referencing topic is less constrained than the referenced topic.
(topic hi-d) (topic simpleSection-c) s(topic simpleP-c)	(topic simpleSection-c) (topic task) (topic hi-d) (topic hi-d) basicHighlightingDomain-c)	Prevented	The referencing document has constraints that are not present in the referenced document, including a strong constraint applied to the <p> element.

Example: Conref push and constraint compatibility

The following table contains scenarios where conref push occurs between constrained and unconstrained document instances. It assumes that the processor has **not** been configured to treat all constraints as strong constraints.

Values of @domains attribute in document type that contains the referencing element	Values of @domains attribute in document type that contains the referenced element	Resolution	Comments
(topic)	(topic shortdescReq-c)	Allowed	Although the content model of the referenced topic is more constrained than the referencing topic, this is a weak constraint and so permitted.
(topic)	s(topic shortdescReq-c)	Prevented	The constraint is specified as a strong constraint, and the content model of the referenced topic is more constrained than the referencing topic.
(topic shortdescReq-c)	(topic)	Allowed	The content model of the referencing topic is more constrained than the referenced topic.

Values of @domains attribute in document type that contains the referencing element	Values of @domains attribute in document type that contains the referenced element	Resolution	Comments
(topic task) (topic hi-d) (topic hi-d basicHighlightingDomain-c)	(topic simpleSection-c) (topic task) (topic task simpleStep-c)	Allowed	The referenced topic has a subset of the vocabulary modules that are integrated into the document-type shell for the referencing topic. For modules used in both document types, the referenced topic is more constrained than the referencing topic, but this is a weak constraint and so permitted.
(topic simpleSection-c) (topic task) (topic hi-d) (topic hi-d basicHighlightingDomain-c)	(topic hi-d) (topic simpleSection-c) s(topic simpleP-c)	Prevented	For the common topic module, the referenced document has more constraints than the referencing document, including a strong constraint applied to the <p> element.

Examples: Constraints

This section of the specification contains examples and scenarios. They illustrate a variety of ways that constraints can be used; they also provide examples of the DTD coding requirements for constraints and how constraints are integrated into document-type shells.

Example: Redefine the content model for the <topic> element

In this scenario, an information architect for Acme, Incorporated wants to redefine the content model for the topic document type. She wants to omit the <abstract> element and make the <shortdesc> element required; she also wants to omit the <related-links> element and disallow topic nesting.

1. She creates a .mod file using the following naming conventions: *qualifierTagNameConstraint.mod*, where *qualifier* is a string that describes the constraint, and *TagName* is the element type name with an initial capital. Her constraint module is named *acme-TopicConstraint.mod*.
2. She adds the following content to *acme-TopicConstraint.mod*:

```

<!-- ===== -->
<!--           CONSTRAINED TOPIC ENTITIES           -->
<!-- ===== -->

<!-- Declares the entity for the constraint module and defines -->
<!-- its contribution to the @domains attribute. -->

```

```

<!ENTITY topic-constraints
  "(topic basic-Topic-c)"
>

<!-- Declares the entities referenced in the constrained content -->
<!-- model. -->

<!ENTITY % title          "title">
<!ENTITY % titlealts     "titlealts">
<!ENTITY % shortdesc     "shortdesc">
<!ENTITY % prolog        "prolog">
<!ENTITY % body          "body">

<!-- Defines the constrained content model for <topic>. -->

<!ENTITY % topic.content
          "(%title;),
          (%titlealts;)?,
          (%shortdesc;),
          (%prolog;)?,
          (%body;)?)"
>

```

3. She then integrates the constraint module into her document-type shell for topic by adding the following section above the "TOPIC ELEMENT INTEGRATION" comment:

```

<!-- ===== -->
<!--          CONTENT CONSTRAINT INTEGRATION          -->
<!-- ===== -->

<!ENTITY % topic-constraints-c-def
  PUBLIC "-//ACME//ELEMENTS DITA Topic Constraint//EN"
  "acme-TopicConstraint.mod">
%topic-constraints-c-def;

```

4. She then adds the constraint to the list of domains and constraints that need to be included in the value of the @domains attribute for <topic>:

```

<!-- ===== -->
<!--          DOMAINS ATTRIBUTE OVERRIDE          -->
<!-- ===== -->

<!ENTITY included-domains
          "&hi-d-att;
          &ut-d-att;
          &indexing-d-att;
          &topic-constraints;"
>

```

5. After updating the catalog.xml file to include the new constraints file, her work is done.

Example: Constrain attributes for the <section> element

In this scenario, an information architect wants to redefine the attributes for the <section> element. He wants to make the @id attribute required and omit the @spectitle attribute.

1. He creates a .mod file named idRequiredSectionConstraint.mod, where "idRequired" is a string that characterizes the constraint.
2. He adds the following content to idRequiredSectionConstraint.mod:

```

<!-- ===== -->
<!--          CONSTRAINED TOPIC ENTITIES          -->

```

```

<!-- ===== -->
<!ENTITY section-constraints
  "(topic idRequired-section-c)"
>

<!-- Declares the entities referenced in the constrained content -->
<!-- model. -->
<!ENTITY % conref-atts
  'conref CDATA #IMPLIED
  conrefend CDATA #IMPLIED
  conaction (mark|pushafter|pushbefore|pushreplace|-dita-use-conref-
target) #IMPLIED
  conkeyref CDATA #IMPLIED' >
<!ENTITY % filter-atts
  'props CDATA #IMPLIED
  platform CDATA #IMPLIED
  product CDATA #IMPLIED
  audience CDATA #IMPLIED
  otherprops CDATA #IMPLIED
  %props-attribute-extensions;' >
<!ENTITY % select-atts
  '%filter-atts;
  base CDATA #IMPLIED
  %base-attribute-extensions;
  importance (default|deprecated|high|low|normal|obsolete|optional|
recommended|required|urgent|-dita-use-conref-target)
#IMPLIED
  rev CDATA #IMPLIED
  status (changed|deleted|unchanged|-dita-use-conref-target)
#IMPLIED' >
<!ENTITY % localization-atts
  'translate (no|yes|-dita-use-conref-target) #IMPLIED
  xml:lang CDATA #IMPLIED
  dir (lro|ltr|rlo|rtl|-dita-use-conref-target) #IMPLIED' >

<!-- Declares the constrained content model. Original definition -->
<!-- included %univ-atts;, spectitle, and outputclass; now includes-->
<!-- individual pieces of univ-atts, to make ID required. -->

<!ENTITY % section.attributes
  "id CDATA #REQUIRED
  %conref-atts;
  %select-atts;
  %localization-atts;
  outputclass CDATA #IMPLIED">

```



Note: The information architect had to declare all the parameter entities that are referenced in the redefined attributes for <section>. If he did not do so, none of the attributes that are declared in the %conref-atts;, %select-atts;, or %localization-atts; parameter entities would be available on the <section> element. Furthermore, since the %select-atts; parameter entity references the %filter-atts; parameter entity, the %filter-atts; must be declared and it must precede the declaration for the %select-atts; parameter entity. The %props-attribute-extensions; and %base-attribute-extensions; parameter entities do not need to be declared in the constraint module, because they are declared in the document-type shells before the inclusion of the constraint module.

3. He then integrates the constraint module into the applicable document-type shells and adds it to his catalog.xml file.

Example: Constrain a domain module

In this scenario, an information architect wants to use only a subset of the elements defined in the highlighting domain. She wants to use `` and `<i>`, but not ``, `<u>`, `<sup>`, `<sub>`, `<tt>`, or `<u>`. She wants to integrate this constraint into the document-type shell for task.

1. She creates `reducedHighlightingDomainConstraint.mod`, where "reduced" is a string that characterizes the constraint.
2. She adds the following content to `reducedHighlightingDomainConstraint.mod`:

```
<!-- ===== -->
<!--          CONSTRAINED HIGHLIGHTING DOMAIN ENTITIES          -->
<!-- ===== -->

<!ENTITY HighlightingDomain-constraints
  "(topic hi-d basic-HighlightingDomain-c)"
>

<!ENTITY % HighlightingDomain-c-ph      "b | i"                >
```

3. She then integrates the constraint module into her company-specific, document-type shell for the task topic by adding the following section directly before the "DOMAIN ENTITY DECLARATIONS" comment:

```
<!-- ===== -->
<!--          DOMAIN CONSTRAINT INTEGRATION          -->
<!-- ===== -->

<!ENTITY % HighlightingDomain-c-dec
  PUBLIC "-//ACME//ENTITIES DITA Highlighting Domain Constraint//EN"
  "acme-HighlightingDomainConstraint.mod"
>%basic-HighlightingDomain-c-dec;
```

4. In the "DOMAIN EXTENSIONS" section, she replaces the parameter entity for the highlighting domain with the parameter entity for the constrained highlighting domain:

```
<!ENTITY % ph
  "ph |
   %HighlightingDomain-c-ph; |
   %sw-d-ph; |
   %ui-d-ph;
  ">
```

5. She then adds the constraint to the list of domains and constraints that need to be included in the value of the `@domains` attribute for `<task>`:

```
<!-- ===== -->
<!--          DOMAINS ATTRIBUTE OVERRIDE          -->
<!-- ===== -->

<!ENTITY included-domains
  "&task-att;
   &hi-d-att;
   &indexing-d-att;
   &pr-d-att;
   &sw-d-att;
   &ui-d-att;
   &taskbody-constraints;
   &HighlightingDomain-constraints;
  "
>
```

6. After updating the `catalog.xml` file to include the new constraints file, her work is done.

Example: Replace a base element with the domain extensions

In this scenario, an information architect wants to remove the <ph> element but allow the extensions of <ph> that exist in the highlighting, programming, software, and user interface domains.

1. The information architect creates an entities file named `noPhConstraint.ent`, where "no" is a qualifier string that characterizes the constraint.
2. The information architect adds the following content to `noPhConstraint.ent`:

```
<!-- ===== -->
<!--          CONSTRAINED HIGHLIGHTING DOMAIN ENTITIES          -->
<!-- ===== -->

<!ENTITY ph-constraints
  "(topic noPh-ph-c)"
>
```



Note: Because the highlighting and programming domains cannot be generalized without the <ph> element, this entity must be defined so that there is a separate parenthetical expression that can be included in the @domains attribute for the topic.

3. The information architect then integrates the constraint module into a document-type shell for concept by adding the following section above the "TOPIC ELEMENT INTEGRATION" comment:

```
<!-- ===== -->
<!--          CONTENT CONSTRAINT INTEGRATION          -->
<!-- ===== -->

<!ENTITY % noPh-ph-c-def
  PUBLIC "-//ACME//ELEMENTS DITA Ph Constraint//EN"
  "acme-PhConstraint-constraints" "noPhConstraint.ent">
%noPh-ph-c-def;
```

4. In the "DOMAIN EXTENSIONS" section, the information architect removes the reference to the <ph> element:

```
<!-- Removed "ph | " so as to make <ph> not available, only the domain extensions. -->
>
<!ENTITY % ph
  "%pr-d-ph; |
  %sw-d-ph; |
  %ui-d-ph;
  ">
```

5. She then adds the constraint to the list of domains and constraints that need to be included in the value of the @domains attribute:

```
<!-- ===== -->
<!--          DOMAINS ATTRIBUTE OVERRIDE          -->
<!-- ===== -->

<!ENTITY included-domains
  "&concept-att;
  &hi-d-att;
  &indexing-d-att;
  &pr-d-att;
  &sw-d-att;
  &ui-d-att;
  &ph-constraint;
  "
>
```

6. After updating the `catalog.xml` file to include the new constraints file, the information architect's work is done.

Example: Apply multiple constraints to a single document-type shell

You can apply multiple constraints to a single document-type shell. However, there can be only one constraint for a given element or domain.

Here is a list of constraint modules and what they do:

File name	What it constrains	Details	Contribution to the @domains attribute
example-TopicConstraint.mod	<topic>	<ul style="list-style-type: none"> Removes <abstract> Makes <shortdesc> required Removes <related-links> Disallows topic nesting 	(topic basic-Topic-c)
example-SectionConstraint.mod	<section>	<ul style="list-style-type: none"> Makes @id required Removes @spectitle attribute 	(topic idRequired-section-c)
example-HighlightingDomainConstraint.mod	Highlighting domain	Reduces the highlighting domain elements to and <i>	(topic hi-d basic-HighlightingDomain-c)
example-PhConstraint.ent	<ph>	Removes the <ph> element	(topic noPh-ph-c)

All of these constraints can be integrated into a single document-type shell for <topic>, since they constrain distinct element types and domains. The constraint for the highlighting domain must be integrated before the "DOMAIN ENTITIES" section, but the order in which the other three constraints are listed does not matter.

Each constraint module provides a unique contribution to the @domains attribute. When integrated into the document-type shell for <topic>, the effective value of the domains attribute will include the following values, as well as values for any other modules that are integrated into the document-type shell:

```
(topic basic-Topic-c) (topic idRequired-section-c) (topic hi-d basic-HighlightingDomain-c) (topic noPh-ph-c)
```

Coding practices for DITA grammar files

This section collects all of the rules for creating modular DTD, RELAX NG, or XML Schema grammar files to represent DITA document types, specializations, and constraints.

Recognized XML-document grammar mechanisms

The DITA standard recognizes three XML-document grammar mechanisms by which conforming DITA vocabulary modules and document types can be constructed: document type declarations (DTDs), XML Schema declarations (XSDs), and RELAX NG grammars.

This specification defines implementation requirements for all of these document grammar mechanisms. The OASIS DITA Technical Committee recognizes that other XML grammar languages might provide similar modularity and extensibility mechanisms. However, because the Technical Committee has not yet defined implementation requirements for those languages, their conformance cannot be determined.

Of these three document grammar mechanisms, RELAX NG grammars offer the easiest-to-use syntax and the most precise constraints. For this reason, the RELAX NG definitions of the standard DITA vocabularies are the normative versions. The DTD and XSD versions shipped by OASIS are generated from the RELAX NG version using open source tools.

Normative versions of DITA grammar files

The OASIS DITA Technical Committee uses the RELAX NG XML syntax for the normative versions of the XML grammar files that comprise the DITA release.

The DITA Technical Committee chose the RELAX NG XML syntax for the following reasons:

Easy use of foreign markup

The DITA grammar files maintained by OASIS depend on this feature of RELAX NG in order to capture metadata about document-type shells and modules; such metadata is used to generate the DTD- and XSD-based versions of the grammar files.

The foreign vocabulary feature also can be used to include Schematron rules directly in RELAX NG grammars. Schematron rules can check for patterns that either are not expressible with RELAX NG directly or that would be difficult to express.

RELAX NG `<div>` element

This general grouping element allows for arbitrary organization and grouping of patterns within grammar documents. Such grouping tends to make the grammar documents easier to work with, especially in XML-aware editors. The use or non-use of the RELAX NG `<div>` element does not affect the meaning of the patterns that are defined in a RELAX NG schema.

Capability of expressing precise restrictions

RELAX NG is capable of expressing constraints that are more precise than is possible with either DTDs or XSDs. For example, RELAX NG patterns can be context specific such that the same element type can allow different content or attributes in different contexts.

If you plan to generate DTD- or XSD-based modules from RELAX NG modules, avoid RELAX NG features that cannot be translated into DTD or XSD constructs. When RELAX NG is used directly for DITA document validation, the document-type shells for those documents can integrate constraint modules that use the full power of RELAX NG to enforce constraints that cannot be enforced by DTDs or XSDs. The

grammar files provided by the OASIS DITA Technical Committee do not use any features of RELAX NG that cannot be translated into equivalent DTD or XSD constructs.

The DITA use of RELAX NG depends on the *RELAX NG DTD Compatibility* specification, which provides a mechanism for defining default attribute values and embedded documentation. Processors that use RELAX NG for DITA documents in which required attributes (for example, the `@domains` and `@class` attributes) are not explicitly present must implement the DTD compatibility specification in order to get default attribute values.

DTD coding requirements

This section explains how to implement DTD based document-type shells, specializations, and constraints.

DTD: Overview of coding requirements

DITA coding practices for DTDs rely heavily on entities to implement specialization and constraints. As such, an understanding of entities is critical when working with DTD document-type shells, vocabulary modules, or constraint modules.

Entities can be defined multiple times within a single document type, but only the first definition is effective. How entities work shapes DTD coding practices. The following list describes a few of the more important entities that are used in DITA DTDs:

Elements defined as entities

In DITA DTDs, every element is defined as an entity. When elements are added to a content model, they are added using the entity. This enables extension with domain specializations. For example, the entity `%ph;` usually just means "the ph element", but can be (pre)defined in a document-type shell to mean "ph plus several elements from the highlighting domain". Because the document-type shell places that entity definition before the usual definition, every element that included `%ph;` in its content model now includes `<ph>` plus every phrase specialization in the highlighting domain.

Content models defined as entities

Every element in a DITA DTD defines its content model using an entity. For example, rather than directly setting what is allowed in `<ph>`, that element sets its content model to `%ph.content;;` that entity defines the actual content model. This is done to enable constraints; a constraint module can (pre)define the `%ph.content;` model to remove selected elements.

Attribute sets defined as entities

Every element in a DITA DTD defines its attribute using an entity. For example, rather than directly defining attributes for `<ph>`, that element sets its attributes using the `%ph.attributes;` entity; that entity defines the actual attributes. As above, this is done to enable constraints; a constraint module can (pre)define the `%ph.attributes;` model to remove selected attributes.



Note: When constructing a constraint module or document-type shell, new entities are usually viewed as "redefinitions" because they redefine entities that already exist. However, these new definitions only work because they are added to a document-type shell before the existing definitions, which is why they are described here as (pre)definitions. Most topics about DITA DTDs, including others in this specification, will describe these overrides as redefinitions to ease understanding.

DTD: Coding requirements for document-type shells

A DTD-based document-type shell is organized into sections; each section contains entity declarations that follow specific coding rules.

The DTD-based approach to configuration, specialization, and constraints relies heavily upon parameter entities. Several of the parameter entities that are declared in document type shells contain references to other parameter entities. Because parameter entities must be declared before they are used, the order of the sections in a DTD-based document-type shell is significant.

DTD-based document-type shell contains the following sections:

1. *Topic [or map] entity declarations*
2. *Domain constraint integration*
3. *Domain entity declarations*
4. *Domain attributes declarations*
5. *Domain extensions*
6. *Domain attribute extensions*
7. *Topic nesting override*
8. *Domains attribute override*
9. *Content constraint integration*
10. *Topic [or map] element integration*
11. *Domain element integration*

Each of the sections in a DTD-based document-type shell follows a pattern. These patterns help ensure that the shell follows XML parsing rules for DTDs; they also establish a modular design that simplifies creation of new document-type shells. By convention, an `.ent` file extension is used to indicate files that define only parameter entities, while a `.mod` file extension is used to indicate files that define elements or constraints.

Topic [or map] entity declarations

This section declares and references an external parameter entity for each of the following:

- The top-level topic or map type that the document-type shell configures
- Any additional structural modules that are used by the document type shell

Each parameter entity (`.ent`) file contributes a domain token for structural topics or maps. The parameter entity is named `type-name-dec`.

For example, a document-type shell that integrates the `<concept>` specialization would include:

```
<!ENTITY % concept-dec
PUBLIC "-//OASIS//ENTITIES DITA 1.3 Concept//EN"
"concept.ent"
>%concept-dec;
```

Domain constraint integration

For each domain constraint module that is integrated into the document type shell, this section declares a parameter entity and references the constraint module file where the constraint is defined. The parameter entity is named `descriptorDomainName-c-dec`.

In the following example, the entity file for a constraint module that reduces the highlighting domain to a subset is included in a document type shell:

```
<!-- ===== -->
<!-- DOMAIN CONSTRAINT INTEGRATION -->
```

```
<!-- ===== -->
<!ENTITY % HighlightingDomain-c-dec
PUBLIC "-//ACME//ENTITIES DITA Highlighting Domain Constraint//EN"
"acme-HighlightingDomainConstraint.mod"
>%basic-HighlightingDomain-c-dec;
```

Domain entity declarations

For each element domain that is integrated into the document-type shell, this section declares a parameter entity and references the external entities file where the element domain is defined. The parameter entity is named *shortDomainName-dec*.

In the following example, the entity file for the highlighting domain is included in a document-type shell:

```
<!ENTITY % hi-d-dec PUBLIC
"//OASIS//ENTITIES DITA Highlight Domain//EN"
"highlightDomain.ent"
>%hi-d-dec;
```

Domain attributes declarations

For each attribute domain that is integrated into the document-type shell, this section declares a parameter entity and references the external entities file where the attribute domain is defined. The parameter entity is named *domainName-dec*.

In the following example, the entity file for the @deliveryTarget attribute domain is included in a document-type shell:

```
<!ENTITY % deliveryTargetAtt-d-dec
PUBLIC "-//OASIS//ENTITIES DITA 1.3 Delivery Target Attribute Domain//EN"
"deliveryTargetAttDomain.ent"
>%deliveryTargetAtt-d-dec;
```

Domain extensions

For each element that is extended by one or more domains, this section redefines the parameter entity for the element. These entities are used by later modules to define content models; redefining the entity adds domain specializations wherever the base element is allowed.

In the following example, the entity for the <pre> element is redefined to add specializations from the programming, software, and user interface domains:

```
<!ENTITY % pre
"pre |
%pr-d-pre; |
%sw-d-pre; |
%ui-d-pre;">
```

Domain attribute extensions

For each attribute domain that is integrated into the document-type shell, this section redefines the parameter entities for the attribute. It adds an extension to the parameter entity for the relevant attribute.

In the following example, the @props attribute is specialized to create the @new and @othernew attributes, while the @base attribute is specialized to create @newfrombase and @othernewfrombase attributes:

```
<!ENTITY % props-attribute-extensions
"%newAtt-d-attribute;
```

```

    %othernewAtt-d-attribute;">
<!ENTITY % base-attribute-extensions
    "%newfrombaseAtt-d-attribute;
    %othernewfrombaseAtt-d-attribute;">

```

Topic nesting override

For each topic type that is integrated into the document-type shell, this section specifies whether and how subtopics nest by redefining the *topic-type-info-types* entity. The definition is usually an OR list of the topic types that can be nested in the parent topic type. Use the literal root-element name, not the corresponding parameter entity. Topic nesting can be disallowed completely by specifying the `<no-topic-nesting>` element.

In the following example, the parameter entity specifies that `<concept>` can nest any number of `<concept>` or `<myTopicType>` topics, in any order:

```
<!ENTITY % concept-info-types "concept | myTopicType">
```

Domains attribute override

This section sets the effective value of the `@domains` attribute for the top-level document type that is configured by the document type shell. It redefines the *included-domains* entity to include the text entity for each domain, constraint, and structural specialization that is either included or reused in the document type shell.

In the following example, entities are included for both the troubleshooting specialization and the task specialization on which the troubleshooting specialization depends; for the highlighting and utilities element domains; for the *newAtt-d* attribute domain, and for the *noBasePre-c* constraint module:

```

<!ENTITY included-domains
    "&troubleshooting-att;
    &task-att;
    &hi-d-att;
    &ut-d-att;
    &newAtt-d-att;
    &noBasePre-c-ph;
    "
>

```



Note: Although parameter entities (entities that begin with "%") must be defined before they are referenced, text entities (entities that begin with "&") can be referenced before they are defined. This allows the *included-domains* entity to include the constraint entity, which is not defined until the constraint module is referenced later in the document type shell.

Content constraint integration

For each constraint module that is integrated into the document-type shell, this section declares and references the external module file where the constraint is defined. The parameter entity is named *constraintName-c-def*.

In the following example, the constraint module that constrains the content model for the `<taskbody>` element is integrated into the document-type shell for strict task:

```

<!ENTITY % strictTaskbody-c-def
    PUBLIC "-//OASIS//ELEMENTS DITA 1.3 Strict Taskbody Constraint//EN"
    "strictTaskbodyConstraint.mod"
>%strictTaskbody-c-def;

```


Topic [or map] element integration

For each structural module that is integrated into the document-type shell, this section declares a parameter entity and references the external module file where the structural module is defined. The parameter entity is named *structuralType-type*. The modules must be included in ancestry order, so that the parameter entities that are used in an ancestor module are available for use in specializations. When a structural module depends on elements from a vocabulary module that is not part of its ancestry, the module upon which the structural module has a dependency (and any ancestor modules not already included) should be included before the module with a dependency.

The following example declares and references the structural modules that are integrated into the document-type shell for troubleshooting:

```
<!ENTITY % topic-type
  PUBLIC "-//OASIS//ELEMENTS DITA 1.3 Topic//EN"
         "../..base/dtd/topic.mod"
>%topic-type;

<!ENTITY % task-type
  PUBLIC "-//OASIS//ELEMENTS DITA 1.3 Task//EN"
         "task.mod"
>%task-type;

<!ENTITY % troubleshooting-type
  PUBLIC "-//OASIS//ELEMENTS DITA 1.3 Troubleshooting//EN"
         "troubleshooting.mod"
>%troubleshooting-type;
```

Domain element integration

For each element domain that is integrated into the document-type shell, this section declares a parameter entity and references the external module file where the element domain is defined. The parameter entity is named *domainName-def*.

For example, the following code declares and references the parameter entity used for the highlighting domain:

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



Note: If a structural module depends on a domain, the domain module should be included before the structural module. This erases the boundary between the final two sections, but it is necessary to ensure that modules are embedded after their dependencies. Technically, the only solid requirement is that both domain and structural modules be declared after all other modules that they specialize from or depend on.

DTD: Coding requirements for element type declarations

This topic covers general coding requirements for defining element types in both structural and element-domain vocabulary modules. In addition, it covers how to create the `@domains` attribute contribution for these modules.

A vocabulary module that defines a structural or element domain specialization is composed of two files:

- An entity declaration (`.ent`) file, which declares the text entities that are used to integrate the vocabulary module into a document-type shell
- A definition module (`.mod`) file, which declares the element names, content models, and attribute lists for the element types that are defined in the vocabulary module

@domains attribute contribution

A domain declaration entity is used to construct the effective value of the @domains attribute for a map or topic type.

Text entity name

The name of the text entity is the structural type name or the domain abbreviation, followed by a hyphen ("-") and the literal `att`.

Text entity values

The value of the text entity is the @domains attribute contribution for the current module. See [domains attribute rules and syntax](#) on page 122 for details on how to construct this value.

For example, the @domains attribute contributions for the concept structural module and the highlighting domain module are constructed as follows.

- `<!ENTITY concept-att "(topic concept)">`
- `<!ENTITY hi-d-att "(topic hi-d)">`.

Element definitions

A structural or domain vocabulary module must contain a declaration for each element type that is named by the module. While the XML standard allows content models to refer to undeclared element types, the DITA standard does not permit this. All element types or attribute lists that are named within a vocabulary module must be declared in one of the following objects:

- The vocabulary module
- A base module of which the vocabulary module is a direct or indirect specialization
- (If the vocabulary module is a structural module) A required domain module

The following components make up a single element definition in a DITA DTD-based vocabulary module.

Element name entities

For each element type, there must be a parameter entity with a name that matches the element type name. The default value is the element type name. This parameter entity provides a layer of abstraction when setting up content models; it can be redefined in a document-type shell in order to create domain extensions or constraints. Element name entities for a single vocabulary module are typically grouped together at the top of the vocabulary module.

For example: `<!ENTITY % topichead "topichead">`

Content-model parameter entity

For each element type, there must be a parameter entity that defines the content model. The name of the parameter entity is `tagname.content`, and the value is the content model definition. Consistent use and naming of the `tagname.content` parameter entity enables the use of constraint modules to restrict the content model.

For example:

```
<!ENTITY % topichead.content
  "( (%topicmeta;)?,
    (%anchor; |
    %data.elements.incl; |
    %navref; |
    %topicref;)* )"
">
```

Attribute-list parameter entity

For each element type, there must be a parameter entity that declares the attributes that are available on the element. The name of the parameter entity is *tagname.attributes*, and the value is a list of the attributes that are used by the element type (except for `@class` and the attributes provided by the `global-atts` parameter entity). Consistent use and naming of the *tagname.attributes* parameter entity enables the use of constraint modules to restrict attributes.

For example:

```
<!ENTITY % topichead.attributes
"navtitle CDATA #IMPLIED
outputclass CDATA #IMPLIED
keys CDATA #IMPLIED
copy-to CDATA #IMPLIED
%topicref-atts;
%univ-atts;"
>
```

Element declaration

For each element type, there must be an element declaration that consists of a reference to the content-model parameter entity.

For example:

```
<!ELEMENT topichead %topichead.content;>
```

Attribute list declaration

For each element type, there must be an attribute list declaration that consists of a reference to the attribute-list parameter entity.

For example:

```
<!ATTLIST topichead %topichead.attributes;>
```

Specialization attribute declarations

A vocabulary module must define a `@class` attribute for every element that is declared in the module. The value of the attribute is constructed according to the rules in [class attribute rules and syntax](#) on page 121. The ATTLIST declaration for the `@class` attribute should also include a reference to the `global-atts` parameter entity.

For example, the ATTLIST definition for the `<topichead>` element (a specialization of the `<topicref>` element in the base map type) includes global attributes with an entity, then the definition of the `@class` attribute, as follows:

```
<!ATTLIST topichead %global-atts; class CDATA "+ map/topicref mapgroup-d/
topichead ">
```

DTD: Coding requirements for structural modules

A structural vocabulary module defines a new topic or map type as a specialization of a topic or map type.

Required topic and map element attributes

The topic or map element type must set the @DITAArchVersion attribute to the version of DITA in use, typically by referencing the arch-atts parameter entity. It must also set the @domains attribute to the included-domains entity. These attributes give processors a reliable way to check the architecture version and look up the list of domains available in the document type.

The following example shows how these attributes are defined for the <concept> element in DITA 1.3:

```
<!ATTLIST concept
  %concept.attributes;
  %arch-atts;
  domains CDATA "&included-domains;">
```

Controlling nesting in topic types

Specialized topics typically use a parameter entity to define what topic types are permitted to nest. While there are known exceptions described below, the following rules apply when using parameter entities to control nesting.

Parameter entity name

The name of the parameter entity is the topic element name plus the `-info-types` suffix.

For example, the name of the parameter entity for the concept topic is `concept-info-types`.

Parameter entity value

To set up default topic nesting rules, set the entity to the desired topic elements. The default topic nesting will be used when a document-type shell does not set up different rules.

For example, the following parameter entity sets up default nesting so that <concept> will nest only other <concept> topics:

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

As an additional example, the following parameter entity sets up a default that will not allow any nesting:

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

Default topic nesting in a structural module often set up to use the `%info-types;` parameter entity rather than using a specific element. When this is done consistently, a shell that includes multiple structural modules can set common nesting rules for all topic types by setting `%info-types;` entity. The following example shows a structural module that uses `%info-types;` for default topic nesting:

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

Content model of the root element

The last position in the content model defined for the root element of a topic type *SHOULD* be the `topic-type-info-types` parameter entity. A document-type shell then can control how topics are allowed to nest for this specific topic type by redefining the `topic-type-info-types` entity for each

topic type. If default nesting rules reference the `info-types` parameter entity, a shell can efficiently create common nesting rules by redefining the `info-types` entity.

For example, with the following content model defined for `<concept>`, a document-type shell that uses the concept specialization can control which topics are nested in `<concept>` by redefining the `concept-info-types` parameter entity:

```
<!ENTITY % concept.content
  "(%title;),
  (%titlealts;)?,
  (%abstract; | %shortdesc;)?,
  (%prolog;)?,
  (%conbody;)?,
  (%related-links;)?,
  (%concept-info-types;)*)"
>
```

In rare cases, it may not be desirable to control topic nesting with a parameter entity. For example:

- If a specialized topic type should never allow any nested topics, regardless of context, it can be defined without any entity or any nested topics.
- If a specialized topic type should only ever allow specific nesting patterns, such as allowing only other topic types that are defined in the same module, it can nest those topics directly in the same way that other nested elements are defined.

DTD: Coding requirements for element domain modules

The vocabulary modules that define element domains have an additional coding requirement. The entity declaration file must include a parameter entity for each element that the domain extends.

Parameter entity name

The name of the parameter entity is the abbreviation for the domain, followed by a hyphen ("-"), and the name of the element that is extended.

Parameter entity value

The value of the parameter entity is a list of the specialized elements that can occur in the same locations as the extended element. Each element must be separated by the vertical line (|) symbol.

Example

Because the highlighting domain extends the `<ph>` element, the entity declaration file for that domain must include a parameter entity corresponding to the `<ph>` element. The name of the entity uses the short name of the domain (`hi-d`) followed by the name of the base element. The value includes each specialization of `<ph>` in the domain.

```
<!ENTITY % hi-d-ph "b | u | i | line-through | overline | tt | sup | sub">
```

DTD: Coding requirements for attribute domain modules

The vocabulary modules that define attribute domains have additional coding requirements. The module must include a parameter entity for the new attribute, which can be referenced in document-type shells, as well as a text entity that specifies the contribution to the `@domains` attribute for the attribute domain.

An attribute domain's name is the name of the attribute plus "Att". For example, for the attribute named "deliveryTarget" the attribute domain name is "deliveryTargetAtt". The attribute domain name is used to construct entity names for the domain.

Parameter entity name and value

The name of the parameter entity is the attribute domain name, followed by the literal "-d-attribute". The value of the parameter entity is a DTD declaration for the attribute.

Text entity name and value

The text entity name is the attribute domain name, followed by the literal `-d-Att`. The value of the text entity is the `@domains` attribute contribution for the module; see [domains attribute rules and syntax](#) on page 122 for details on how to construct this value.

Example

The `@deliveryTarget` attribute can be defined in a vocabulary module using the following two entities.

```
<!ENTITY % deliveryTargetAtt-d-attribute
  "deliveryTarget CDATA #IMPLIED"
>

<!ENTITY deliveryTargetAtt-d-att "a(props deliveryTarget)" >
```

DTD: Coding requirements for constraint modules

A structural constraint module defines the constraints for a map or topic element type. A domain constraint module defines the constraints for an element or attribute domain.

Structural constraint modules

Structural constraint modules have the following requirements:

`@domains` contribution entity name and value

The constraint module should contain a declaration for a text entity with the name `tagname-constraints`, where `tagname` is the name of the element type to which the constraints apply. The value of the text entity is the `@domains` attribute contribution for the module; see [domains attribute rules and syntax](#) on page 122 for details on how to construct this value.

For example, the following text entity provides the declaration for the strict task constraint that is shipped with the DITA standard.

```
<!ENTITY taskbody-constraints
  "(topic task strictTaskbody-c)"
>
```

The *tagname.attributes* parameter entity

When the attribute set for an element is constrained, there must be a declaration of the *tagname.attributes* parameter entity that defines the constrained attributes.

For example, the following parameter entity defines a constrained set of attributes for the `<note>` element that removes most of the values defined for `@type`, and also removes `@spectitle` and `@othertype`:

```
<!ENTITY % note.attributes
    "type (attention | caution | note ) #IMPLIED
    %univ-atts;
    outputclass CDATA #IMPLIED">
```

The *tagname.content* parameter entity

When the content model for an element is constrained, there must be a declaration of the *tagname.content* parameter entity that defines the constrained content model.

For example, the following parameter entity defines a more restricted content model for `<topic>`, in which the `<shortdesc>` element is required.

```
<!ENTITY % topic.content
    "( (%title;),
    (%titlealts;)?,
    (%shortdesc;),
    (%prolog;)?,
    (%body;)?,
    (%topic-info-types;)* )"
>
```

Domain constraint modules

Domain constraint modules have the following requirements:

`@domains` contribution entity name and value

The constraint module should contain a declaration for a text entity with the name *domainDomain-constraints*, where *domain* is the name of the domain to which the constraints apply, for example, "Highlighting" or "Programming". The value of the text entity is the `@domains` attribute contribution for the module; see [domains attribute rules and syntax](#) on page 122 for details on how to construct this value.

For example, the following text entity provides the declaration for a constraint module that restricts the highlighting domain:

```
<!ENTITY HighlightingDomain-constraints
    "(topic hi-d basic-HighlightingDomain-c) "
>
```

Parameter entity

When the set of extension elements are restricted, there must be a parameter entity that defines the constrained content model.

For example, the following parameter entity restricts the highlighting domain to `` and `<i>`:

```
<!ENTITY % HighlightingDomain-c-ph    "b | i" >
```


Constraining to replace a base element with domain extensions

When element domains are used to extend a base element, those extensions can be used to replace the base element. This form of constraint is done inside the document-type shell.

Within a document-type shell, *domain extensions* are implemented by declaring an entity for a base element. The value of the entity can omit any base element types from which the other element types that are listed are specialized. Omitting a base type constitutes a form of constraint; as with any other constraint, this form of constraint must contribute a token to the @domains attribute. That token can be defined in a module file (which does not define any other entities or values), or the token can be placed directly into the document-type shell definition for the included-domains entity.

In the following example, the <pre> base type is removed from the entity declaration, effectively allowing only specializations of <pre> but not <pre> itself. This omission would require the use of a @domains contribution token within the included-domains entity.

```
<!ENTITY % pre
  "%pr-d-pre; |
  %sw-d-pre; |
  %ui-d-pre;">
```

RELAX NG coding requirements

This section explains how to implement RELAX NG based document-type shells, specializations, and constraints.

RELAX NG: Overview of coding requirements

RELAX NG modules are self-integrating, which means that they automatically add to the content models and attribute sets they extend. This means that information architects do not have much work to do when assembling vocabulary modules and constraints into document type shells.

In addition to simplifying document-type shells, the self-integrating aspect of RELAX NG results in the following coding practices:

- Each specialized vocabulary module consists of a single file, unlike the two required for DTDs.
- Domain modules directly extend elements, unlike DTDs, which rely on an extra file and extensions within the document-type shell.
- Constraint modules directly include the modules that they extend, which means that just by referencing a constraint module, the document-type shell gets everything it needs to both define and constrain a vocabulary module.

RELAX NG grammars for DITA document-type shells, vocabulary modules, and constraint modules *MAY* do the following:

- Use the <a:documentation> element anywhere that foreign elements are allowed by RELAX NG. The <a:documentation> element refers to the <documentation> element type from the <http://relaxng.org/ns/compatibility/annotations/1.0> as defined by the DTD compatibility specification. The prefix "a" is used by convention.
- Use <div> to group pattern declarations.
- Include embedded Schematron rules or any other foreign vocabulary. Processors *MAY* ignore any foreign vocabularies within DITA grammars that are not in the <http://relaxng.org/ns/compatibility/annotations/1.0> or <http://dita.oasis-open.org/architecture/2005/> namespaces.

Syntax for RELAX NG grammars

The RELAX NG specification defines two syntaxes for RELAX NG grammars: the XML syntax and the compact syntax. The two syntaxes are functionally equivalent, and either syntax can be reliably converted into the other by using, for example, the open-source Trang tool.

DITA practitioners can author DITA modules using one RELAX NG syntax, and then use tools to generate modules in the other syntax. The resulting RELAX NG modules are conforming if there is a one-to-one file correspondence. Conforming RELAX NG-based DITA modules *MAY* omit the annotations and foreign elements that are used in the OASIS grammar files to enable generation of other XML grammars, such as DTDs and XML Schema. When such annotations are used, conversion from one RELAX NG syntax to the other might lose the information, as processors are not required to process the annotations and information from foreign vocabularies.

The DITA coding requirements are defined for the RELAX NG XML syntax. Document type shells, vocabulary modules, and constraint modules that use the RELAX NG compact syntax can use the same organization requirements as those defined for the RELAX NG XML syntax.

RELAX NG: Coding requirements for document-type shells

A document-type shell integrates one or more topic type or map type modules, zero or more domain modules, and zero or more constraint modules.

Because RELAX NG modules are self-integrating, document-type shells only need to include vocabulary modules. Unlike DTDs, there is no separate specification required in order to integrate domain and nested topic elements into the base content models.

Root element declaration

Document type shells use the RELAX NG start declaration to specify the root element of the document type. The `<start>` element defines one root element, using a reference to a `tagname.element` pattern.

For example:

```
<div>
  <a:documentation>ROOT ELEMENT DECLARATION</a:documentation>
  <start combine="choice">
    <ref name="topic.element"/>
  </start>
</div>
```

DITA domains attribute

The document-type shell must list the domain or structural modules that are named as dependencies in the `@domains` attribute value. Unlike DTDs, a default value for `@domains` cannot automatically be constructed using RELAX NG facilities. Instead, the values used to construct `@domains` are taken from each vocabulary and constraint module, in addition to any domains contributions based on constraints implemented within the shell.

For example:

```
<div>
  <a:documentation>DOMAINS ATTRIBUTE</a:documentation>
  <define name="domains-att">
    <optional>
      <attribute name="domains"
        a:defaultValue="(topic hazard-d)
          (topic hi-d)
          (topic indexing-d)
          (topic ut-d)">
```

```

                                a(props deliveryTarget) "
    />
  </optional>
</define>
</div>

```

Content constraint integration

The document-type shell must include any constraint modules. Constraint modules include the module they override and override the patterns that they constrain directly in the constraint module itself. Any module that is constrained in this section (including the base topic or map modules) will be left out of the following section.

For example, when the following constraint is included for the task module, the task module itself will *not* be included in the shell; the task module itself is included within `strictTaskbodyConstraintMod.rng`:

```

<div>
<a:documentation>CONTENT CONSTRAINT INTEGRATION</a:documentation>
  <include href="strictTaskbodyConstraintMod.rng">
    <define name="task-info-types">
      <ref name="task.element"/>
    </define>
  </include>
</div>

```

Module inclusions

The document-type shell must include any unconstrained domain or structural module. If the top-level map or topic type is unconstrained, it is also included in this section.

For example:

```

<div>
  <a:documentation>MODULE INCLUSIONS</a:documentation>
  <include href="topicMod.rng"/>
  <include href="highlightDomainMod.rng"/>
  <include href="utilitiesDomainMod.rng"/>
  <include href="indexingDomainMod.rng"/>
  <include href="hazardstatementDomainMod.rng"/>
</div>

```

Constraining domains in the shell

Domains can be constrained to disallow some extension elements without the use of a separate module file. This is done by overriding the base type pattern within the reference to the domain module. In this case, the constraint represented by the pattern redefinition still must be declared in the `@domains` attribute; the `@domains` contribution should be documented in the document-type shell with the constraint. There is not a designated section of the document-type shell for this type of constraint; it can be placed either in [Content constraint integration](#) on page 0 or [Module inclusions](#) on page 0 .

The following example demonstrates the portion of a document type shell that includes the highlight domain module while directly constraining that module to remove the `<u>` element. The `<ditaarch:domainsContribution>` element is used to document the `@domains` contribution for this constraint.

```

<div>
  <a:documentation>MODULE INCLUSIONS</a:documentation>
  <include href="topicMod.rng"/>

```

```

<include href="hazardstatementDomainMod.rng"/>
<include href="highlightDomainMod.rng">
  <ditaarch:domainsContribution
    >(topic hi-d-noUnderline-c)</ditaarch:domainsContribution>
  <define name="u">
    <notAllowed></notAllowed>
  </define>
</include>
<include href="indexingDomainMod.rng"/>
<include href="utilitiesDomainMod.rng"/>
</div>

```

ID-defining element overrides

This section must declare any element in the document type that uses an @id attribute with an XML data type of "ID". This declaration is required in order to avoid errors from RELAX NG parsers that would otherwise complain about different uses of the @id attribute.

Typically, this section lists only a few elements, such as topic types or the <anchor> element, but it could include specializations that constrain @id. In addition, foreign vocabularies require you to include exclusions for the namespaces used by those domains.

For example, this section declares that <topic> and <task> use an @id attribute with an XML data type of ID, along with any elements in the SVG or MathML namespaces. Each of these is excluded from the "any" pattern by placing them within the <except> rule as shown below:

```

<div>
  <a:documentation> ID-DEFINING-ELEMENT OVERRIDES </a:documentation>
  <define name="any">
    <zeroOrMore>
      <choice>
        <ref name="idElements"/>
        <element>
          <anyName>
            <except>
              <name>topic</name>
              <name>task</name>
              <nsName ns="http://www.w3.org/2000/svg"/>
              <nsName ns="http://www.w3.org/1998/Math/MathML"/>
            </except>
          </anyName>
          <zeroOrMore>
            <attribute>
              <anyName/>
            </attribute>
          </zeroOrMore>
          <ref name="any"/>
        </element>
        <text/>
      </choice>
    </zeroOrMore>
  </define>
</div>

```

RELAX NG: Coding requirements for element type declarations

Structural and domain vocabulary modules have the same coding requirements for element type declarations. Each RELAX NG vocabulary module consists of a single module file.

Element definitions

A structural or element-domain vocabulary module must contain a declaration for each specialized element type that is named in the module. While the XML standard allows content models to refer to undeclared element types, all element types that are named in content models or attribute list declarations within a vocabulary module must have a RELAX NG element declaration. The RELAX NG element declaration can occur in one of the following:

- The vocabulary module
- A base module of which the vocabulary module is a direct or indirect specialization
- (If the vocabulary module is a structural module) A required domain or structural module

The element type patterns are organized into the following sections:

Element type name patterns

For each element type that is declared in the vocabulary module, there must be a pattern whose name is the element type name and whose content is a reference to the element type *tagname.element pattern*. For example:

```
<div>
  <a:documentation>ELEMENT TYPE NAME PATTERNS</a:documentation>
  <define name="b">
    <ref name="b.element"/>
  </define>
  <!-- ... -->
</div>
```

The element-type name pattern provides a layer of abstraction that facilitates redefinition. The element-type name patterns are referenced from content model and domain extension patterns. Specialization modules can re-declare the patterns to include specializations of the type, allowing the specialized types in all contexts where the base type is allowed.

The declarations can occur in any order.

Common content-model patterns

Structural and element-domain modules can include a section that defines the patterns that contribute to the content models of the element types that are defined in the module.

Common attribute sets

Structural and element-domain modules can include a section that defines patterns for attribute sets that are common to one or more of the element types that are defined in the module.

Element type declarations

For each element type that is declared in the vocabulary module, the following set of patterns must be used to define the content model and attributes for the element type. Each set of patterns is typically grouped within a `<div>` element for clarity.

- *tagname.content* defines the complete content model for the element *tagname*. The content model pattern can be overridden in constraint modules to further constrain the content model for the element type.

- `tagname.attributes` defines the complete attribute list for the element `tagname`, except for `@class` and the attributes provided by the `global-atts` pattern. The attribute list declaration can be overridden in constraint modules to further constrain the attribute list for the element type.
- `tagname.attlist` is an additional attribute list pattern with a `@combine` attribute set to the value "interleave". This pattern contains only a reference to the `tagname.attributes` pattern.
- `tagname.element` is the actual element type definition. It contains an `<element>` element whose `@name` value is the element type name and whose content is a reference to the `tagname.content` and `tagname.attlist` patterns. In OASIS grammar files, the `@longName` attribute in the DITA architecture namespace is also used to help enable generation of DTD and XSD grammar files.

The following example shows the declaration for the `<topichead>` element, including the definition for each pattern described above.

```
<div>
  <a:documentation>LONG NAME: Topic Head</a:documentation>
  <define name="topichead.content">
    <optional>
      <ref name="topicmeta"/>
    </optional>
    <zeroOrMore>
      <choice>
        <ref name="anchor"/>
        <ref name="data.elements.incl"/>
        <ref name="navref"/>
        <ref name="topicref"/>
      </choice>
    </zeroOrMore>
  </define>
  <define name="topichead.attributes">
    <optional>
      <attribute name="navtitle"/>
    </optional>
    <optional>
      <attribute name="outputclass"/>
    </optional>
    <optional>
      <attribute name="keys"/>
    </optional>
    <optional>
      <attribute name="copy-to"/>
    </optional>
    <ref name="topicref-atts"/>
    <ref name="univ-atts"/>
  </define>
  <define name="topichead.element">
    <element name="topichead" ditaarch:longName="Topic head">
      <a:documentation>The <code><topichead></code> element provides a title-only entry in a
navigation map,
      as an alternative to the fully-linked title provided by the <code><topicref></code>
element.
      Category: Mapgroup elements</a:documentation>
      <ref name="topichead.attlist"/>
      <ref name="topichead.content"/>
    </element>
  </define>
  <define name="topichead.attlist" combine="interleave">
    <ref name="topichead.attributes"/>
  </define>
</div>
```

idElements pattern contribution

Element types that declare the `@id` attribute as type "ID", including all topic and map element types, must provide a declaration for the `idElements` pattern. This is needed to correctly configure the "any" pattern override in document-type shells and avoid errors from RELAX NG parsers. The declaration is specified with a `@combine` attribute set to the value "choice". For example:

```
<div>
  <a:documentation>LONG NAME: Map</a:documentation>
  <!-- ... -->
  <define name="idElements" combine="choice">
    <ref name="map.element"/>
  </define>
</div>
```

Specialization attribute declarations

A vocabulary module must define a `@class` attribute for every specialized element. This is done in a section at the end of each module that includes a `tagname.attlist` pattern for each element type that is defined in the module. The declarations can occur in any order.

The `tagname.attlist` pattern for each element defines that element's `@class` attribute, and also includes a reference to the `global-atts` attribute list pattern. `@class` is declared as an optional attribute; the default value is declared using the `@a:defaultValue` attribute, and the value of the attribute is constructed according to the rules in [class attribute rules and syntax](#) on page 121.

For example:

```
<define name="anchorref.attlist" combine="interleave">
  <ref name="global-atts"/>
  <optional>
    <attribute name="class"
      a:defaultValue="+ map/topicref mapgroup-d/anchorref "
    />
  </optional>
</define>
```

RELAX NG: Coding requirements for structural modules

A structural vocabulary module defines a new topic or map type as a specialization of a topic or map type.

All vocabulary and constraint modules must document their `@domains` attribute contribution. The value of the contribution is constructed according to the rules found in [domains attribute rules and syntax](#) on page 122. The OASIS grammar files use a `<domainsContribution>` element to document the contribution; this element is used to help enable generation of DTD and XSD grammar files. An XML comment or `<a:documentation>` element can also be used.

Required topic and map element attributes

The topic or map element type must reference the `arch-atts` pattern, which defines the `@DITAArchVersion` attribute in the DITA architecture namespace and sets the attribute to the version of DITA in use. In addition, the topic or map element type must reference the `domains-att` pattern, which will pull in a definition for the `@domains` attribute. These attributes give processors a reliable way to check the architecture version and list of available domains.

For example, the following definition references the `arch-atts` and `domains-att` patterns as part of the definition for the `<concept>` element.

```
<div>
  <a:documentation> LONG NAME: Concept </a:documentation>
  <!-- ... -->
  <define name="concept.attlist" combine="interleave">
    <ref name="concept.attributes"/>
    <ref name="arch-atts"/>
    <ref name="domains-att"/>
  </define>
  <!-- ... -->
</div>
```

Controlling nesting in topic types

Specialized topics typically define an `info-types` style pattern to specify default topic nesting. Document type shells can then control how topics are allowed to nest by redefining the pattern. While there are known exceptions described below, the following rules apply when using a pattern to control topic nesting.

Pattern name

The pattern name is the topic element name plus the suffix `-info-types`. For example, the `info-types` pattern for the `concept` topic type is `concept-info-types`.

Pattern value

To set up default topic nesting rules, set the pattern to the desired topic elements. The default topic nesting will be used when a document-type shell does not set up different rules. For example:

```
<div>
  <a:documentation>INFO TYPES PATTERNS</a:documentation>
  <define name="mytopic-info-types">
    <ref name="subtopic-01.element"/>
    <ref name="subtopic-02.element"/>
  </define>
  <!-- ... -->
</div>
```

If the topic does not permit nested topics by default, this pattern uses the `<empty>` element. For example:

```
<define name="learningAssessment-info-types">
  <empty/>
</define>
```

The `info-types` pattern can also be used to refer to common nesting rules across the document-type shell. For example:

```
<div>
  <a:documentation>INFO TYPES PATTERNS</a:documentation>
  <define name="mytopic-info-types">
    <ref name="info-types"/>
  </define>
  <!-- ... -->
</div>
```

Content model of the root element

In the declaration of the root element of a topic type, the last position in the content model *SHOULD* be the *topic-type-info-types* pattern. For example, the `<concept>` element places the pattern after `<related-links>`:

```
<div>
  <a:documentation>LONG NAME: Concept</a:documentation>
  <define name="concept.content">
    <!-- ... -->
    <optional>
      <ref name="related-links"/>
    </optional>
    <zeroOrMore>
      <ref name="concept-info-types"/>
    </zeroOrMore>
  </define>
</div>
```

In rare cases, it may not be desirable to control topic nesting with the *info-types* pattern. For example:

- If a specialized topic type should never allow any nested topics, regardless of context, it can be defined without any pattern or any nested topics.
- If a specialized topic type should only ever allow specific nesting patterns, such as allowing only other topic types that are defined in the same module, it can nest those topics directly in the same way that other nested elements are defined.

RELAX NG: Coding requirements for element domain modules

Vocabulary modules that define element domains must define an extension pattern for each element that is extended by the domain. These patterns are used when including the domain module in a document-type shell.

All vocabulary and constraint modules must document their `@domains` attribute contribution. The value of the contribution is constructed according to the rules found in [domains attribute rules and syntax](#) on page 122. The OASIS grammar files use a `<domainsContribution>` element to document the contribution; this element is used to help enable generation of DTD and XSD grammar files. An XML comment or `<a:documentation>` element can also be used.

For each element type that is extended by the element domain module, the module must define a domain extension pattern. The pattern consists of a choice group of references to element-type name patterns, with one reference to each extension of the base element type.

The name of the pattern uses the following format, where *shortName* is the short name for the domain, and *elementName* is the name of the element that is extended:

```
shortName-d-elementName
```

For example, the following pattern extends the `<ph>` element type by adding the specializations of `<ph>` that are defined in the highlighting domain:

```
<define name="hi-d-ph">
  <choice>
    <ref name="b.element"/>
    <ref name="i.element"/>
    <ref name="sup.element"/>
    <ref name="sub.element"/>
    <ref name="tt.element"/>
    <ref name="u.element"/>
  </choice>
</define>
```

For each element type that is extended by the element domain module, the module extends the element type pattern with a `@combine` value of "choice" that contains a reference to the domain extension pattern. Because the pattern uses a `@combine` value of "choice", the effect is that the domain-provided elements are automatically added to the effective content model of the extended element in any grammar that includes the domain module.

For example, the following pattern adds the highlighting domain specializations of the `<ph>` element to the content model of the `<ph>` element:

```
<define name="ph" combine="choice">
  <ref name="hi-d-ph"/>
</define>
```

RELAX NG: Coding requirements for attribute domain modules

An attribute domain vocabulary module declares a new attribute specialized from either the `@props` or `@base` attribute. An attribute domain module defines exactly one new attribute type.

All vocabulary and constraint modules must document their `@domains` attribute contribution. The value of the contribution is constructed according to the rules found in [domains attribute rules and syntax](#) on page 122. The OASIS grammar files use a `<domainsContribution>` element to document the contribution; this element is used to help enable generation of DTD and XSD grammar files. An XML comment or `<a:documentation>` element can also be used.

An attribute domain's name is the name of the attribute plus "Att". For example, for the attribute named "deliveryTarget" the attribute domain name is "deliveryTargetAtt". The attribute domain name is used to construct pattern names for the domain.

An attribute domain consists of one file, which has three sections:

Domains attribute contribution

The `@domains` contribution must be documented in the module. The value is constructed according to the rules found in [domains attribute rules and syntax](#) on page 122.

Attribute extension pattern

The attribute extension pattern extends either the `@props` or `@base` attribute set pattern to include the attribute specialization.

For specializations of `@props` the pattern is named `props-attribute-extensions`. The pattern specifies a `@combine` value of "interleave", and the content of the pattern is a reference to the specialized attribute declaration pattern. For example:

```
<define name="props-attribute-extensions" combine="interleave">
  <ref name="deliveryTargetAtt-d-attribute"/>
</define>
```

For specializations of `@base` the pattern is named `base-attribute-extensions`. The pattern specifies a `@combine` value of "interleave", and the content of the pattern is a reference to the specialized attribute declaration pattern. For example:

```
<define name="base-attribute-extensions" combine="interleave">
  <ref name="myBaseSpecializationAtt-d-attribute"/>
</define>
```

Attribute declaration pattern

The specialized attribute is declared in a pattern named *domainShortName-d-attribute*. The attribute must be defined as optional. For example, the `@deliveryTarget` specialization of `@props` is defined as follows:

```
<define name="deliveryTargetAtt-d-attribute">
  <optional>
    <attribute name="deliveryTarget"/>
  </optional>
</define>
```

RELAX NG: Coding requirements for constraint modules

A structural constraint module defines the constraints for a map or topic element type. A domain constraint module defines the constraints for an element or attribute domain.

All vocabulary and constraint modules must document their `@domains` attribute contribution. The value of the contribution is constructed according to the rules found in *domains attribute rules and syntax* on page 122. The OASIS grammar files use a `<domainsContribution>` element to document the contribution; this element is used to help enable generation of DTD and XSD grammar files. An XML comment or `<a:documentation>` element can also be used.

Constraint modules are implemented by importing the constraint module into a document type shell in place of the module that the constraint modifies. The constraint module itself imports the base module to be constrained; within the import, the module redefines patterns as needed to implement the constraint.

For example, a constraint module that modifies the `<section>` element needs to import the base module `topicMod.rng`. Within that import, it will constrain the `section.content` pattern:

```
<include href="topicMod.rng">
  <define name="section.content">
    <!-- Define constrained model here -->
  </define>
</include>
```

For a more complete example, see `strictTaskbodyConstraintMod.rng`, delivered with the DITA 1.3 grammar files.

Because the constraint module imports the module that it modifies, only one constraint module can be used per vocabulary module (otherwise the module being constrained would be imported multiple times). If multiple constraints are combined for a single vocabulary module, they must be implemented in one of the following ways:

- The constraints may be combined into a single module. For example, when combining separate constraints for `<section>` and `<shortdesc>`, a single module may be defined as follows:

```
<include href="topicMod.rng">
  <define name="section.content">
    <!-- Constrained model for section -->
  </define>
  <define name="shortdesc.content">
    <!-- Constrained model for shortdesc -->
  </define>
</include>
```

- Constraints may be chained so that each constraint imports another, until the final constraint imports the base vocabulary module. For example, when combining separate constraints for `<section>`, `<shortdesc>`, and `` from the base vocabulary, the `<section>` constraint can import the `<shortdesc>` constraint, which in turn imports the `` constraint, which finally imports `topicMod.rng`.

Example: contribution to the @domains attribute for structural constraint module

The following code fragment specifies the contribution to the @domains attribute as (topic task strictTaskbody-c):

```
<moduleDesc>
  <!-- ... -->
  <moduleMetadata>
    <!-- ... -->
    <domainsContribution>(topic task strictTaskbody-c)</domainsContribution>
  </moduleMetadata>
</moduleDesc>
```

Example: contribution to the @domains attribute for domain constraint module

The following code fragment illustrates the domains contribution for a constraint module that restricts the task requirements domain:

```
<moduleDesc>
  <!-- ... -->
  <moduleMetadata>
    <!-- ... -->
    <domainsContribution>(topic task taskreq-d requiredReqcondsTaskreq-c)</
domainsContribution>
  </moduleMetadata>
</moduleDesc>
```

XML Schema coding requirements

This section explains how to implement XML Schema (XSD) based document-type shells, specializations, and constraints.

XML Schema: Overview and limitations of coding requirements

DITA coding practices for XML Schema rely on the XSD redefine facility in order to implement specializations or constraints. However, limitations in the redefine facility can present problems for some DITA modules implemented in XML Schema.

Specializations and constraints in XML Schema are implemented using the XSD `<xs:redefine>` facility. However, this facility does not allow sequence groups to be directly constrained. Thus, to support both specialization and constraints, it might be necessary to refactor content models into named groups that can be redefined. In order to keep the XSD, RELAX NG, and DTD implementations as consistent as possible, the DITA Technical Committee only refactored those content models that were required for OASIS-provided grammars, the strict task body and machinery-industry task. The other DITA content models distributed by OASIS have not been refactored.

You *MAY* modify OASIS-provided XSD modules to refactor content models if required by your constraint. You *SHOULD* notify the DITA Technical Committee (TC) of your constraint requirements, so the TC can consider adding the required refactoring to the OASIS-provided XSDs.

XML Schema: Coding requirements for document-type shells

XSD-based document-type shells are organized into sections; each section contains a specific type of declaration.

XSD-based document-type shells use the XML Schema redefine feature (`<xs:redefine>`) to override base group definitions for content models and attribute lists. This facility is analogous to the parameter entities that are used for DTD-based document-type shells. Unlike DTD parameter entities, an `<xs:redefine>` both includes the XSD file that it redefines and holds the redefinition that is applied to the groups in the included XSD file. Thus, for XSD files that define groups, the file can be included using `<xs:include>` if it is used without modification or using `<xs:redefine>` if any of its groups are redefined.

XSD-based document-type shells contain the following sections.

Topic or map domains

For each element or attribute domain that is integrated into the document-type shell, this section uses an `<xs:include>` element to include the XSD module for that domain.

For example:

```
<xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:deliveryTargetAttDomain.xsd:1.3"/>
<xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:highlightDomain.xsd:1.3"/>
<xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:indexingDomain.xsd:1.3"/>
```

Group definitions

The group inclusion section contains `<xs:include>` or `<xs:redefine>` references for element groups. The group files define named groups that are used to integrate domain-provided element and attribute types into base content models. There is one group file for each structural type; domain files can also have group files.

For both map and topic shells, this section also must include or redefine the following groups; it must also include the module file for each group:

- Common element group (`commonElementGrp.xsd` and `commonElementMod.xsd`)
- Metadata declaration group (`metaDeclGrp.xsd` and `metaDeclMod.xsd`)
- Table model group (`tblDeclGrp.xsd` and `tblDeclMod.xsd`)

The group files and the module files for base groups can be specified in any order.

For each element extended by one or more domains, the document-type shell must redefine the model group for the element to a list of alternatives including the literal name of the element and the element extension model group from each domain that is providing specializations. To integrate a new domain in the document-type shell, use the schema `<xs:redefine>` mechanism to import a group definition file while redefining and extending an element from that group. The model group requires a reference to itself to extend the base model group.

For each attribute extended by one or more domains, the document-type shell must redefine the attribute extension model group for the attribute to a list of alternatives including the literal name of the attribute and the attribute extension model group from each domain that is providing specializations. To integrate a new attribute domain in the document-type shell, use the schema `<xs:redefine>` mechanism to import the `commonElementGrp.xsd` group file while redefining and extending the base attribute.

For example, the following portion of a document-type shell includes the common metadata module and then adds a domain extension of the `<metadata>` element from the metadata group. It also extends the `@props` attribute from the common element module to add the specialized attribute `@deliveryTarget`.

```
<xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:metaDeclMod.xsd:1.3"/>
<!-- ... -->
<xs:redefine schemaLocation="urn:oasis:names:tc:dita:xsd:commonElementGrp.xsd:1.3">
  <!-- ...Redefinition of any elements in common module -->
  <xs:attributeGroup name="props-attribute-extensions">
    <xs:attributeGroup ref="props-attribute-extensions"/>
    <xs:attributeGroup ref="deliveryTargetAtt-d-attribute"/>
  </xs:attributeGroup>
</xs:redefine>
<xs:redefine schemaLocation="urn:oasis:names:tc:dita:xsd:metaDeclGrp.xsd:1.3">
  <xs:group name="metadata">
    <xs:choice>
      <xs:group ref="metadata"/>
      <xs:group ref="relmgmt-d-metadata"/>
    </xs:choice>
  </xs:group>
</xs:redefine>>
```

Module inclusions

The module inclusion section includes the module XSD files for structural types used in the shell. These must be placed after the group and files and redefinitions.

This section must also include any other module XSD files required by the topic or map types. For example, if a troubleshooting specialization is specialized from topic but includes elements from task, then the task structural module must be included in the document shell.

If a structural type is constrained, that constraint will be included rather than the module itself; for example, in a document-type shell that constrains the task specialization, the task constraint module will be included rather than the task module.

For example, the following portion of a document-type shell includes the structural modules for topic and concept:

```
<xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:topicMod.xsd:1.3"/>
<xs:include schemaLocation="urn:oasis:names:tc:dita:xsd:conceptMod.xsd:1.3"/>
```

Domains attribute declaration

The `@domains` attribute declaration section declares the `@domains` attribute for the shell. It does this by redefining the `domains-att` group, adding one token for each vocabulary and constraint module integrated by the shell. See [domains attribute rules and syntax](#) on page 122 for details on the syntax for domains tokens.

For example, the following sample defines the `domains-att` to include the OASIS domains for map group, indexing, and `@deliveryTarget`:

```
<xs:attributeGroup name="domains-att">
  <xs:attribute name="domains" type="xs:string"
    default="(map mapgroup-d) (topic indexing-d) a(props deliveryTarget)"/>
</xs:attributeGroup>
```

Info-types definition

This section defines whether and how topics can nest by redefining the `info-types` group. That group is referenced but undefined in the module files, so it must be defined in the shell. Topic testing can be

disallowed by setting the `info-types` group to reference the `<no-topic-nesting>` element, with the `@maxOccurs` and `@minOccurs` attributes each set to "0".

Optionally, `topic-type-info-types` groups can be redefined to provide more fine grained control of nesting with specialized topic types. As with domain extensions, this is done by redefining the group while importing the module that defines the group.

For example, in the concept vocabulary module delivered by OASIS, the `concept-info-types` group controls which topics can nest inside the `<concept>` element. That group is defined as including `<concept>` plus the `info-types` group. The following examples demonstrate how to control topic nesting within `<concept>` using a document-type shell.

- To have `<concept>` only nest itself, the `info-types` group must be defined so that it does not add any additional topics:

```
<xs:group name="info-types">
  <xs:choice>
    <xs:element ref="no-topic-nesting" maxOccurs="0" minOccurs="0"/>
  </xs:choice>
</xs:group>
```

- In order to turn off topic nesting entirely within `<concept>`, the `concept-info-types` group must be redefined to remove `<concept>`, and the `info-types` group must be defined as above:

```
<xs:group name="info-types">
  <xs:choice>
    <xs:element ref="no-topic-nesting" maxOccurs="0" minOccurs="0"/>
  </xs:choice>
</xs:group>

<xs:redefine schemaLocation="urn:oasis:names:tc:dita:xsd:conceptMod.xsd:1.3" >
  <xs:group name="concept-info-types">
    <xs:choice>
      <xs:group ref="info-types"/>
    </xs:choice>
  </xs:group>
</xs:redefine>
```

- In order to add `<topic>` as a nesting topic within `<concept>`, define `info-types` to allow any number of `<topic>` elements:

```
<xs:group name="info-types">
  <xs:choice>
    <xs:element ref="topic" maxOccurs="unbounded" minOccurs="0"/>
  </xs:choice>
</xs:group>
```

- With the preceding example, `<concept>` is allowed to nest either `<concept>` or `<topic>`. In order to make `<topic>` the *only* valid child topic, the `concept-info-types` must be redefined as follows:

```
<xs:redefine schemaLocation="urn:oasis:names:tc:dita:xsd:conceptMod.xsd:1.3" >
  <xs:group name="concept-info-types">
    <xs:choice>
      <xs:group ref="info-types"/>
    </xs:choice>
  </xs:group>
</xs:redefine>
```

XML Schema: Coding requirements for element type declarations

Structural and domain vocabulary modules have the same XSD coding requirements for element type declarations.

Element definitions

A structural or domain vocabulary module must contain a declaration for each specialized element type named by the module. While the XSD standard allows content models to refer to undeclared element types, all element types named in content models within a vocabulary module must have an `<xs:element>` declaration, either in the vocabulary module, in a base module from which the vocabulary module is specialized, or in a required domain module.

Domain modules consist of a single XSD document. Structural modules consist of two modules; one module contains all element name groups, and the other contains all other declarations for the module.

For each element type that is declared in the vocabulary module, the following set of groups and declarations must be used to define the content model and attributes for the element type. These groups are typically placed together within the module for clarity.

- For each element type declared in the vocabulary module there must be an `<xs:group>` element whose name is the element type name, and whose one member is a reference to the element type. This element name group provides a layer of abstraction that facilitates redefinition. A document-type shell can redefine an element group to add domain-specialized elements or to replace a base element type with one or more specializations of that type.
- Each element type must have a corresponding content model group named `tagname.content`. The value of the group is the complete content model definition; the content model group can be overridden in constraint modules to further constrain the content model.
- Each element type must have a corresponding attribute group named `tagname.attributes`. The value of the group is the complete attribute set for the element type, except for the `@class` attribute. Like the content model, this group can be overridden in a constraint module to constrain the attribute set.
- Each element type must have a complex type definition named `tagname.class`, which references the `tagname.content` and `tagname.attributes` groups.
- Each element type must have an `<xs:element>` declaration named `tagname`, that uses as its type the `tagname.class` complex type and extends that complex type to add the `@class` attribute for the element.

For example, the following set of declarations shows each of the required groups and definitions for the specialized `<codeph>` element.

```
<xs:group name="codeph">
  <xs:sequence>
    <xs:choice>
      <xs:element ref="codeph"/>
    </xs:choice>
  </xs:sequence>
</xs:group>

<xs:group name="codeph.content">
  <xs:sequence>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:group ref="basic.ph.notm"/>
      <xs:group ref="data.elements.incl"/>
      <xs:group ref="draft-comment"/>
      <xs:group ref="foreign.unknown.incl"/>
      <xs:group ref="required-cleanup"/>
    </xs:choice>
  </xs:sequence>
</xs:group>
```

```

<xs:attributeGroup name="codeph.attributes">
  <xs:attributeGroup ref="univ-attns"/>
  <xs:attribute name="outputclass" type="xs:string"/>
  <xs:attributeGroup ref="global-attns"/>
</xs:attributeGroup>

<xs:complexType name="codeph.class" mixed="true">
  <xs:sequence>
    <xs:group ref="codeph.content"/>
  </xs:sequence>
  <xs:attributeGroup ref="codeph.attributes"/>
</xs:complexType>

<xs:element name="codeph">
  <xs:annotation>
    <xs:documentation> <!-- documentation for codeph --> </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="codeph.class">
        <xs:attribute ref="class" default="+ topic/ph pr-d/codeph "/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

```

XML Schema: Coding requirements for structural modules

An XSD structural module declares a top-level map or topic type. It is implemented as a pair of XSD documents, one that defines groups used to integrate and override the type and one that defines the element types specific to the type.

All vocabulary and constraint modules must document their @domains attribute contribution. The OASIS grammar files use a < dita:domainsModule > element to document the contribution; this element is used consistently to make it easy to find values when creating a document type shell. An XML comment or < xs:appinfo > element can also be used.

Module files

A structural vocabulary module has two files:

- A module schema document.
- A module group definition schema document.

Required topic and map element attributes

The root element for a structural type must reference the @DITAArchVersion attribute and the @domains attribute. These attributes give processors a reliable way to check the architecture version and look up the list of domains available in the document type. The attributes are referenced as in the following example:

```

<xs:attributeGroup name="concept.attributes">
  <!-- Various other attributes -->
  <xs:attribute ref="ditaarch:DITAArchVersion"/>
  <xs:attributeGroup ref="domains-att"/>
  <xs:attributeGroup ref="global-attns"/>
</xs:attributeGroup>

```

Controlling nesting in topic types

For topic modules, the last position in the content model is typically a reference to the *topic-type-info-types* group. Document types shells can control how topics are allowed to nest by redefining the group. If topic nesting is hard coded in the structural module, and cannot be modified from the document-type shell, the *topic-type-info-types* group is not needed.

For example, the vocabulary module for the `<concept>` structural type uses the group `concept-info-types`; this group is given a default value, and then referenced from the content model for the `<concept>` element type:

```
<xs:group name="concept-info-types">
  <xs:choice>
    <xs:group ref="concept"/>
    <xs:group ref="info-types"/>
  </xs:choice>
</xs:group>

<xs:group name="concept.content">
  <xs:sequence>
    <xs:group ref="title"/>
    <!-- ...other elements, such as shortdesc and body, and then... -->
    <xs:group ref="related-links" minOccurs="0"/>
    <xs:group ref="concept-info-types" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:group>
```

XML Schema: Coding requirements for attribute domain modules

An attribute domain vocabulary module declares one new attribute specialized from either the `@props` or `@base` attribute.

All vocabulary and constraint modules must document their `@domains` attribute contribution. The OASIS grammar files use a `<dita:domainsModule>` element to document the contribution; this element is used consistently to make it easy to find values when creating a document type shell. An XML comment or `<xs:appinfo>` element can also be used.

An attribute domain consists of one file. The file must have a single attribute group definition that contains the definition of the attribute itself, where the attribute group is named `attnameAtt-d-attribute`.

For example, the `@deliveryTarget` attribute is defined with the following attribute group:

```
<xs:attributeGroup name="deliveryTargetAtt-d-attribute">
  <xs:attribute name="deliveryTarget" type="xs:string">
    <!-- Documentation for attribute -->
  </xs:attribute>
</xs:attributeGroup>
```

XML Schema: Coding requirements for constraint modules

A structural constraint module defines the constraints for a map or topic element type. A domain constraint module defines the constraints for an element or attribute domain.

All vocabulary and constraint modules must document their `@domains` attribute contribution. The OASIS grammar files use a `<dita:domainsModule>` element to document the contribution; this element is used consistently to make it easy to find values when creating a document type shell. An XML comment or `<xs:appinfo>` element can also be used.

For each vocabulary module with a content model or attributes to be constrained, there must be an `<xs:redefine>` element that references the vocabulary module. Within the `<xs:redefine>` element, for each

element type content model to be constrained, an `<xs:group>` element is needed with the name `element.content`. Also within the `<xs:redefine>` element, for each attribute set to be constrained, an `<xs:attributeGroup>` element is needed with the name `element.attributes`. The constrained model is defined inside of the `<xs:group>` or `<xs:attributeGroup>`.



Note: This means that when adding a constraint module to an existing document-type shell, you must remove any `<xs:include>` elements that refer to the XSD module to which the redefine is applied. For example, to redefine groups defined in `commonElementsMod.xsd`, you must remove any `<xs:include>` reference to `commonElementsMod.xsd`.

Because the constraint module includes the module that it modifies, only one constraint module can be used per vocabulary module (otherwise the module being constrained would be included multiple times). If multiple constraint modules are needed for a single vocabulary module, they must be combined into a single XSD module. For example, when combining existing constraint modules for `<p>` and `<div>`, a single module must be created that combines the `<xs:group>` and `<xs:attributeGroup>` constraints from existing modules inside a single `<xs:redefine>` reference to `commonElementsMod.xsd`.

When constraining a list of elements provided by a domain, there must be a group that lists the subset of domain elements in a constraints module. The group name *SHOULD* be named "`qualifierdomain-c-tagname`" where *qualifier* is a description for the constraint module, *domain* is the name of the domain, map, or topic being constrained, and *tagname* is the name of the extension element being restricted.

Example: Structural constraint module

The following code fragment shows how the `<topic>` element can be constrained to disallow the `<body>` element. This `<xs:redefine>` element is located in a constraint module that references the file `topicMod.xsd`, which means that a document-type shell using this constraint would reference this module *instead of* referencing `topicMod.xsd` (it would not reference both).

```
<xs:redefine schemaLocation="urn:oasis:names:tc:dita:xsd:topicMod.xsd:1.2">
  <xs:group name="topic.content">
    <xs:sequence>
      <xs:sequence>
        <xs:group ref="title"/>
        <xs:group ref="titlealts" minOccurs="0"/>
        <xs:choice minOccurs="1" >
          <xs:group ref="shortdesc" />
          <xs:group ref="abstract" />
        </xs:choice>
        <xs:group ref="prolog" minOccurs="0"/>
        <!--<xs:group ref="body" minOccurs="0"/>-->
        <xs:group ref="related-links" minOccurs="0"/>
        <xs:group ref="topic-info-types" minOccurs="0"
          maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:sequence>
  </xs:group>
</xs:redefine>
```

For a more complete example, see `strictTaskbodyConstraintMod.xsd`, delivered with the DITA 1.3 grammar files.

Example: Domain constraint module

The following code fragment shows how the highlighting domain can be constrained to limit the elements that are available in the domain to only `` and `<i>`.

```
<xs:group name="basicHighlight-c-ph">
  <xs:choice>
    <xs:element ref="b"/>
```

```
<xs:element ref="i"/>  
</xs:choice>  
</xs:group>
```

Index

A

- addressing mechanisms
 - effect on conref resolution [83](#)
 - same-topic fragment identifier
 - authoring responsibility [83](#)
 - effect on conref resolution [83](#)
- attributes
 - conditional processing [38](#)

B

- base sort phrase [114](#)
- bi-directional text [112](#)
- binding controlled values [28](#)
- branch filtering [91, 92](#)

C

- @cascade attribute
 - example [20](#)
- cascading
 - map-to-map
 - attributes [46](#)
 - exceptions [48](#)
 - metadata elements [47](#)
- cascading, definition [41](#)
- classifying content [27](#)
- coding requirements
 - DTD
 - constraints [152](#)
 - RELAX NG
 - constraints [164](#)
 - XSD
 - constraints [171](#)
- collation [114](#)
- conditional processing
 - attributes [38](#)
- conref
 - combining attributes [82](#)
 - constraint modules and compatibility [133](#)
 - determining validity [113](#)
 - overview [81](#)
 - processing expectations [82](#)
 - pull [81](#)
 - push [81](#)
 - range [81](#)
 - validity of [82](#)
 - xrefs and conref within a conref [83](#)
- constraints
 - compatibility checking [133](#)
 - compatibility enforcement [133](#)
 - conref compatibility [133](#)
 - design and implementation rules [132](#)
 - DTD

XSD (*continued*)

- DTD (*continued*)
 - coding requirements [152](#)
 - integrating into document type shells [152](#)
- examples
 - applying multiple constraints [141](#)
 - redefining the content model [136](#)
 - replacing base element with domain extensions [140](#)
 - restricting attributes for an element [137](#)
 - restricting content model for a domain [139](#)
- overview [131](#)
- processing and interoperability [133](#)
- RELAX NG
 - coding requirements [164](#)
 - integrating into document type shells [164](#)
- weak and strong [133](#)
- XSD
 - coding requirements [171](#)
 - examples [171](#)
 - integrating into document-type shells [171](#)
- content references, *See* conref
- controlled values
 - binding to attributes [28](#)
 - classifying content for flagging and filtering [27](#)
 - defining a taxonomy [31](#)
 - definition of [27](#)
 - overview [27](#)
 - precedence rules [28](#)
 - validation of [28, 29](#)
- conventions
 - file extensions [10](#)
- copied elements
 - determining validity [113](#)
- core concepts
 - addressing [9](#)
 - conditional processing [9](#)
 - configuration [9](#)
 - constraints [9](#)
 - content reuse [9](#)
 - information typing [9](#)
 - maps [9](#)
 - specialization [9](#)
 - topics [9](#)
- cross references
 - resolving within conrefs [83](#)

D

- definitions
 - base sort phrase [114](#)
 - cascading [41](#)
 - controlled values [27](#)
- @deliveryTarget
 - defining values for [28](#)
- @dir attribute [112](#)

DITA maps, *See* maps
 -dita-use-conref-target [82](#)
 DITaval

processing expectations [29](#)

domain constraint modules

DTD

coding requirements [152](#)

RELAX NG

coding requirements [164](#)

XSD

coding requirements [171](#)

examples [171](#)

@domains attribute

constraint compatibility [133](#)

E

effective sort phrase [114](#)

examples

constraint modules and conref compatibility [133](#)

constraints

applying multiple constraints [141](#)

redefining the content model [136](#)

replacing base element with domain extensions [140](#)

restricting attributes for an element [137](#)

restricting content model for a domain [139](#)

@domains attribute

constraint contribution to [141](#)

effective sort phrase [114](#)

maps

@collection-type and @linking in relationship

tables [20](#)

relationship tables [20](#)

use of @cascade attribute [20](#)

processing

filtering or flagging a hierarchy [29, 32](#)

xrefs and conref within a conref [83](#)

subjectScheme

binding controlled values [28](#)

defining a taxonomy [31](#)

defining values for @deliveryTarget [36](#)

extending a subject scheme [34, 35](#)

filtering or flagging a hierarchy [29, 32](#)

providing a subject-definition resource [27](#)

@xml:lang [110](#)

F

file extensions

conditional processing profiles [10](#)

DITaval [10](#)

maps [10](#)

topics [10](#)

file names

DTD

domain constraint modules [152](#)

structural constraint modules [152](#)

RELAX NG

domain constraint modules [164](#)

XSD (*continued*)

RELAX NG (*continued*)

structural constraint modules [164](#)

XSD

domain constraint modules [171](#)

structural constraint modules [171](#)

filtering

attributes [38](#)

filtering and flagging

classifying content for [27](#)

processing expectations [29](#)

flagging

attributes [38](#)

formatting

processing expectations [11](#)

formatting expectations

@xml:lang [110](#)

G

generalization

conref resolution [82](#)

grouping [114](#)

I

<index-sort-as> [114](#)

information typing

benefits [14](#)

history [14](#)

overview [14](#)

interoperability

constraints [133](#)

K

key reference

conref resolution, effect on [83](#)

key scopes

conref resolution, effect on [83](#)

M

map-to-map cascading

attributes [46](#)

exceptons [48](#)

metadata elements [47](#)

maps

attributes

shared with topics [20](#)

unique to maps [20](#)

elements [18](#)

examples

relationship tables [20](#)

overview [17](#)

purposes [17](#)

subject scheme, *See* subjectScheme

use of @xml:lang [110](#)

metadata
 cascading [37](#)
 conditional processing attributes [38](#)
 elements [37](#)

N

nested topics [12, 15](#)
 notation
 attribute types [5](#)
 element types [5](#)

O

outputclass attribute
 example [38](#)
 overview [38](#)

P

precedence rules
 combining attributes on conrefs [82](#)
 controlled values [28](#)
 processing
 conrefs [82](#)
 controlled values [29](#)
 examples
 filtering or flagging a hierarchy [29, 32](#)
 xrefs and conref within a conref [83](#)
 sorting [114](#)
 xrefs and conref within a conref [83](#)
 processing expectations
 attribute values, hierarchies of [29](#)
 base sort phrase, documentation of [114](#)
 bi-directional text [112](#)
 checking of constraint compatibility [133](#)
 combining attributes on conrefs [82](#)
 conrefs, validity of [82](#)
 controlled values [27](#)
 DITAVAL [29](#)
 enforcing constraint compatibility [133](#)
 filtering and flagging [29](#)
 formatting [11](#)
 generalization during conref resolution [82](#)
 parameters for referencing subjectScheme [27](#)
 subject-definition resources [27](#)
 validating controlled values [28](#)
 validity of copied elements [113](#)
 @xml:lang [110](#)
 xrefs and conref within a conref [83](#)

R

relationship tables
 examples [20](#)

S

same-topic fragment identifier
 authoring responsibility [83](#)
 effect on conref resolution [83](#)
 single sourcing [11](#)
 <sort-as> [114](#)
 sorting [114](#)
 specialization
 best practices [14](#)
 overview [118](#)
 strong constraints [133](#)
 structural constraint modules
 DTD
 coding requirements [152](#)
 RELAX NG
 coding requirements [164](#)
 XSD
 coding requirements [171](#)
 examples [171](#)
 subject-definition resources [27](#)
 subjectScheme
 binding controlled values [28](#)
 defining a taxonomy [31](#)
 defining controlled values [27](#)
 examples
 binding controlled values [28](#)
 defining a taxonomy [31](#)
 defining values for @deliveryTarget [36](#)
 extending a subject scheme [34, 35](#)
 filtering or flagging a hierarchy [29, 32](#)
 providing a subject-definition resource [27](#)
 extending [27](#)
 overview [27](#)

T

taxonomy
 defining [31](#)
 terminology
 addressing and linking [5](#)
 basic concepts [5](#)
 modules [5](#)
 non-normative information [5](#)
 normative information [5](#)
 specialization [5](#)

topics
 benefits [12](#)
 content [16](#)
 generic topic type [14](#)
 information typing [14](#)
 overview [12](#)
 reuse [12](#)
 structure [15](#)
 use of @xml:lang [110](#)

translation
 @xml:lang [110](#)

U

use by reference, *See* [conref](#)

V

validating controlled values [28](#), [29](#)

W

weak constraints [133](#)

X

`@xml:lang` attribute

best practices [110](#)

default values [110](#)

example [110](#)

overview [110](#)

use with `@conref` or `@conkeyref` [110](#)

xrefs, *See* cross references
