An OASIS DITA Adoption Technical Committee Publication

DITA 1.3 Feature Article: Understanding Scoped Keys in DITA 1.3

Author: Leigh W. White, IXIASOFT Technologies
On behalf of the DITA Adoption Technical Committee

Date: 09 September 2015

OASIS Legal New

OASIS (Organization for the Advancement of Structured Information Standards) is a not-for-profit, international consortium that drives the development, convergence, and adoption of e-business standards. Members themselves set the OASIS technical agenda, using a lightweight, open process expressly designed to promote industry consensus and unite disparate efforts. The consortium produces open standards for Web services, security, e-business, and standardization efforts in the public sector and for application-specific markets. OASIS was founded in 1993. More information can be found on the OASIS website at http://www.oasis-open.org.

The OASIS DITA Adoption Technical Committee members collaborate to provide expertise and resources to educate the marketplace on the value of the DITA OASIS standard. By raising awareness of the benefits offered by DITA, the DITA Adoption Technical Committee expects the demand for, and availability of, DITA conforming products and services to increase, resulting in a greater choice of tools and platforms and an expanded DITA community of users, suppliers, and consultants.

DISCLAIMER: All examples presented in this article were produced using one or more tools chosen at the author's discretion and in no way reflect endorsement of the tools by the OASIS DITA Adoption Technical Committee.

This white paper was produced and approved by the OASIS DITA Adoption Technical Committee as a Committee Draft. It has not been reviewed and/or approved by the OASIS membership at-large.

Document History

Revision	Date	Author	Summary
First Draft	2015-07-13	White	Initial draft
Second Draft		White	
Last Draft		White	
Final	2015-08-17	White	Approved
Committee Approved Draft	2015-09-09	White	Adoption TC approved final draft

Table of Contents

Summary

DITA 1.3 introduces scoped keys, which represent an extension of keys as introduced in DITA 1.2. In a nutshell, DITA 1.2 allowed for only one definition of a given key per root map. If you wanted your *product-name* key to be resolved as "Widget" in some topics and as "Gadget" in other topics, you're out of luck.

On the other hand, scoped keys in DITA 1.3 allow you to do just this kind of thing. Scoped keys apply to a defined key space within a map. This key space, defined by the *keyscope* attribute, can be the entire root map, a sub-map, a topichead or topicgroup, or even a single topicref. A map can contain multiple key spaces. In the case of parallel key spaces, if the same key is defined in more than one key space, each definition holds only for the key space that includes it. In the case of nested key spaces, the definition in the parent key space holds for all child key spaces. A key definition at the root map necessarily overrides any definitions of the same key at a lower level within the root map.

This feature article does not include a full discussion of key functionality. For a complete description of keys as introduced in DITA 1.2, refer to the *Understanding DITA Keys and Key Spaces* feature article on that subject at *oasis-open.org*.

The following sections first briefly discuss the limitations of keys in DITA 1.2 and then explain several different possible configurations of scoped keys and give examples of each.

Key limitations in DITA 1.2

To fully grasp the significance of scoped keys, it's helpful to understand how keys worked—and did not work—in DITA 1.2. Keys were introduced as a means of indirect addressing, meaning that the reference is to the name of the link target rather than to the location of the link target.

To clarify the difference, consider a conref, which is an example of direct addressing. The syntax for a conref is similar to the following:

```
<p conref="myconrefstore.dita#conrefstore/para1"/>
```

where the conref references a topic named myconrefstore.dita, whose id is "conrefstore." Within that topic, the conref specifically references a paragraph whose id is "para1." Regardless of the context in which the topic is used, the example paragraph is always resolved to the same content because the path is explicit and fixed.

On the other hand, indirect addressing using keys takes a form similar to the following example:

```
<p conkeyref="conrefstore/para1"/>
```

where the conkeyref references a target named "conrefstore" and within that target, references a specific element whose id is "para1."

The value of *conrefstore* is then defined within the current context. The value can come from a key definition added directly to the root map or from a key definition added to a separate map, which is then referenced by the root map.

This means that the topic can include different content for different outputs, depending on the context-specific definition of the *conrefstore* key.

The introduction of keys in DITA 1.2 was therefore a huge step forward in facilitating content reuse. Unfortunately, it contained a major limitation. Within a root map, a key could have only one value, which made it impossible, for example, to define a key with one value in one sub-map and with another value in a second sub-map, both referenced by a single root map.

Here is a common example of this scenario. You write documentation for a product that has several modules. The documentation for each of these modules is contained in separate sub-maps which you then reference from a master root map to create one large publication. There is a topic which is similar enough for all modules that you have written it once in such a way that it is suitable for use in all the module maps, thus maximizing your reuse of that information. The only thing that must differ between occurrences of the topic is the module name used.

Before DITA 1.2, you tackled this problem with conditions:

```
<ph module="widget">Widget</ph><ph module="gadget">Gadget</ph><ph module="doodad">Doodad</ph>
```

Eager to reduce the overhead of condition management, as soon as you learned of the new key functionality in DITA 1.2, you converted all these instances into a single key: <ph keyref="module-name"/>.

Note: The examples in this article use simple keyword keys. While these are a less sophisticated use of keys, they are a visually simple example to follow. Generally speaking, keys have a greater value when used for image or link replacement or, as part of conkeyref, pointers to alternating topics in different contexts. The scoped key principles hold true for any kind of key definition.

Then, in each module sub-map, you added a definition for the *module-name* key that corresponded to the name of the module. In your root map, the Widget sub-map was referenced first, followed by the Gadget sub-map, followed by the Doodad sub-map. With great anticipation, you created your first output with the new keys, expecting the first occurrence of the topic in the Widget sub-map to feature the module name "Widget," the second (in the Gadget sub-map) to feature the module name "Gadget," and the third (in the Doodad sub-map) to feature the module name "Doodad." Your disappointment was crushing when you discovered that all three instances of the topic in all three sub-maps featured the product name "Widget."

You had run up against the major limitation of keys in DITA 1.2: that in all cases, the first definition of the key was the one that prevailed throughout the root map.

What exactly does "first definition" mean? It means slightly different things in different contexts—for example, depending on whether you are processing a single map, or a root map and sub-maps, and whether the keys are defined directly in the root map or in referenced maps. The full explanation is too lengthy to include here, so if you want the details, refer to this article published by the OASIS DITA Technical Committee: *http://dita.xml.org/resource/dita-tc-faq-about-keys*, which clarifies how a processor should determine which key definition is in use.

Until this one-map, one-definition limitation could be resolved, the full power of keys would lie outside the reach of DITA content creators and Information Architects.

Scoped keys

The solution to the key limitations found in DITA 1.2 was to allow a key to be defined multiple times within a map, with each definition being effective within a specific scope. DITA 1.3 introduces that functionality with scoped keys.

A key scope defines its own key space. The mechanism for defining a key scope is simple. DITA 1.3 includes a new attribute, *keyscope*, which defines a given "section" of a map as a key scope. This section can be the root map, a sub-map, a *mapref*, a *topichead*, a *topicgroup*, or even an individual *topicref*, as shown in the following simple map

```
<map keyscope="Keys1">
  <keydef keys="module-name">
    <topicmeta>
      <keywords>
        <keyword>Widget</keyword>
      </keywords>
    </topicmeta>
  </keydef>
  <topichead navtitle="Section One" keyscope="Keys2">
    <keydef keys="location">
      <topicmeta>
        <keywords>
          <keyword>Milwaukee</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <topicref href="TopicA.dita"/>
  </topichead>
  <topicgroup keyscope="Keys3">
    <keydef keys="section">
      <topicmeta>
        <keywords>
          <keyword>3C</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <topicref href="TopicB.dita"/>
  </topicgroup>
  <topicref href="TopicC.dita" keyscope="Keys4">
    <keydef keys="bin-no">
      <topicmeta>
        <keywords>
          <keyword>8463</keyword>
        </keywords>
      </topicmeta>
    </keydef>
  </topicref>
  <mapref href="MapD.ditamap" keyscope="Keys5">
    <keydef keys="part-no">
      <topicmeta>
        <keywords>
          <keyword>AE483H5</keyword>
        </keywords>
      </topicmeta>
    </keydef>
  </mapref>
</map>
```

In this map, the *module-name* key is defined in the "Keys1" key space, within the root map. By definition, the root map always has an implicit key space; it's not necessary to explicitly declare the key space for the root map. There are advantages and disadvantages to doing so, depending on your use case.

The *location* key is defined in the "Keys2" key space, within the *topichead*. The *section* key is defined within the "Keys3" keyspace, within the *topicgroup*. The *bin-no* key is defined within the "Keys4" key space, within the *topicref*. (Note that keys cannot be defined within topics themselves, though they can be defined within a *topicref* that references a topic.) Finally, the *part-no* key is defined in the "Keys5" key space, within the *mapref*.

As you see, a key space can be very broad or very specific. A key defined within a key space can only be referenced by its (unqualified) key name from within that same key space. In effect, a key space is a "fence" around a certain set of content.

No other content set can use the key definitions found within the scoped content set unless explicitly directed to do so via another new mechanism, which we will look at shortly. You might guess that this mechanism has something to do with the "(unqualified)" phrase just now, and you would be correct.

The fact that key definitions are inherently effective only within their defined scope allows content creators a great deal more flexibility when reusing topics at different locations within a map or when combining multiple maps into a single publication. At this point, though, scoped keys probably still seem a bit abstract, so let's look at some specific examples of scoped keys in action.

Parallel key spaces

Parallel key spaces, or those that do not overlap in any way, represent the most straightforward use of scoped keys.

As explained, a key space can be defined at multiple levels within a map. The highest level at which a key space can be defined is the root map level. One of the most common uses of scoped keys is likely to be the case of multiple sub-maps within a root map (this example is mentioned earlier in this article). For example, a content creator might need to produce an omnibus publication that combines several smaller publications. Here is a root map that references two sub-maps:

```
<map>
  <title>Training Courses</title>
  <mapref href="widget.ditamap"/>
  <mapref href="gadget.ditamap"/>
</map>
```

The **widget.ditamap** map defines a key, *module-name*, with the value "Widget." It also references the topic **get-started.dita**.

```
<map>
  <title>Using the Widget</title>
  <keydef keys="module-name">
    <topicmeta>
      <keywords>
        <keyword>Widget</keyword>
      </keywords>
    </topicmeta>
  </keydef>
  <topicref href="get-started.dita"/>
</map>
```

The **get-started.dita** topic contains a paragraph with a reference to the *module-name* key:

```
<p>Now that you have integrated the <ph keyref="module-name">, many new options are available to you.</p>
```

Likewise, the **gadget.ditamap** map defines the same key, *module-name*, with the value "Gadget." It also references the topic **get-started.dita**:

```
<map>
  <title>Using the Gadget</title>
  <keydef keys="module-name">
    <topicmeta>
      <keywords>
        <keyword>Gadget</keyword>
      </keywords>
    </topicmeta>
  </keydef>
  <topicref href="get-started.dita"/>
</map>
```

In DITA 1.2, this scenario would have resulted in the key being resolved to "Widget" in both occurrences of **get-started.dita**.

Note: Clearly, the processor must create two copies of **get-started.dita** to resolve the key to two different values, and this requirement is stated in the DITA 1.3 specification. The processor will likely name the copies based on its own algorithm (the specification does not outline requirements for this aspect of processing). If you need specific URIs for the copies, you can use the *copy-to* attribute to specify an appropriate URI for each reference to a topic.

By adding distinct key spaces to each sub-map using the new *keyscope* attribute, it's possible to specify that within the "widget" keyscope, which encompasses the **widget.ditamap**, the *module-name* key should be defined as "widget." Within the "gadget" keyscope, which encompasses the **gadget.ditamap**, the *module-name* key should be resolved as "gadget."

```
<map>
  <title>Training Courses</title>
```

```
   <mapref href="widget.ditamap" keyscope="widget"/>
   <mapref href="gadget.ditamap" keyscope="gadget"/>
</map>
```

The *keyscope* attributes ensure that all key definitions within each sub-map are applicable only within that map and that no key definition in **gadget.ditamap** is overidden by a different definition for the same key in **widget.ditamap**, or vice-versa.

As mentioned earlier, a key space need not be an entire map. The *keyscope* attribute can be used with the *topichead* and *topicgroup* elements as well, facilitating the definition of a key space at various levels within a map. In the following example, the topic **get-started.dita** is included in a root map twice, once as a child of a *topicgroup* and again as a child of a *topichead*:

```
<map>
   <title>Training Courses</title>
   <topicgroup keyscope="widget">
      <keydef keys="module-name">
         <topicmeta>
            <keywords>
               <keyword>Widget</keyword>
            </keywords>
         </topicmeta>
      </keydef>
     <topicref href="get-started.dita"/>
   </topicgroup>
   <topichead keyscope="gadget" navtitle="Using the Gadget">
      <keydef keys="module-name">
         <topicmeta>
            <keywords>
               <keyword>Gadget</keyword>
            </keywords>
         </topicmeta>
      </keydef>
      <topicref href="get-started.dita"/>
   </topichead>
</map>
```

Again, assume an implicit use of the *copy-to* attribute to create two copies of the topic. The result of this scenario is that the first occurrence of the **get-started.dita** topic (and any other topic within the *topicgroup*) is output with the *module-name* key resolved to "Widget." Likewise, the second occurrence of the **get-started.dita** topic (and any other topic within the *topichead*) is output with the *module-name* key resolved to "Gadget."

Finally, a key space can be as granular as a single topic. The following scenario illustrates this use case.

In many cases, there is no need to create sub-maps for a simple publication. Likewise, it's simply extra work to create *topichead* or *topicgroup* elements within the map just to define separate key spaces. In such cases, the key space can be defined on the *topicref* itself. In this example, the same map is used to produce a deliverable for both the Widget and the Gadget. The Information Architect does not want authors to edit the map itself (to change key definitions, for example) and so everything must be set up once and the output managed via a combination of keys and conditional attributes. Therefore, the topic **get-started.dita** appears twice in the map, conditionalized using the *product* attribute:

```
<map>
   <title>Training Courses</title>
   <topicref href="get-started.dita" keyscope="widget" product="widget">
      <keydef keys="module-name">
         <topicmeta>
            <keywords>
               <keyword>Gadget</keyword>
            </keywords>
         </topicmeta>
      </keydef>
   </topicref>
   <topicref href="get-started.dita" keyscope="gadget" product="gadget">
      <keydef keys="module-name">
         <topicmeta>
            <keywords>
               <keyword>Widget</keyword>
            </keywords>
         </topicmeta>
```

```
    </keydef>
  </topicref>
</map>
```

This topic includes the following paragraph:

```
If you also choose to integrate the <ph keyref="module-name"/>, many more options are available to you.
```

The topic as it applies to the Widget should resolve the *module-name* key as "Gadget" so that the paragraph reads "If you also choose to integrate the Gadget, many more options are available to you." If someone is reading the output as produced for the Widget, they obviously already have the Widget but they might also want to know that there is a Gadget available.

Likewise, the topic as it applies to the Gadget should resolve the *module-name* key as "Widget" so that the paragraph reads "If you also choose to integrate the Widget, many more options are available to you." Again, if someone is reading the output as produced for the Gadget, they obviously already have the Gadget but they might also want to know that there is a Widget available.

Interaction between scoped keys and ditaval filtering

This example also brings up the interesting question of how scoped keys and ditaval filtering—also introduced in DITA 1.3—are going to interact. Again, a full discussion of this interaction is outside the scope of this article. Briefly, though, say a map branch specifies two ditaval files that result in two copies of a topic, each filtered using the conditions specified in the associated ditaval file, such as this example, taken from the DITA 1.3 specification:

```
<topicref href="productFeatures.dita" keys="features" keyscope="prodFeatures">
   <ditavalref href="novice.ditaval"/>
   <ditavalref href="admin.ditaval"/>
   <topicref href="newFeature.dita" keys="newThing"/>
</topicref>
```

Post *ditavalref* processing, there will be two copies of both **productFeatures.dita** and **newFeature.dita**, each with different content based on the associated ditaval filtering.

The conflict is that if another topic references the *features* key, for example, it's not apparent which copy of productFeatures.dita should be the target.

New *ditavalref* metadata elements have been included to allow you to specify prefixes or suffixes to be added to the start and end of resource or key scope names for each resource or key scope in the branch. These prefixes and suffixes create predictable resource names and key scopes for each copy of a branch that is filtered using the conditions specified by the associated ditavals. The prefixed or suffixed key scope names can be used to explicitly refer to the copy of the topic created by application of the corresponding *ditavalref*.

A more complete explanation is available in the DITA 1.3 specification on *the OASIS website*.

The relationship, or rather, lack of relationship, between keys in parallel key spaces should be clear at this point. Let's take a look now at a more complex use case—that of nested key spaces.

Nested key spaces

Nested key spaces represent a more complex use of scoped keys, and it's critical for content creators and Information Architects to understand how nested key spaces interact.

In the following example, the root map is within a key space named "Keys1." The sub-map is within a key-space named "Keys2." Within the sub-map, the *topicgroup* is within a key space named "Keys3."

```
<map keyscope="Keys1">
  <keydef keys="module-name">
    <topicmeta>
      <keywords>
        <keyword>Widget</keyword>
      </keywords>
    </topicmeta>
  </keydef>
  <map keyscope="Keys2">
    <keydef keys="module-name">
      <topicmeta>
        <keywords>
          <keyword>Gadget</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <topicgroup keyscope="Keys3">
      <keydef keys="module-name">
        <topicmeta>
          <keywords>
            <keyword>Doodad</keyword>
          </keywords>
        </topicmeta>
      </keydef>
      <topicref href="get-started.dita"/>
    </topicgroup>
  </map>
</map>
```

Within "Keys1," the *module-name* key is defined as "Widget." Within "Keys2," the same key is defined as "Gadget." Within "Keys3," the key is defined as "Doodad."

The topic **get-started.dita** contains the following paragraph: The <ph keyref="module-name"/> will give you years of reliable use.

Intuitively, one might conclude that the *module-name* key should ultimately be resolved to "Doodad" in the output, because it is the key definition closest to the topicref. It's not illogical to think that the key is first resolved to "Widget," and then that definition is overridden when the processor encounters the "Gadget" definition and then **that** definition is overridden when the processor encounters the "Doodad" definition.

While this is a logical thought process, it is not what happens. The key actually resolves to "Widget."

To understand why, keep in mind that with the introduction of scoped keys, there is a mandate to retain complete backwards-compatibility with keys as they functioned in DITA 1.2. Recall that in DITA 1.2, the "first" definition of a key within a root map was the one that held throughout the publication. Many documentation teams have a large body of content that was designed around this principle. To require re-architecting of that content to accommodate a different method of key resolution would be quite burdensome. The architectural complexity of managing nested or overlapping key spaces in heavy reuse situations would also be quite considerable.

The point to remember is that if the same key is defined at both a parent and child level, the parent definition always wins.

Sharing keys between key spaces

There are theoretically two approaches to sharing keys between key spaces. The first is to simply allow a key definition at a higher level to be "inherited" at a lower level if no lower-level definition is present. This approach is not possible with scoped keys as implemented in DITA 1.3, as explained previously in the "Nested key spaces" section: ("if the same key is defined at both a parent and child level, the parent definition always wins") and in more detail in the following section, "Using inherited or 'fallback' key definitions." The second approach is to explicitly include the name of the key space when referencing a key, and this approach is accommodated by the scoped key functionality. The following sections take a look at both approaches.

Using inherited or "fallback" key definitions

The fact that a key definition at a higher level always takes precedence over one at a lower level would seem to prevent the use of inherited, or fallback key definitions, and that is indeed the case. By "fallback," we mean the following situation:

You have used keys widely in your content. You now have a very large root map, and you are not sure exactly which keys have been used or if they have all been properly defined, but to make sure there are no empty spots in your output, you've created a "master keydef map" of all your keys with "fallback" values. Your intention is to add this master keydef map to the root map so that any keys not explicitly defined elsewhere can pick up a fallback value from the master keydef map. As we have seen, this approach does not work. The key values in the master keydef map override any other values for the same keys defined elsewhere in the root map or sub-maps.

Here's an example.

The root map defines a key, *trim*, with a value of "TR". The second sub-map defines the same key with a value of "XR". Of course, the trusty *module-name* key is defined in both sub-maps as well.

```
<map>
  <title>Training Courses</title>
  <keydef keys="trim">
    <topicmeta>
      <keywords>
        <keyword>TR</keyword>
      </keywords>
    </topicmeta>
  </keydef>
  <mapref href="widget.ditamap" keyscope="widget"/>
    <keydef keys="module-name">
      <topicmeta>
        <keywords>
          <keyword>Widget</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <keydef keys="version">
      <topicmeta>
        <keywords>
          <keyword>4.2</keyword>
        </keywords>
      </topicmeta>
    </keydef>
  <mapref href="gadget.ditamap" keyscope="gadget"/>
    <keydef keys="module-name">
      <topicmeta>
        <keywords>
          <keyword>Gadget</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <keydef keys="trim">
      <topicmeta>
        <keywords>
          <keyword>XR</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <topicref href="get-started.dita"/>
</map>
```

Within the **get-started.dita** topic in the second sub-map, there is a paragraph that references both the *module-name* and *trim* keys:

```
<p>The <keyword keyref="module-name"/><keyword keyref="widget.trim"/> does everything but walk your dog.</p>
```

In this example, the "TR" definition of the *trim* key was meant to be only a fallback; the "XR" definition, added specifically to the sub-map that references the **get-started.dita** topic, was the one that the Information Architect intended to be used. As we now understand, the parent, or highest-level definition of the *trim* key takes precedence over any subsequent definitions of the key and so it is actually the "TR" value that is used throughout this publication.

Using the key space name in the key reference (scope-qualification)

While you cannot share keys between key spaces by inheriting keys from a parent key space as a fallback, you can share keys between parallel key spaces by explicitly referring to the key space.

```
<map>
  <title>Training Courses</title>
  <mapref href="widget.ditamap" keyscope="widget"/>
    <keydef keys="module-name">
      <topicmeta>
        <keywords>
          <keyword>Widget</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <keydef keys="version">
      <topicmeta>
        <keywords>
          <keyword>4.2</keyword>
        </keywords>
      </topicmeta>
    </keydef>
  <mapref href="gadget.ditamap" keyscope="gadget"/>
    <keydef keys="module-name">
      <topicmeta>
        <keywords>
          <keyword>Gadget</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <keydef keys="trim">
      <topicmeta>
        <keywords>
          <keyword>XR</keyword>
        </keywords>
      </topicmeta>
    </keydef>
    <topicref href="get-started.dita"/>
</map>
```

In this example, within the **get-started.dita** topic, there is a paragraph that references both the *module-name* and *version* keys:

```
<p>The <keyword keyref="module-name"/><keyword keyref="version"/> does everything but walk your dog.</p>
```

When this map is output, the *version* key will be unresolved. Why? Because there is no definition for the *version* key in the "gadget" key space. Remember that there is a "fence" around the second sub-map, created by the definition of the "gadget" key space. No key definitions from other key spaces can get inside this fence, and no key definitions from inside this fence can escape to other key spaces. By default, the reference to the *version* key in the **get-started.dita** topic cannot see the definition for the *version* key that is within the "widget" key space.

There is, however, a way to make key definitions in one key space visible to key references in another key space—by including the name of the key space in the key reference:

```
<p>The <keyword keyref="module-name"/><keyword keyref="widget.version"/> does everything but walk your dog.</p>
```

Because the name of the key space, "widget," is in the key reference, a compliant processor can locate that key space and within it, locate the referenced key. It's analogous to including the path in a link to ensure the link can locate the referenced file.

And, just as a file path can be relative or full, the path to a key space can also be relative or full. Consider this example:

```
<map>
  <mapref keyscope="widget">
    <topichead keyscope="gadget">
      <topicgroup keyscope="doodad">
        <keydef keys="trim">
          <topicmeta>
            <keywords>
              <keyword>XR</keyword>
            </keywords>
          </topicmeta>
        </keydef>
      </topicgroup>
    </topichead>
  </mapref>
  <mapref>
    <topicref href="get-started.dita"/>
  </mapref>
</map>
```

In this (fairly improbable) example, the root map references two sub-maps. The "widget" key space is defined for the first sub-map. Within that map, a nested "gadget" key space is defined on the *topichead* and within the *topichead*, another nested key space, "doodad," is defined on the *topicgroup*. Within the "doodad" key space, the *trim* key is defined.

Within the second sub-map, the topic **get-started.dita** includes a paragraph that references the *trim* key:

```
<p>The <keyword keyref="widget.gadget.doodad.trim"/> line of tools offers the best value.</p>
```

Notice that the full path to the "doodad" key space is included in the key reference. It is not sufficient to simply include "doodad.trim" as the keyref because the "doodad" key space is not at the highest level in the map; it is nested within other key spaces. You must start at the shared parent level and navigate down to the correct key space.

Resolving duplicate keys in parent and child scopes

There is one more important use case to look at for shared keys. Take a look at this map:

```
<map>
   <topicgroup keyscope="widget">
      <keydef key="intro" href="get-started.dita"/>
   </topicgroup>
   <keydef key="widget.intro" href="welcome.dita"/>
</map>
```

Here, it appears the key resolution is ambiguous—for all practical purposes, the *intro* and *widget.intro* keys are the same thing in this scenario, yet they point to two different topics. In fact, there is no ambiguity.

The scope-qualified keys from any scope are taken to occur within the parent scope at the point of occurrence of the child scope, meaning that a key defined in a child scope overrides the same qualified key name that occurs later in the parent scope. What this means for the example above is that the *intro* key that occurs within the *topicgroup* (the child scope) is taken to occur at the *topicgroup* point within the root map—that is, before the *widget.intro* key definition.

Effectively, the map becomes this:

```
<map>
   <keydef key="widget.intro" href="get-started.dita"/>
   <topicgroup keyscope="widget">
```

```
        <keydef key="intro" href="get-started.dita"/>
    </topicgroup>
    <keydef key="widget.intro" href="welcome.dita"/>
</map>
```

Given this structure, and the fact that the "first" definition of a key always wins, it's clear that the *widget.intro* key will be resolved to **get-started.dita**.

Reuse implications for named key spaces

Not surprisingly, there are considerations when reusing a topic that contains a key reference with a named key space. For example, in another context where the topic **get-started.dita** is used, there might not be a "widget" key space with a nested "gadget" key space with a nested "doodad" key space that includes a *trim* key definition. Depending on the specific situation, it might be feasible to build out this key space hierarchy and create a nested *trim* key definition. More than likely though, this kind of architecture is a bit complex to reproduce for the sake of resolving a key (not to mention the added complexity when there are multiple keys with multiple hierarchies).

In such cases, or in any case where you want to avoid having to create a multi-level key space, you can simply define the key as "fully scope qualified" in the reused topic as the key name. For example, to reuse **get-started.dita** in a simple map without named key spaces, define the key with the name *widget.gadget.doodad.trim*:

```
<map>
  <keydef keys="widget.gadget.doodad.trim">
      <topicmeta>
         <keywords>
            <keyword>XR</keyword>
         </keywords>
      </topicmeta>
   </keydef>
   <topicref href="get-started.dita"/>
   <topicref href="instructions.dita"/>
   <topicref href="error-codes.dita"/>
</map>
```

Being able to define a key in this way makes it much easier to reuse topics that reference keys with key spaces in their names, but it does not take care of the need to know that a topic includes key references in the first place. This need is not unique to scoped keys or to DITA 1.3. It still falls to tool vendors to develop tools that help content creators identify keys within their content, and to Information Architects to develop best practices around key usage.

To name or not to name the root map key space?

As mentioned earlier, all root maps have an implicit key space; it's not necessary to explicitly name it. In fact, it might be a best practice not to do so. For example, the following map references two sub-maps, both of which define their own key space at the map reference level:

```
<map>
    <mapref keyscope="Keys1" href="map1.ditamap"/>
    <mapref keyscope="Keys2" href="map2.ditamap"/>
</map>
```

The Information Architect who created this map does not have access to map1.ditamap; he or she merely knows it must be a sub-map of the root map for this publication.

Looking at map1.ditamap, we see that it defines its own key space, which is unfortunately also named "Keys2."

```
<map keyscope="Keys2">
    ...
</map>
```

References to key names qualified with the "Keys2" key space will resolve to map1.ditamap, not map2.ditamap, and it might not be immediately obvious why, creating confusion for everyone.

Similarly, it's not difficult to imagine a situation where an Information Architect defines a key space for a map at the map reference level ("Keys2"):

```
<map>
  <mapref href="map1.ditamap" keyscope="Keys2"/>
  <mapref href="map2.ditamap"/>
</map>
```

while the same map (map1.ditamap) defines a different key space ("Keys3") for itself at the map level:

```
<map keyscope="Keys3">
   ...
</map>
```

In short, the best policy for avoiding potential double or conflicting key space definitions is simply never to define a key space on a map element and instead always define it on the map reference.

Cross-deliverable linking

There is one final use case to discuss for scoped keys: cross-deliverable linking.

It has been a goal of DITA content creators and Information Architects for a long time to find a mechanism that could reliably create links from one publication to another. For example, say a company uses two maps to create two independent help systems. There are numerous occasions when it makes sense to link from a topic in Help System One to a topic in Help System Two. Many groups have "rolled their own" solutions to this need, but there has been no out-of-the-box mechanism in DITA to facilitate it.

Scoped keys introduces that mechanism. Take a look at this simple scenario.

The **widget.ditamap** is used to create a help system for the Widget product. Of course, it includes a number of *topicref* elements, including one to **get-started-widget.dita**. It also includes a reference to **gadget.ditamap**, which is the map used to create the help system for the Gadget product. Notice that **gadget.ditamap** is referenced as a peer map and that it defines a key space for itself, "gadget."

```
<map>
  <title>Widget Online Help</title>
  <mapref href="gadget.ditamap" scope="peer" keyscope="gadget"/>
  <topicref href="get-started-widget.dita"/>
</map>
```

The **gadget.ditamap** also includes a number of *topicref* elements, as well as a key definition, which defines the key *integ-gadget* as the topic **integrating-gadget.dita**.

```
<map>
  <title>Gadget Online Help</title>
  <keydef keys="integ-gadget" href="integrating-gadget.dita"/>
</map>
```

The final piece of this example is the topic **get-started-widget.dita**. In this topic, we want to include a link to the **integrating-gadget.dita** topic, which we do by including a reference to the *integ-gadget* key:

```
<concept id="getstartwidget">
  <title>Getting started with the Widget</title>
  <conbody>
    <p>If you choose to <xref keyref="gadget.integ-gadget">integrate the Gadget
      module</xref>, many more options will be available to you.</p>
  </conbody>
</concept>
```

Because this key definition is available to **integrating-gadget.dita** by virtue of having been defined in **gadget.ditamap**, which in turn names the "gadget" keyspace, a processor can resolve the *xref* as <xref href="[some path]/integrating-gadget.dita"/> and perhaps ultimately as an active hyperlink.

The "perhaps" is important. The DITA 1.3 specification does not outline how a processor should implement generation of cross-deliverable links. Logically, a processor would create active links but that implementation is entirely up to the developers of the DITA Open Toolkit and other tools vendors. The DITA 1.3 specification simply outlines the basic mechanism for setting up the potential for cross-deliverable linking.

Takeaways for scoped keys

To summarize this feature article, here are the key (pun intended) points to remember.

- DITA 1.3 introduces a mechanism to define the same key with multiple values within a single root map: the *keyscope* attribute.
- You can define a key space on the entire root map, a sub-map, a *topichead*, *topicgroup*, or even an individual *topicref*.
- An unqualified key defined within a key space applies only within that key space.
- In the case of nested key spaces, definitions in the highest-level key space always win.
- A key reference in one key space can use a definition in another key space by including the name of the key space in the keyref (scope qualification).
- Reuse becomes more complex with scoped keys.
- Scoped keys include a mechanism for cross-deliverable linking.
- It is up to individual processors and tools to determine how to render cross-deliverable links.