Junos® Fundamentals Series

DAY ONE: NAVIGATING THE JUNOS XML HIERARCHY

You have a job to do. So understanding how to leverage the Junos XML hierarchy can help you to quickly and efficiently apply Junos automation.

By Tim Fiola

DAY ONE:
NAVIGATING THE JUNOS XML HIERARCHY

Understanding XML structure and XML hierarchies helps you to more efficiently use Junos automation and leverage native Junos XML capabilities. XML is useful for expressing data in a hierarchy because it uses a series of opening and closing tags to make it easy to navigate through a hierarchy to find specific data.

Junos uses XML natively. While not visible to most users, Junos uses an XML API to convert the CLI commands into XML, and the resulting XML output into a more readable CLI output. Junos automation communicates with Junos via this same XML API. It thereby offers a powerful and comprehensive toolset for interacting with and directing the Junos processes.

"This book provides a clear and concise understanding of the XML hierarchy with many examples that help lay a solid foundation about the basics of XML, it's hierarchy, and mechanics. This Day One book will expand your thinking about what Junos automation can do for you and how to efficiently apply it in your own organization."

Sean Watson, JNCIE-M #450, JNCIS-SEC, JNCIS-ER
Juniper Networks Professional Services Engineer

IT'S DAY ONE AND YOU HAVE A JOB TO DO, SO LEARN HOW TO:

- Understand what XML is and the properties of an XML hierarchy.
- Understand how Junos automation everages the native XML capabilities of the operating system.
- Understand and effectively use current and context nodes in your scripts.
- Effectively use predicates and XML axes to qualify which nodes your script selects.
- Create location path expressions to isolate the specific data that interests you.

Juniper Networks Day One books provide just the information you need to know on day one. That's because they are written by subject matter experts who specialize in getting networks up and running. Visit www.juniper.net/dayone to peruse the complete library.

Published by Juniper Networks Books

ISBN 978-1-936779-18-5
51200
9 781936 779185
7100 1234

Junos® Automation Series

Day One: Navigating the Junos XML Hierarchy

By Tim Fiola

Supplemental Junos automation information is available in the PDF version of this booklet:

JUNIPer NETWORKS®

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice. Products made or sold by Juniper Networks or components thereof might be covered by one or more of the following patents that are owned by or licensed to Juniper Networks: U.S. Patent Nos. 5,473,599, 5,905,725, 5,909,440, 6,192,051, 6,333,650, 6,359,479, 6,406,312, 6,429,706, 6,459,579, 6,493,347, 6,538,518, 6,538,899, 6,552,918, 6,567,902, 6,578,186, and 6,590,785.

Published by Juniper Networks Books
Writers: Tim Fiola
Editor in Chief: Patrick Ames
Copyediting and Proofing: Nancy Koerbel
Junos Program Manager: Cathy Gadecki

ISBN: 978-1-936779-18-5 (print)
Printed in the USA by Vervante Corporation

ISBN: 978-1-936779-19-2 (ebook)
Version History: v3 January 2011
3 4 5 6 7 8 9 10 #7100123

About the Author
Tim Fiola is a Network Engineer in Juniper Networks' Professional Services organization. He is JNCIE-M #419 and has over 6 years experience working with JUNOS devices.

Author's Acknowledgments
The author would like to thank all those who helped in the creation of this booklet. Roy Lee, the Junos automation Product Line Manager, saw the potential of turning my white paper on how to move around in an XML hierarchy into a full Day One book. Curtis Call did a fantastic job with the technical editing and offering suggestions for presentation of material. Patrick Ames assisted greatly in the presentation and clarification of the material. Cathy Gadecki, the Day One Series Editor, played a key role in starting this project and bringing it to a successful conclusion. Thank you all so much.

This book is available in a variety of formats at: www.juniper.net/dayone.
Send your suggestions, comments, and critiques by email to dayone@juniper.net.

Welcome to Day One

Day One booklets help you to start quickly in a new topic with just the information that you need on day one. The Day One series covers the essentials with straightforward explanations, step-by-step instructions, and practical examples that are easy to follow, while also providing lots of references for learning more.

Why Day One Booklets?

It's a simple premise – you want to use your Juniper equipment as quickly and effectively as possible. You don't have the time to read through a lot of different documents. You may not even know where to start. All you want to know is what to do on day one.

Day One booklets let you learn from Juniper experts, so you not only find out how to run your device, but where the short cuts are, how to stay out of trouble, and what are best practices.

What This Booklet Offers You

This fourth booklet in the Junos Automation series helps you to understand how Junos automation leverages the XML hierarchy and how to search the XML hierarchy for the specific information you are interested in. When you're done with this booklet, you'll be able to:

- ✓Understand what XML is and the properties of an XML hierarchy.
- ✓Understand how Junos automation leverages Junos software's native XML capabilities.
- ✓ Understand and effectively use current and context nodes in your scripts.
- ✓Effectively use predicates and XML axes to qualify which nodes your script selects.
- ✓ Create location path expressions to isolate the specific data that interests you.

What You Need to Know Before Reading

Before reading this booklet, you should be familiar with the basic administrative functions of the Junos operating system. This includes the ability to work with operational commands and to read, understand, and change the Junos configuration. The Day One booklets of the Junos Fundamentals series, and the training materials available on the Fast Track portal, can help to provide this background (see page 68 for these and other references).

Other things that you will find helpful as you explore these pages:

✓ Having access to a Junos device while reading this booklet is very useful as a number of practice examples reinforcing the concepts being taught are included in these pages. Most of the examples require creating or modifying a script and then running the script on a Junos device in order to see and understand the effect.

✓ The best way to edit scripts is to use a text editor on your local PC or laptop and then to transfer the edited file to the Junos device using a file transfer application. Doing this requires access to a basic ASCII text editor on your local computer as well as the software to transfer the updated script using scp or ftp.

✓ While a programming background is not a prerequisite for using this booklet, a basic understanding of programming concepts is beneficial.

✓ This is the fourth volume of the Junos Automation series. Reading the previous volumes is recommended but not necessarily required. This booklet assumes you are familiar with the concepts of Junos automation and the material covered in the previous publications of this Day One series.

Supplemental Appendix

If you're reading the print edition of this booklet, there's more pages available in the PDF version, which includes a supplemental appendix. Go to www.juniper.net/dayone and download the free PDF version of this booklet to get the additional content.

NOTE We'd like to hear your comments and critiques. Please send us your suggestions by email at dayone@juniper.net.

Chapter 1

Introduction to XML

Understanding XML structure and XML hierarchies is essential to producing Junos automation scripts and leveraging Junos's native XML capabilities. To assist you with this, this chapter will detail what XML is and how Junos automation leverages Junos's native XML capabilities. It will also examine the concept of a *node*, different node types, and what role nodes play in an XML hierarchy. Finally, this chapter demonstrates how to view and understand Junos output in its native XML format, the same type of format that Junos and Junos automation scripts work with.

What is XML and How is it Used?

XML stands for *extensible markup language*. It is useful for expressing data in a hierarchy because it uses a series of opening and closing tags to make it easy to navigate through a hierarchy to find specific data.

Junos uses XML natively. A Junos user typically will not notice this because Junos also uses an XML API (application programming interface) to convert user commands in the CLI into XML and the resulting XML output back into a more readable CLI output. Junos automation scripts communicate with Junos via the same XML API, allowing the scripts to leverage Junos's native XML capabilities.

An XML object has the following basic components: opening tag, contents, closing tag. Sample 1.1 shows a very basic sample XML hierarchy in Junos. This hierarchy shows a series of nodes, with each node delimited with a set of opening and closing tags. For instance, the node `system` begins with an opening tag of `<system>` and ends with a closing tag of `</system>`. Everything between the opening and closing tags are the contents of the `system` node. XML clearly defines a set of nodes, where each node begins and ends, the contents (elements) of each node, and shows a clear path to any element within the hierarchy.

Sample 1.1 A Sample XML Hierarchy

```
<configuration>
    <version>9.4R3.5</version>
    <system>
      <host-name>r1</host-name>
      <syslog>
```

```
        <file inactive="inactive">
          <name>messages</name>
          <contents>
            <name>any</name>
            <info/>
          </contents>
        </file>
      </syslog>
    </system>
</configuration>
```

Now imagine that you are trying to describe how to get to the node `<contents>` within the sample XML hierarchy in Sample 1.1. Because each node in this architecture is clearly defined with opening and closing tags, you can describe the *path* to the `contents` node as `/configuration/system/syslog/file/contents`. In plain English, this path means:

1. Start at the `root`,
2. Look in the node `<configuration>` for node `<system>`,
3. Look in the node `<system>` to find node `<syslog>`,
4. Look in the node `<syslog>` to find the `<file>` node,
5. Look in the `<file>` node and find the `<contents>` node.

Since each node's beginning and end is clearly marked, it's easy locate a specific node and its contents.

NOTE The path `/configuration/system/syslog/file/contents` is known as a *location path expression*. Location path expressions will be covered in-depth in Chapter 2.

XML's power is its ability to leverage these opening and closing tags to make it simple to find the desired information in an XML document.

MORE? For more information on XML and how it applies to Junos automation, see the booklet *Day One: Applying Junos Operations Automation*, Chapter 1: XML Basics.

Node Ancestry

Since XML is hierarchical, it is useful to understand a specific node's location in the hierarchy relative to other nodes. This makes it easier to quickly describe a node and the other nodes' relative location to that node. Look again at the sample hierarchy in Sample 1.1. Examine the node relationships relative to the node `<contents>`.

- Parent and Child: `<contents>` lies directly below `<file>` in the hierarchy, between the opening and closing tags `<file>` and `</file>`, respectively. This makes `<file>` the *parent* of `<contents>`. Likewise, `<contents>` is the *child* of `<file>`.

- Sibling: the node `<name>` is also a child of `<file>`. This makes the `<name>` node a *sibling* node of `<contents>` since both the `<contents>` and `<name>` nodes have the same parent.

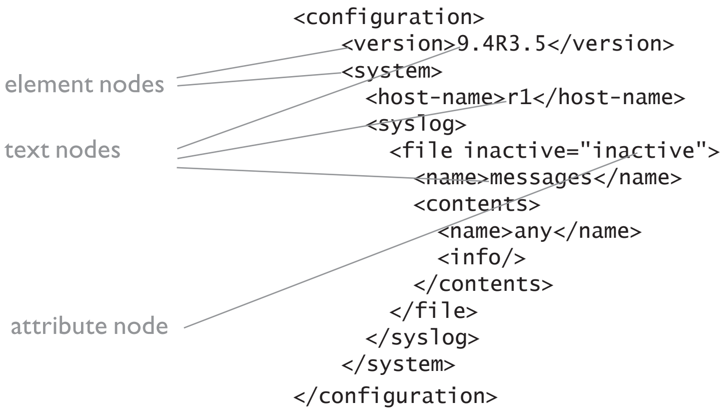
- Ancestor and Descendant: the parent of the node `<contents>`, node `<file>`, lies within `<syslog>` in the hierarchy, between the opening and closing tags `<syslog>` and `</syslog>`, respectively. Node `<syslog>` is the parent of `<file>` and `<file>` is the parent of `<contents>`. This makes node `<syslog>` an *ancestor* node of `<contents>`. Conversely, `<contents>` is a *descendant* of `<syslog>`. A child node's parent is also its *ancesto*r; a parent node's child is also its *descendant* node.

- Root: not shown explicitly in the XML architecture is the `root` node. Assuming that the architecture shown in Sample 1.1 comprises the entire hierarchy, the root node is a parent of `<configuration>` and an ancestor to all other nodes. Accordingly, the node `<configuration>` is a child of `root` and all other nodes are descendants of `root`.

Node Types

There are several different types of nodes commonly encountered in Junos automation. Sample 1.2 contains examples of the node types commonly found in Junos automation using the sample XML hierarchy found on the previous page.

Sample 1.2 **A Sample XML Hierarchy**



```
<configuration>
    <version>9.4R3.5</version>
    <system>
      <host-name>r1</host-name>
      <syslog>
        <file inactive="inactive">
          <name>messages</name>
          <contents>
            <name>any</name>
            <info/>
          </contents>
        </file>
      </syslog>
    </system>
</configuration>
```

Root

The <root> is the beginning of any absolute path and provides access to any point in an XML hierarchy. It has no parent node and can have child nodes of the *element* and *comment* types.

Element nodes

Element nodes are the contents of a parent node. Element nodes can have other element nodes, text nodes, or comment nodes as children. Element nodes can be children of <root> or children of other element nodes. In the example above, the node <configuration> has elements that include <version> and <system>.

Text/leaf nodes

Text nodes (sometimes called *leaf* nodes) are alphanumeric character strings such as 9.4R3.5, messages, and r1 in Sample 1.2. Text nodes cannot have child nodes.

CAUTION Any white space between nodes will become a text node as well, containing newlines, tabs, or spaces (whatever is separating the nodes).

Attribute nodes

Attribute nodes show descriptive attributes of a node they are paired with. In Sample 1.2, the element node `<file>` has an attribute node called `inactive`. Attribute nodes are somewhat unique in that an *element* node is the *parent* of each attribute node, but the attribute node is not a *child* of its parent element.

Other node types

Other XML node types exist as well, such as *namespace* nodes, *comment* nodes, and *processing instruction* nodes. These node types are rarely used in Junos automation and so are outside the scope of this Day One booklet.

NOTE An XML comment node is not the same thing as a comment in Junos created with the `annotate` configuration command.

Table 1.1 below summarizes node categories and possible parent and child node types for each node category.

Table 1.1 Node Types and Possible Node Ancestry

Node Type	Can have child nodes of type	Can be a child node of type
Root	Element, Comment	none
Element	Element, Text, Comment	Root, Element
Text	none	Element
Attribute	none	none*

*An element node can be a parent of an attribute node, but that attribute node is not a child node of the parent.

Tags, Nodes, Elements . . .

A few more words about tags, nodes, and elements might help put it all together. A *tag* is a syntax construct, like open tag (ex: `<system>`), close tag (ex: `</system>`), or an empty tag (ex: `<info/>`). *Elements* are the full contents between the open tag and close tag. A *node* can be thought of as a representation in the hierarchy of a specific opening tag, the associated closing tag, and all of its component elements.

In Sample 1.3 our same sample hierarchy is repeated again, this time highlighting examples of *tags*, *nodes*, and *elements*.

Sample 1.3 **Tags, Nodes, and Elements**

```
<configuration>
    <version>9.4R3.5</version>
    <system>
      <host-name>r1</host-name>
      <syslog>
        <file inactive="inactive">
          <name>messages</name>
          <contents>
            <name>any</name>
            <info/>
          </contents>
        </file>
      </syslog>
    </system>
</configuration>
```

system open **tag**

<system>'s **elements** are child and descendant nodes of <system>

The entire <system> **node** is in bold.

system close **tag**

Empty node

In the example above, `<info/>` is shorthand for `<info></info>`. There is nothing between the opening and closing tags. This is an *empty* node, meaning the element node has no child nodes.

Viewing the XML Hierarchy in Junos

It's quite easy to view the Junos configuration's XML hierarchy. To see the XML structure in a Junos configuration, simply add the `| display xml` option:

Operational mode:

```
> show configuration | display xml
```

Configuration mode:

```
# show | display xml
```

Using the `| display xml` option, the familiar configuration stanza names display in a slightly different format. Sample 1.4 shows the normal CLI results of a configuration stanza. Sample 1.5 shows the equivalent XML output. Notice the nodes nested beneath other nodes in a hierarchical format in Sample 1.5.

NOTE When using the `| display xml` command, you can ignore the <rpc-reply> node because most, if not all, Junos automation scripts act within the `<configuration>` node and do not need to act at the <rpc-reply> level. Chapter 2 covers the `<rpc>` and `<rpc-reply>` nodes in greater depth

Sample 1.4 A Junos Configuration Stanza

```
[edit]
ps@r1# show system login user ps
uid 2002;
class super-user;
authentication {
   encrypted-password "$1$x7uUz4Wj$oQynzLmlHr1EbWIJukWOh.";
## SECRET-DATA
}
```

Sample 1.5 A Junos Configuration Stanza Showing the XML Hierarchy

```
ps@r1# show system login user ps | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.4R3/junos">
   <configuration junos:changed-seconds="1258499128" junos:changed-
localtime="2009-11-17 23:05:28 UTC">
         <system>
            <login>
               <user>
                  <name>ps</name>
                  <uid>2002</uid>
                  <class>super-user</class>
                  <authentication>
                     <encrypted-password>$1$x7uUz4Wj$oQynzLmlHr1EbWIJukWOh.</
encrypted-password>
                  </authentication>
               </user>
            </login>
         </system>
   </configuration>
   <cli>
      <banner>[edit]</banner>
   </cli>
</rpc-reply>
```

system opening **tag**

`<system>` contents consists of additional descendant nodes

system closing **tag**

In Sample 1.5 the user requested to view the configuration information specifically for the `ps` user. When displaying the XML formatting, Junos shows all the ancestor nodes to the specific location.

Try It Yourself: Viewing the XML Hierarchy

1. Run the command `show configuration interfaces lo0`.
2. Run the command `show configuration interfaces lo0 | display xml`.
3. Compare the output of the two commands.
    a. Do the two versions of output contain the same information?
    b. Which version of output is easier to read?
    c. Which version of output is easier to use to describe where to find the interface's IP address?

Additionally, a Junos user can view the contents of a Junos operational command in XML format. Sample 1.6 below shows the familiar CLI output of the Junos operational command `show ospf interface`.

Sample 1.6 A Junos Operational Command's Output

```
ps@r1> show ospf interface
Interface         State   Area          DR ID        BDR ID        Nbrs
ge-1/1/0.0        DR     0.0.0.2        10.0.6.2      0.0.0.0          0
lo0.0            DRother 0.0.0.1        0.0.0.0       0.0.0.0          0

ps@r1>
```

When the same operational command is given with the `| display xml` qualifier, the output contains the same information, but looks strikingly different. See Sample 1.7 for the equivalent output in the XML format. Again, the XML tags are all present, clearly identifying each node in the output. The XML hierarchy in the output makes it possible to parse the information to isolate only the information you are interested in.

Sample 1.7 A Junos Operational Command Showing the Output in XML Format

```
ps@r1> show ospf interface | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.4R3/
junos">
   <ospf-interface-information xmlns="http://xml.juniper.net/
junos/9.4R3/junos-routing">
      <ospf-interface>
         <interface-name>ge-1/1/0.0</interface-name>
         <ospf-interface-state>DR</ospf-interface-state>
         <ospf-area>0.0.0.2</ospf-area>
         <dr-id>10.0.6.2</dr-id>
         <bdr-id>0.0.0.0</bdr-id>
         <neighbor-count>0</neighbor-count>
      </ospf-interface>
      <ospf-interface>
         <interface-name>lo0.0</interface-name>
         <ospf-interface-state>DRother</ospf-interface-state>
         <ospf-area>0.0.0.1</ospf-area>
         <dr-id>0.0.0.0</dr-id>
         <bdr-id>0.0.0.0</bdr-id>
         <neighbor-count>0</neighbor-count>
      </ospf-interface>
   </ospf-interface-information>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>

ps@r1>
```

Try It Yourself: An Operational Command's XML Output

1. Run the command `show interface lo0`.
2. Run the command `show interface lo0 | display xml`.
3. Compare the output of the 2 commands. Is the same information contained in each version of output?
4. What is the location path expression to get to the physical interface name for the lo0 interface in the XML output?

Summary

This chapter quickly covered XML basics and how Junos automation leverages Junos's native XML capabilities. You should know what node ancestry is, and how to explain nodes' relationships to each other depending on their position in the XML hierarchy. You should also know the different varieties of nodes found in Junos automation. Finally, you should know how to view and understand the XML output for Junos configurations and operational commands.

These basics will help you better understand the concepts presented in the following chapters. Return to this chapter at any time to review the basics.

Chapter 2

Location Path Expressions

XML's power is its ability to precisely locate specific information within a hierarchy, whether the hierarchy is simple or relatively large and complex. *Location path expressions*, a type of *XPath,* are the means by which a user can describe where to find specific data within the XML hierarchy.

MORE? For more information about XPath, see the XPath specification at http://www.w3.org/TR/xpath/. Junos automation scripts support XPath version 1.

Location Path Expressions: XML's Driving Directions

Let's quickly examine the Junos XML hierarchy that was shown in Sample 1.5 again, repeated here as Sample 2.1 for ease of reference.

Sample 2.1 A Junos Configuration Stanza Showing the XML Hierarchy

```
ps@r1# show system login user ps | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.4R3/junos">
   <configuration junos:changed-seconds="1258499128" junos:changed-
localtime="2009-11-17 23:05:28 UTC">
         <system>
            <login>
               <user>
                  <name>ps</name>
                  <uid>2002</uid>
                  <class>super-user</class>
                  <authentication>
                     <encrypted-password>$1$x7uUz4Wj$oQynzLmlHr1EbWIJukWOh.</
encrypted-password>
                  </authentication>
               </user>
            </login>
         </system>
   </configuration>
   <cli>
      <banner>[edit]</banner>
   </cli>
</rpc-reply>
```

If the `<name>` node is the node of interest, there has to be a way to describe how to reach that node within this hierarchy. Starting from the `<rpc-reply>` node, look within `<configuration>`, then look within the `<system>` child node. The `<system>` node contains the `<login>`

node, which in turn contains the `<user>` node. Within `<user>`, there are several sibling nodes. Look for the `<name>` child node. This process *steps* deeper into the hierarchy, looking for our data.

The ordered list of nodes to pass through to reach `<name>` is:

```
(root), <rcp-reply>, <configuration>, <system>, <login>, <user>,
<name>
```

A *location path expression* uses the / operator to step further within the hierarchy to the specific node of interest. In this example, then, the location path expression would read:

```
 /rpc-reply/configuration/system/login/user/name
```

Setting Context: The Current Node and Context Node

At this point it is beneficial to understand the concepts of the *current* and *context* nodes in order to create proper location path expressions that will search for the data that you are interested in.

The *current* node is the point in the XML hierarchy that the script's `for-each()` flow control or template structures use as a reference point. The *context* node is the point in the XML hierarchy that the script is acting on.

Initially the context and current node are the same. However, when a location path expression is introduced, the context node and current node can differ.

Current Node

The *current node* is a reference point in the XML hierarchy. The content within `for-each(){}` stanza or template can use the current node (instead of the `root` node) as a starting point for their location path expressions. At the beginning of a `for-each(){}` stanza or a template, the current node and the context node are identical. In order to understand that previous statement better, examine the boilerplates for commit and op scripts. This chapter examines commit scripts first.

By default, a commit script will receive the device's candidate configuration as part of the commit script's XML source tree. The output in Sample 2.2, taken from a commit script log file configured under `traceoptions`, shows this input. Note that the `<configuration>` node's parent is `<commit-script-input>`.

Sample 2.2 Commit Input Configuration to a Commit Script – as Shown in Log File

```
Nov 19 22:49:14 begin dump
<?xml version="1.0"?>
<commit-script-input xmlns:junos="http://xml.juniper.net/
junos/*/junos">

<configuration junos:changed-seconds="1258617706"
junos:changed-localtime="2009-11-19 08:01:46 UTC">
<version>9.4R3.5</version>
<system>
<host-name>r1</host-name>
<root-authentication>
<encrypted-password>$1$tMMaImCO$l0LXIOpZ0eIic2TvbOOud0</
encrypted-password>
</root-authentication>
<scripts>
. . .
. . .
. . .
```

The commit script boilerplate shown in Sample 2.3 contains the `match configuration { }` stanza. This stanza sets the initial current node (and context node) in a commit script to `/commit-script-input/configuration` in the XML source tree, with the `<configuration>` node containing the device’s candidate configuration.

Sample 2.3 The Match Configuration {} Stanza in Commit Script Boilerplate

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";
match configuration {
    /*
        * Insert your code here.
    */
}
```

Initially sets current and context nodes in the main template to `/commit-script-input/configuration`

Try It Yourself: Viewing Commit Script Input

To view the actual input for a commit script for yourself, configure the following:

1.Configure traceoptions under [system scripts commit] to flag input. The file name can be of your choosing:

```
ps@r1> show configuration system scripts commit
traceoptions {
   file commit-scripts;
   flag input;
}
```

2. Clear the configured log file (this clears out any extraneous log entries that may be in the file so it is easier to view the script input)

3. Configure a commit script to run

4. Perform a commit

5. Run the following CLI command:

```
ps@r1> show log commit-scripts
Nov 19 22:49:11 script-compy clear-log[14466]: logfile cleared
Nov 19 22:49:13 cscript script processing begins
Nov 19 22:49:14 reading commit script configuration
Nov 19 22:49:14 testing commit script configuration
Nov 19 22:49:14 commit script input
Nov 19 22:49:14 begin dump
<?xml version="1.0"?>
<commit-script-input xmlns:junos="http://xml.juniper.net/junos/*/junos">

<configuration junos:changed-seconds="1258666164" junos:changed-localtime="2009-11-19
21:29:24 UTC">
<version>9.4R3.5</version>
<system>
. . .
. . .
```

By default, op and event scripts don't initially receive the device's configuration as input. The `match /` statement in the op and event-script boilerplate sets the current/context node to the `root` node of the script's source tree. See Sample 2.4 for the op and event script boilerplate.

Sample 2.4 Match/ Statement in Op and Event Script

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {

    <op-script-results> {

  /*
         * Insert your code here.
    */
    }
}
```

The current and context nodes in a script can be changed at points within a script via a `for-each()` flow control loop. Within op scripts it is very likely that, in addition to changing the current/context nodes, the current/context nodes will be in a completely different data structure than that of the initially empty source tree. This chapter examines that concept in a later section titled *Background on the XML Structures Used In Junos Automation*. For now, understand that the op script starts with an empty source tree, and typically must import additional XML data structures (such as the device's configuration). Sample 2.5 shows a simple op script, with a `for-each()` loop. With each iteration of the loop, the current and context nodes change from `root` to each unique value for `/configuration/interfaces/interface.`

NOTE Junos code versions released before 9.3S7, 9.4R4, 9.5R4, 9.6R3, 10.0R2, and 10.1R1 return the `<junoscript>` and `<rpc-reply>` nodes in the variable's XML structure, making the path look like `/junoscript/rpc-reply/configuration/interfaces/interface`. And while that is technically correct, the `<junoscript>` and `<rpc-reply>` nodes are not useful in Junos automation itself. This is discussed later in this chapter.

NOTE In the examples below, the variable `$configuration` contains the router's configuration in an XML hierarchical format, so it is able to be parsed. A later section in this chapter, *Background on the XML Structures Used In Junos Automation,* covers variables that contain XML hierarchical information.

Sample 2.5 Current versus Context Node Example 1

1. Initially sets the current and context nodes in the op script to the root of the script's source tree

2. The for-each() loop sets the current node and context node to: /configuration/interfaces/interface

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {

   <op-script-results> {
    var $rpc-config-req = <get-configuration
database="committed" inherit="inherit">;
    var $configuration = jcs:invoke($rpc-config-req);

    for-each($configuration/interfaces/interface) {
      <output> name;
    }
  }
}
```

Sample 2.6 shows an example configuration hierarchy for the script in Sample 2.5 with the current (and context) node highlighted in **bold**. The reference point (current node) within the `for-each()` loop is the `<interface>` node. The node `<name>` is resolved in the context of `<interface>`, so at this point the context node is the same as the current node.

Sample 2.6 Hierarchy Example for Sample 2.5

The `for-each()` loop sets the current and context nodes in the op script to `<interface>`

`<name>` is resolve from the context of `<interface>`

```
<configuration>
  <interfaces>
    <interface>
      <name>ge-0/0/0</name>
      <unit>
            <name>100</name>
      </unit>
    </interface>
    <interface>
      <name>ge-0/0/1</name>
      <unit>
            <name>200</name>
      </unit>
    </interface>
  </interfaces>
</configuration>
```

Context Node

The *context* node is the point in the XML hierarchy that the script is acting on. Initially the current node and context node are the same node. The context node can be different from the current node if location path steps are introduced.

Look at Sample 2.7(see Sample 2.8 for the sample hierarchy that goes with Sample 2.7.) It is the same script as in Sample 2.5 except for the introduction of the location path expression `unit/name` within the `for-each()` loop. As in Sample 2.5, the initial current and context nodes are set to the `root` node of the source tree by the `match /` statement and the current and context nodes are changed to `<interface>` by the `for-each($configuration/interfaces/interface)` location path expression. However, within the `for-each()` loop in Sample 2.7, the `unit/name` location path expression changes the context node to `<unit>`, because `<name>` is now resolved from the context of `<unit>` (`/configuration/interfaces/interface/unit`). In Sample 2.5 `<name>` was resolved from the context of `<interface>` (`/configuration/interfaces/interface`). Location path expressions within templates and `for-each()` loops can make the context node different from the current node.

Sample 2.7 Current Versus Context Node Example 2

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {

    <op-script-results> {
    var $rpc-config-req = <get-configuration
database="committed" inherit="inherit">;
    var $configuration = jcs:invoke($rpc-config-req);

    for-each($configuration/interfaces/interface) {
      <output> unit/name;
    }
  }
}
```

1. Initially sets the current node in the op script to `root` of the script's source tree

2. The for-each() loop sets both the current node and context node to: `/configuration/interfaces/interface`

3. The `unit/name` step then sets the context node to `/configuration/interfaces/interface/unit` because the script is resolving `<name>` from the context of <unit>

The current node remains `/configuration/interfaces/interface`

Sample 2.8 Hierarchy Example for Sample 2.7

1. The for-each() loop sets the current and context nodes in the op script to <interface>

2. The <unit> step changes the context node from <interface> to <unit>, so the script resolves <name> from the context of <unit> instead of <interface>

```
<configuration>
       <interfaces>
              <interface>
                     <name>ge-0/0/0</name>
                     <unit>
                            <name>100</name>
                     </unit>
              </interface>
              <interface>
                     <name>ge-0/0/1</name>
                     <unit>
                            <name>200</name>
                     </unit>
              </interface>
       </interfaces>
</configuration>
```

It is important to understand that both the current and context nodes will return to <configuration> in a commit script and root (/) in an op script when the script exits a for-each() loop. Any further location path expressions in a commit script at a point outside of a for-each() loop need to start from the <configuration> node as the both current and context nodes. In the op script examples in Samples 2.5 and 2.7, any location path expressions outside the for-each() loop would start at the root node of the op script's source tree because that is the current and context node outside of the for-each() loop.

if() And else-if()

Unlike for-each() statements, location path expressions that are part of if(), else-if(), and else loops do not change the current or context nodes within the if(){} and else-if(){} stanzas. Within if() or else-if() statements, location path expressions in parentheses are evaluated just as they are in for-each() statements. However, in if() and else-if() statements, if the location path expression in parentheses returns an empty set of nodes (meaning the location path expression does not point to any existing nodes) then that expression returns a logical FALSE. If the location path expression within the parentheses returns a non-empty set of nodes (it points to one or more existing nodes), then that location path expression within parentheses evaluates to a logical TRUE. Think of it as resolving to if(TRUE) or if(FALSE).

Sample 2.9 below contains a commit script that adds a description to any `<unit>` child of `<interface>` child of `<interfaces>` child of `<configuration>` child of `<commit-script-input>` child of `root`. Notice the `if(interfaces/interface/unit)` statement and then the definition of `$dot` within the `if(){}` stanza. The path for `$dot` starts at `<interfaces>`, not `<unit>`. This is because the current and context nodes remain `<configuration>`, even inside the `if(interfaces/interface/unit){}` stanza. So, in order to make the configuration change at the appropriate `<description>` level, the location path expression that defines `$dot` must change the script's context node from `<configuration>` to `<unit>`. Remember, the `if (interfaces/interface/unit)` statement will evaluate to either `if (TRUE)` or `if (FALSE)` and will not change the script's current/context nodes.

Sample 2.9 Commit Script Example Using if() Statement

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match configuration {
  if (interfaces/interface/unit) {
    <xnm:warning> {
      call jcs:edit-path();
      <message> "adding description to logical unit";
        }
    call jcs:emit-change($dot = interfaces/interface/unit) {
             with $content = {
        <description> "this description put here by commit
script";
          }
       }
   }
}
```

More About Location Path Expressions

A location path expression can be either *absolute* or *relative*. An absolute location path begins with a /, indicating that the path begins at `root` and a relative location path does not begin with a /, and so it does not begin at `root`.

Absolute Location Path Expressions

Absolute location paths start at the root. In plain English, the sample absolute expression, `/configuration/protocols/bgp`, is saying:

1. start at `root`,
2. look for a `<configuration>` child node,
3. within `<configuration>` look for node `<protocols>` ,
4. and within `<protocols>` look for node `<bgp>`.

Or:

```
<configuration>
    <protocols>
      <bgp>
        . . .
        . . .
      </bgp>
      <ospf>
        . . .
        . . .
      </ospf>
    </protocols>
</configuration>
```

Relative Location Path Expressions

A relative location path does not begin with a `/`, and so it does not begin at `root`. Rather, a relative location path expression begins at the *context node*. Recall that the context node is the point in the XML hierarchy that the script is acting on. Let's use a commit script as an example.

Sample 2.10 shows a commit script that illustrates this point because it contains four simple relative location path expressions (shown in bold). In function, this commit script will list the first configured `<unit>`'s `<name>` for each configured `<interface>` node. Sample output is also included in Sample 2.10. A later section of this chapter, *Predicates,* discusses how to pick a specific unit of an interface when there are multiple units configured.

Sample 2.10 Commit Script Example Using Relative Location Path Expressions

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match configuration {
   for-each (interfaces/interface) {
    <xnm:warning> {
             <message> {
        expr "interface " _ name _ "'s first configured";
        expr " unit is " _ unit/name _ ".";
      }
    }
  }
}

[edit]
ps@r1# commit and-quit
warning: interface ge-0/0/1's first configured unit is 0.
warning: interface ge-1/0/1's first configured unit is 100.
warning: interface ge-1/0/2's first configured unit is 0.
warning: interface lo0's first configured unit is 0.
commit complete
Exiting configuration mode

ps@r1>
```

This chapter showed that the commit script's input is `/commit-script-input`. This is an absolute location path expression because it starts with a / and so it begins from `root`. In the commit script in Sample 2.10 the `match configuration` statement is a *relative* location path expression, because it does not have a `/` at the beginning. The current and context nodes are set to `/commit-script-input/configuration` when the script begins.

Within the `for-each ()` statement, `interfaces/interface` is another relative location path expression (current and context node set to `/commit-script-input/configuration/interfaces/interface`). The third relative location path expression is `name` (begins from the context node `<interface>`). The fourth relative location path expression is `unit/name`, also within the `for-each ()` loop (`unit/name` changed the context node to `/commit-script-input/configuration/interfaces/interface/unit` because `<name>` resolves from the context of `<unit>`).

An op script uses an absolute path to set the initial context and current node to the `root` node of the script's source tree (via the `match /` syntax). A commit script initially sets the current and context nodes to `/commit-script-input/configuration` via the `match configuration {}` syntax. In Sample 2.11, the `for-each()` loop modifies the current and context nodes from `/commit-script-input/configuration` to `/commit-script-input/configuration/interfaces/interface/vlan-tagging`.

Sample 2.11 Commit Script Example Showing A Change In Current And Context Nodes

Sets current and context nodes to `/commit-script-input/configuration`

Relative location path expression sets the current and context node in the for-each loop to `/commit-script-input/configuration/interfaces/interface/vlan-tagging`

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match configuration {

    var $limit = 3;
    for-each (interfaces/interface/vlan-tagging) {
    ....<snip>....
```

A Practical Example Using Relative Location Path Expressions

Op, event, and commit scripts can use location path expressions to act on a specific part of the XML configuration hierarchy. Sample 2.12 displays a simple configuration that only has a lo0.0 interface configured in the `interfaces` stanza.

Sample 2.12 A Simple Junos Interface Configuration In XML Form

```
ps@r1> show configuration interfaces | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.4R3/junos">
    <configuration junos:commit-seconds="1258539564" junos:commit-localtime="2009-11-18
10:19:24 UTC" junos:commit-user="ps">
          <interfaces>
             <interface>
                <name>lo0</name>
                <unit>
                    <name>0</name>
                    <family>
                       <inet>
                          <address>
```

```
                           <name>10.0.6.2/32</name>
                        </address>
                     </inet>
                  </family>
               </unit>
            </interface>
         </interfaces>
   </configuration>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>

ps@r1>
```

Sample 2.13 shows a fairly simple commit script that adds a specific description to the lo0.0 interface in the XML hierarchy in Sample 2.12. Commit scripts are provided in the device's configuration in XML format as part of the script's source tree. The first part of the script uses the relative location path expression `interfaces/interface/unit` in the `if()` statement to verify the existence of the logical interface. The second part of the script uses the same XPath to act specifically on the `interfaces/interface/unit` level in the hierarchy.

Sample 2.13 A Junos Commit Script that Acts on the Hierarchy in Sample 2.12

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";


import "../import/junos.xsl";

match configuration {

   if(interfaces/interface/unit) {
      <xnm:warning> {
         call jcs:edit-path();
         <message> "adding description to interface lo0 unit 0";
      }

      call jcs:emit-change($dot = interfaces/interface/unit) {
         with $content = {
            <description> "this description put here by commit script";
            }
         }
      }
}
```

Location path expressions

The output below shows how the configuration hierarchy has changed after the commit script adds the <description> node under the <unit> node (shown in bold). Compare this with Sample 2.12 and then with the XPath in the script in Sample 2.13.

```
ps@r1> show configuration interfaces | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.3I0/junos">
   <configuration junos:commit-seconds="1265104528" junos:commit-localtime="2010-02-02
09:55:28 UTC" junos:commit-user="ps">
         <interfaces>
            <interface>
               <name>lo0</name>
               <unit>
                  <name>0</name>
                  <description>this description put here by commit script</description>
                  <family>
                     <inet>
                        <address>
                           <name>10.0.6.2/32</name>
                        </address>
                     </inet>
                  </family>
               </unit>
            </interface>
         </interfaces>
   </configuration>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>

ps@r1>
```

Understanding XPath and creating accurate location path expressions is essential to creating effective and efficient Junos automation scripts.

The examples of the configuration and script in Samples 2.12 and 2.13 are very simple. A later section of this chapter, *Predicates,* discusses how to create scripts that can handle more practical configurations.

MORE? For more information about location path expressions, operators, and predicates, see the Junos *Configuration and Diagnostic Automation Guide* for the specific version of Junos you are using at: http://www.juniper.net/techpubs/en_US/junos/information-products/topic-collections/config-guide-automation/frameset.html

Background on the XML Structures Used in Junos Automation

It is helpful to understand the different XML structures used in Junos automation. Earlier, this chapter mentioned `<junoscript>` and `<rpc-reply>` nodes. Additionally, when the `| display xml` option is used at the CLI, the results return an `<rpc-reply>` node that contains the results. In practice, these nodes do not come into play in Junos automation scripts. However, it is useful to understand when and where they are used in order to firmly grasp some of the concepts of XML and how they affect Junos automation. The next section gives some background on the source tree for commit and op scripts and on the `<rpc>` and `<rpc-reply>` nodes and the purpose they serve in the returned XML data structures.

Commit Scripts

Recall that a commit script, by default, receives the device's candidate configuration in its source tree. A script's source tree is the XML data structure that is provided by Junos for the script when the script begins. So, a commit script's source tree is the candidate configuration within `/commit-script-input/configuration`. See Sample 2.14 below for a snip of the beginning of that source tree. The commit script boilerplate then matches on the `<configuration>` node.

Sample 2.14 Input Configuration to Commit Script – as Shown in Log File (Sample 2.2)

```
Nov 19 22:49:14 begin dump
<?xml version="1.0"?>
<commit-script-input xmlns:junos="http://xml.juniper.net/junos/*/junos">

<configuration junos:changed-seconds="1258617706" junos:changed-localtime="2009-11-19
08:01:46 UTC">
<version>9.4R3.5</version>
<system>
<host-name>r1</host-name>
<root-authentication>
```

```
<encrypted-password>$1$tMMaImCO$lOLXIOpZOeIic2TvbOOudO</encrypted-password>
</root-authentication>
<scripts>
. . .
. . .
. . .
```

The commit script boilerplate, shown below, contains the `match configuration` location path expression. That expression changes current and context node to `/commit-script-input/configuration`.

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match configuration {
  /*
    *Insert your code here.
  */

}
```

Op Scripts

Op script XML data structures can be a bit more complicated than those of commit scripts because an op script, by default, does not receive any input; the source tree is `/op-script-input`. This op script source tree contains no embedded information, unlike the commit script's source tree. Op scripts obtain information via remote procedure calls, which import additional XML data structures into the script. Each remote procedure call (rpc) generates a *junoscript* session to the mgd (management daemon). This session takes place within a `<junoscript>` node. The rpc itself is delivered to the mgd within the junoscript session using an `<rpc>` node. For example, when a script uses an rpc to get the device's configuration, the syntax may look like this:

```
var $rpc-config-req = <get-configuration>;
var $configuration = jcs:invoke($rpc-config-req);
```

Invoking that rpc generates a junoscript session with the mgd. The general XML structure of that session looks like this:

```
<junoscript>
```

```
<rpc>
    (--information request--)
</rpc>
</junoscript>
```

In the example above, the variable `$configuration` now contains the device's XML configuration hierarchy, and has this general structure (remember that there is an unseen root node above the `<junoscript>` node):

```
<junoscript>
<rpc-reply>
<configuration junos:commit-seconds="1264454710" junos:commit-localtime="2010-01-25
21:25:10 UTC" junos:commit-user="ps">
<version>9.3I0 [builder]</version>
<system>
<host-name>r1</host-name>
. . .
. . .
(--truncated for brevity--)
. . .
. . .
</rpc-reply>
</junoscript>
```

Although the `<junoscript>` and `<rpc-reply>` nodes are ancestors of `<configuration>`, the rpc-reply, by its nature, will return its child node to the script; in this case it returns `<configuration>`. That is why `<junoscript>` and `<rpc-reply>` do not appear in location path expressions within scripts written for Junos automation. In fact, Junos versions 9.3S7, 9.4R4, 9.5R4, 9.6R3, 10.0R2, and 10.1R1 and later do not return the `<junoscript>` and `<rpc-reply>` nodes in the variable's XML structure.

Look at the sample op script in Sample 2.15. The expression in the `for-each()` loop changes the current and context node from `/` (the initial source tree in the op script) to `/configuration/protocols/ospf` because the script is now parsing the XML structure of the `$configuration` variable, which is a completely separate XML data structure (complete with its own separate *root* node) than that of the op script's initial, empty source tree (`/`).

NOTE Event-scripts receive event information in their source tree but otherwise are like op scripts in that they don't receive any other information, such as the router's configuration.

Sample 2.15 Sample Op Script with for-each() Loop

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
    <op-script-results> {
     var $rpc-config-req = <get-configuration>;
         var $configuration = jcs:invoke($rpc-config-req);

     for-each($configuration/protocols/ospf) {
       . . .
       . . .
       (---truncated for brevity---)
     }
  }
}
```

Variables Containing XML Data Structures

Assigning the output from a Junos CLI command to a variable is a common technique when writing scripts for Junos automation. You may want to access the Junos device's configuration or the results of the `show interfaces terse` operational command. Taking the latter as an example, an easy way to accomplish it is to code something like the following:

```
  var $rpc-int-info-req = {
    <get-interface-information> {
      <terse>;
    }
  }

  var $int-info = jcs:invoke($rpc-int-info-req);
```

Now that the variable `$int-info` has the desired information, that variable is a separate XML structure from the script's source tree and, because it is an XML structure, it can be parsed for specific information.

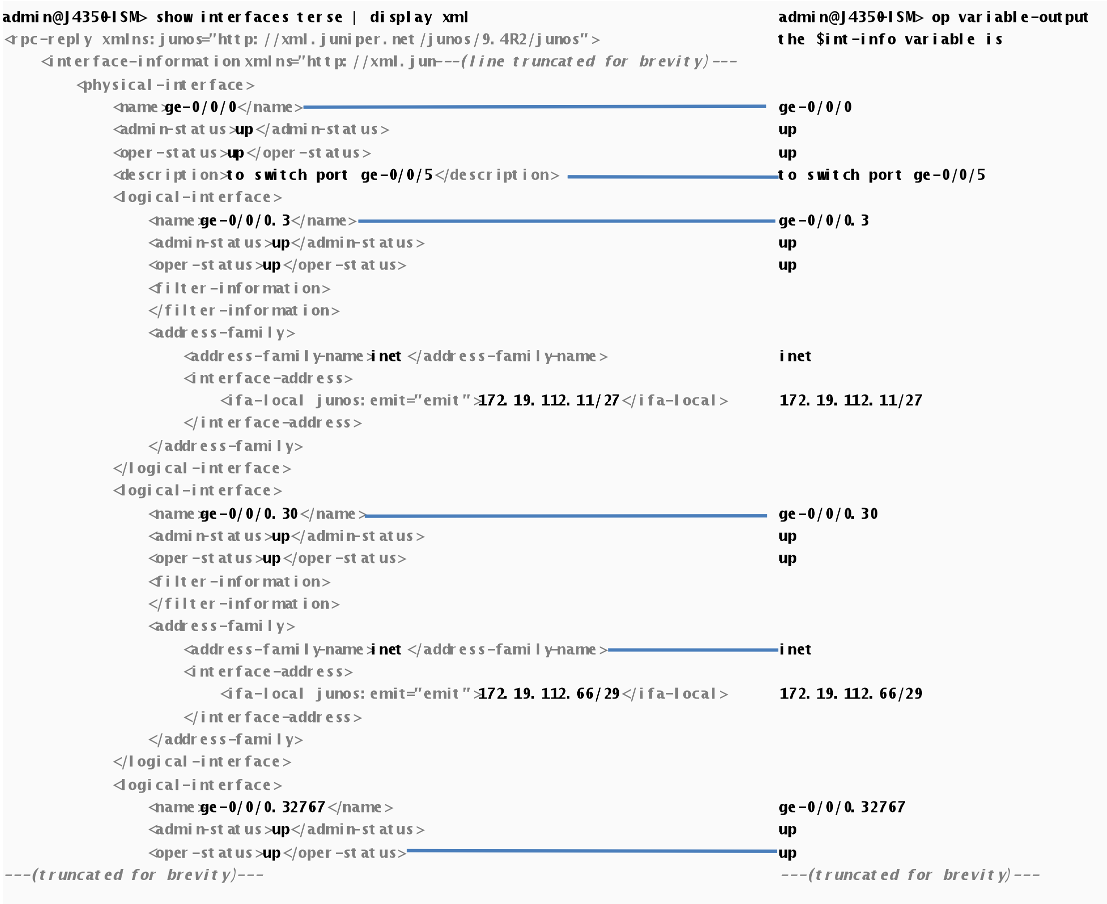
The `$int-info` variable in the example above has the same XML structure as the structure shown in the operational command `show interfaces terse | display xml`. To illustrate this point, examine the op script below. This script assigns the results of the <get-interface-information> rpc with the <terse> option to the variable $int-info and then displays $int-info's contents.

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
  <op-script-results> {
    var $rpc-int-info-req = {
      <get-interface-information> {
        <terse>;
      }
    }

    var $int-info = jcs:invoke($rpc-int-info-req);

    <output> "the $int-info variable is " _ $int-info;

  }
}
```

Sample 2.16 below shows the script's output side by side with the output of the show interfaces terse | display xml operational command. Some alignment lines are added to assist the reader in seeing how some common data lines up.

There are several things to take note of in this side-by-side comparison. The first item to note is that the output of the $int-info variable is missing the visible XML markers and that only the contents of the text nodes are visible. Additionally, notice that the contents of $int-info contain a lot of apparently empty lines. However, those apparently empty lines and text nodes correspond to the visible XML tags and text nodes in the output of show interfaces terse | display xml command.

Sample 2.16 Variable With XML Structure



```
admin@J4350-ISM> show interfaces terse | display xml                  admin@J4350-ISM> op variable-output
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.4R2/junos">     the $int-info variable is
   <interface-information xmlns="http://xml.jun---(line truncated for brevity)---
      <physical-interface>
         <name>ge-0/0/0</name>                                         ge-0/0/0
         <admin-status>up</admin-status>                               up
         <oper-status>up</oper-status>                                 up
         <description>to switch port ge-0/0/5</description>            to switch port ge-0/0/5
         <logical-interface>
            <name>ge-0/0/0.3</name>                                    ge-0/0/0.3
            <admin-status>up</admin-status>                            up
            <oper-status>up</oper-status>                              up
            <filter-information>
            </filter-information>
            <address-family>
               <address-family-name>inet</address-family-name>         inet
               <interface-address>
                  <ifa-local junos:emit="emit">172.19.112.11/27</ifa-local>   172.19.112.11/27
               </interface-address>
            </address-family>
         </logical-interface>
         <logical-interface>
            <name>ge-0/0/0.30</name>                                   ge-0/0/0.30
            <admin-status>up</admin-status>                            up
            <oper-status>up</oper-status>                              up
            <filter-information>
            </filter-information>
            <address-family>
               <address-family-name>inet</address-family-name>         inet
               <interface-address>
                  <ifa-local junos:emit="emit">172.19.112.66/29</ifa-local>   172.19.112.66/29
               </interface-address>
            </address-family>
         </logical-interface>
         <logical-interface>
            <name>ge-0/0/0.32767</name>                                ge-0/0/0.32767
            <admin-status>up</admin-status>                            up
            <oper-status>up</oper-status>                              up
---(truncated for brevity)---                                          ---(truncated for brevity)---
```

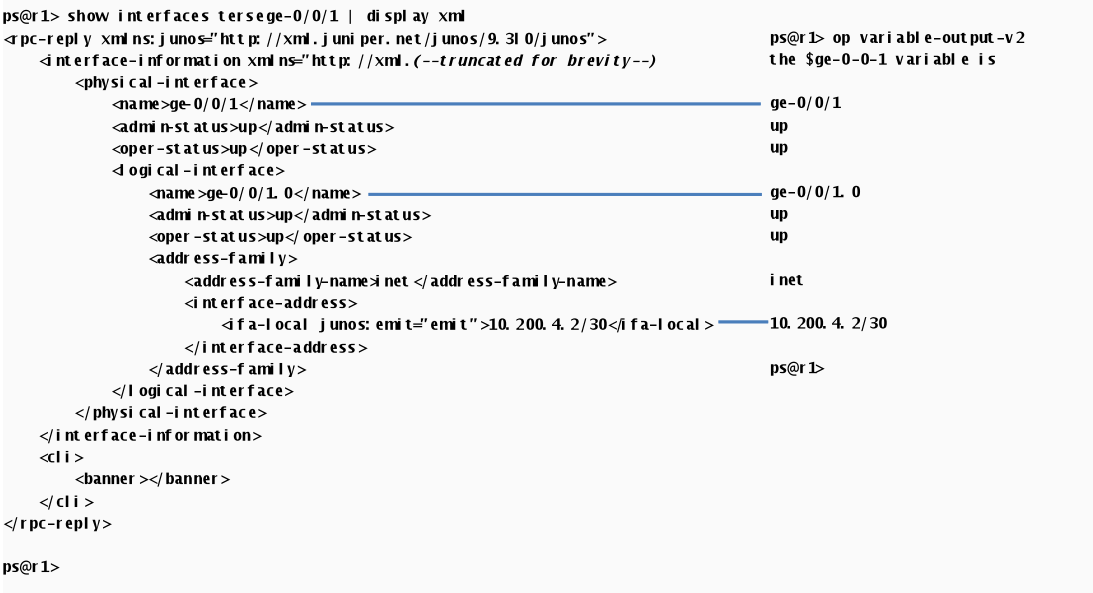
Looking at this side by side comparison it would appear that the empty lines in the `$int-info` variable structure may actually contain unseen XML tags. If the XML tags are in fact present, then the `$int-info` variable can be parsed. The next example demonstrates that that is indeed the case. Look at the script below and the output from the script.

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
import "../import/junos.xsl";

match / {
          <op-script-results> {
            var $rpc-int-info-req = {
              <get-interface-information> {
                <terse>;
              }
            }

            var $int-info = jcs:invoke($rpc-int-info-req);

            var $ge-0-0-1 = $int-info/physical-interface[name == 'ge-0/0/1'];

            <output> "the $ge-0-0-1 variable is \n" _ $ge-0-0-1;

          }
}
```

This script defines a new variable called `$ge-0-0-1`, which should contain only the information from the `$int-info` variable for interface `ge-0/0/1`. Sample 2.17 shows the output of the script side-by-side with the `show interfaces terse ge-0/0/1 | display xml` operational command. Alignment lines should assist the reader in lining up some common data.

Sample 2.17 Variable with XML Structure



```
ps@r1> show interfaces tersege-0/0/1 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.3I0/junos">                        ps@r1> op variable-output-v2
    <interface-information xmlns="http://xml.(--truncated for brevity--)                   the $ge-0-0-1 variable is
        <physical-interface>
            <name>ge-0/0/1</name> ----------------------------------------------------   ge-0/0/1
            <admin-status>up</admin-status>                                               up
            <oper-status>up</oper-status>                                                 up
            <logical-interface>
                <name>ge-0/0/1.0</name> ---------------------------------------------    ge-0/0/1.0
                <admin-status>up</admin-status>                                           up
                <oper-status>up</oper-status>                                             up
                <address-family>
                    <address-family-name>inet</address-family-name>                       inet
                    <interface-address>
                        <ifa-local junos:emit="emit">10.200.4.2/30</ifa-local> -------   10.200.4.2/30
                    </interface-address>
                </address-family>                                                         ps@r1>
            </logical-interface>
        </physical-interface>
    </interface-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

ps@r1>
```

Predicates

When XML structures become more complex, the parsing of the structure to select the nodes of interest becomes more complex as well. Recall from a previous section Sample 2.12. The interfaces configuration presented in Sample 2.12 was a very simple `interfaces` configuration and it was fairly easy to come up with the location path expression `interfaces/interface/unit` to get to `lo0.0` because `lo0.0` is the only `interface/unit` configured. Sample 2.18 adds more complexity to the mix by adding a configuration for interface `ge-1/0/0` as well as another logical unit to the existing `lo0` interface.

Sample 2.18 A Junos Interface Configuration in XML Form

```
ps@r1> show configuration interfaces | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.4R3/junos">
   <configuration junos:commit-seconds="1258616688" junos:commit-localtime="2009-11-19
07:44:48 UTC" junos:commit-user="ps">
          <interfaces>
             <interface>
                <name>ge-1/0/0</name>
                <unit>
                   <name>0</name>
                   <family>
                      <inet>
                         <address>
                            <name>192.168.142.100/24</name>
                         </address>
                      </inet>
                   </family>
                </unit>
             </interface>
             <interface>
                <name>lo0</name>
                <unit>
                   <name>0</name>
                   <family>
                      <inet>
                         <address>
                            <name>10.0.6.2/32</name>
                         </address>
                      </inet>
                   </family>
                </unit>
                <unit>
                   <name>10</name>
                   <family>
                      <inet>
```

```
                        <address>
                           <name>172.16.10.1/32</name>
                        </address>
                     </inet>
                  </family>
               </unit>
            </interface>
         </interfaces>
   </configuration>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>

ps@r1>
```

If your intent is to add a description only to interface lo0.0, will the commit script from Sample 2.13 succeed? The script in Sample 2.13 used the location path expression `interfaces/interface/unit`. When that expression is applied to the configuration in Sample 2.18, it immediately runs into a problem: there are now two `<interface>` nodes. The intent is to select the `<interface>` node with the `name` of `lo0`. There is another problem: once the script selects the appropriate `<interface>` node, how does it select the correct `<unit>` node within `lo0`? The script does not know which `<interface>` or `<unit>` node to select because the existing location path expression `interfaces/interface/unit` is too vague.

Predicates are the answer to our problem. Predicates are used in location path expressions to qualify the subset of nodes that should be selected in an XML hierarchy. Predicates always appear in square brackets `[ ]`. The expression within the brackets evaluate to a Boolean value. A *TRUE* result means that the candidate node is selected. If the evaluation returns a result of *FALSE*, then that node is not selected. This can be a very effective way of finding a specific node in the XML document.

The *TRUE* or *FALSE* result from within the brackets can be derived from a function, an operator, a conversion, or a combination of any of those. Here is a function within a predicate:

```
interfaces/interface[starts-with(name, ‘ge-‘)]
```

Here, the *starts-with()* function evaluates the text node of the `<name>` node within each `<interface>` node. If that text node begins with *ge-*, then the predicate expression within the brackets returns a *TRUE* result and the expression selects that specific `<interface>` node. If the text node does not begin with *ge-*, then the predicate expression returns *FALSE* and the expression does not select that specific node.

And here is an operator within a predicate:

```
interfaces/interface[mtu != 1600]
```

The *does not equal* operator *!=* is used to evaluate the text node of the `<mtu>` child node of each of the `<interface>` nodes that are children of `<interfaces>` to see if that value is not equal to 1600. If the result of that evaluation is *TRUE* (value is not equal to 1600), then that specific `<interface>` node is selected. A *FALSE* value will result in the specific `<interface>` node not being selected.

MORE? For more information on operators and functions, see *Day One: Applying Junos Operations Automation*, Chapter 3, available at www.juniper.net/dayone.

And here is a conversation within a predicate:

```
interfaces/interface[@inactive]
```

You can see that the predicate converts the `@inactive` location path expression to *TRUE* if the `<interface>` node has an `inactive` attribute node or *FALSE* if there is not an `inactive` attribute node present. The expression in brackets is an abbreviation for *attribute::inactive*. This expression searches for the `inactive` attribute of the context node (in this case, `<interface>`). If there is no `inactive` attribute present, the predicate expression resolves to *FALSE*. Chapter 3 covers the *attribute* axis and the expression *attribute::* in more depth.

Let's look at some more examples of predicates. They always appear in square brackets [ ].

More Predicate Examples

The following selects the `<interface>` node (whose `<name>` node has a text node containing *ge-0/0/0*) that is a child of the `<interfaces>` node:

```
interfaces/interface[name == "ge-0/0/0"]
```

```
<interfaces>
```

```
        <interface>
               <name>ge-0/0/0</name>
            . . .
        </interface>
        <interface>
               <name>ge-1/0/0</name>
            . . .
        </interface>
</interfaces>
```

The following selects the third listed `<interface>` node that is a child node of `<interfaces>` that is a child of the `<configuration>` node:

```
configuration/interfaces/interface[position() == 3]
```

Can also be written as:
`configuration/interfaces/interface[3]`
(`configuration/interfaces/interface[3]` *will be converted to* `configuration/interfaces/interface[position()==3]` *and then evaluated*)

```
<configuration>
               <interfaces>
                        <interface>
                              <name>t1-0/0/0:1:1</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>lsq-0/1/0</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>ge-1/0/0</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>ge-1/0/1</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>ge-1/0/2</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
```

```
                              <name>lo0</name>
                              . . .
                              . . .
                       </interface>
               </interfaces>
</configuration>
```

The following selects the next to last listed `<interface>` node that is a child of the `<interfaces>` node that is a child of the `<configuration>` node:

```
configuration/interfaces/interface[last()-1]

<configuration>
               <interfaces>
                        <interface>
                              <name>t1-0/0/0:1:1</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>lsq-0/1/0</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>ge-1/0/0</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>ge-1/0/1</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>ge-1/0/2</name>
                              . . .
                              . . .
                       </interface>
                       <interface>
                              <name>lo0</name>
                              . . .
                              . . .
                       </interface>
               </interfaces>
</configuration>
```

The following selects `<file>` nodes (whose `<name>` nodes have a text node that contains the text *slax*) that are child nodes of the `<commit>` node, that is a child of the `<scripts>` node, that is a child of the `<system>` node:

```
system/scripts/commit/file[contains(name, “slax”)]
```

```
<system>
        <scripts>
                <commit>
                        <file>
                                <name>commit-script-1.slax</name>
                        </file>
                        <file>
                                <name>commit-script-2.xslt</name>
                        </file>
                        <file>
                                <name>commit-script-3.slax</name>
                        </file>
                        <file>
                                <name>commit-script-4.xslt</name>
                        </file>
                </commit>
        </scripts>
</system>
```

The following selects `<interface>` nodes (whose `<name>` nodes have a text node that begins with *ge-*) that are children of the `<area>` node) whose `<name>` node has a child text node with the value *0.0.0.0*) that is a child of an `<ospf>` node that is a child of the `<protocols>` node:

```
protocols/ospf/area[name == “0.0.0.0”]/interface[starts-
with(name, “ge-”)]
```

```
<protocols>
        <ospf>
                <area>
                        <name>0.0.0.0</name>
                        <interface>
                                <name>ge-0/0/0.0</name>
                        </interface>
                        <interface>
                                <name>ge-1/1/0.0</name>
                        </interface>
                        <interface>
                                <name>lo0.0</name>
                                <passive>
                                </passive>
                        </interface>
                </area>
```

```
       </ospf>
</protocols>
```

And the following selects `<interface>` nodes (those with a configured `<mtu>` node that contains a text node with a value that is not equal to *1500* AND whose `<name>` node has a text node that begins with *ge-*) that are children of the `<interfaces>` node:

```
interfaces/interface[mtu != 1500 && starts-with(name, “ge-”)]
```

```
<interfaces>
       <interface>
              <name>ge-1/0/0</name>
              <mtu>2000</mtu>
              . . .
              . . .
       </interface>
       <interface>
              <name>ge-1/0/1</name>
              . . .
              . . .
       </interface>
       <interface>
              <name>ge-1/0/2</name>
              . . .
              . . .
       </interface>
       <interface>
              <name>xe-4/0/0</name>
              <mtu>9000</mtu>
              . . .
              . . .
       </interface>
       <interface>
              <name>lo0</name>
              . . .
              . . .
       </interface>
</interfaces>
```

Predicates to the Rescue

Predicates solve the problem presented in the hierarchy in Sample 2.18. The location path expression can use predicates so that the commit script can specifically select the lo0.0 interface. Sample 2.19 shows the modified commit script using the location path expression:

```
interfaces/interface[name == ‘lo0’]/unit[name == ‘0’]
```

This expression uses two predicates:

- one to identify the specific `<interface>` node that has a child node `<name>` that contains a text node equal to *lo0*
- one to identify the specific `<unit>` node that has a child node `<name>` that contains a text node equal to *0*

This specific location path expression selects the `<interface>` node(s) that meet its *name* == *'lo0'* criteria (*lo0*), and excludes the `<interface>` node(s) that do not meet that criteria (*ge-1/0/0*). It performs a similar function with the name of the `<unit>` node.

Sample 2.19 A Junos Commit Script that Acts on the Hierarchy in Sample 2.18

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";


import "../import/junos.xsl";

match configuration {

  if(interfaces/interface[name == ‘lo0’]/unit[name == ‘0’]) {
    <xnm:warning> {
      call jcs:edit-path();
      <message> "adding description to interface lo0 unit 0";
    }

    call jcs:emit-change($dot =  interfaces/interface[name == ‘lo0’]/unit[name ==
‘0’]) {
      with $content = {
        <description> "this description put here by commit script";
        }
      }
    }
}
```

Location path expressions

More Information On The [position()==x] Predicate

Pages 40-44 showed the `[position()==x]` predicate, which can be abbreviated to `[x]`, where `x` is an integer or a variable that contains an integer value. There are some details to be aware of when using this predicate. If the value of `x` is or represents anything other than an integer, the predicate may evaluate in an unexpected way.

Examine the op script in Sample 2.20. That op script defines three variables `$number`, `$num`, and `$txt`, representing the integer value of 3, a text representation of the number *3*, and the text value *three*, respectively. In the instance where the script uses the `[position()==x]` predicate to evaluate the `$number` variable, it returns the expected result: the third interface listed in document order (*ge-1/0/1* in this example).

When the script attempts to evaluate the `$num` or `$txt` variable, it returns a somewhat surprising result: it lists each configured interface. This is because a text value, when used as the value for `x` in `[position()==x]`, returns a TRUE value for the predicate expression. In this example, this behavior causes the XPath to select every configured interface node. In the case for the variable `$num`, the numeric value of the text can be returned via the `number()` function. This function returns the numeric value for text that represents a number. As Sample 2.20's last scenario shows, the `number()` function cannot be used on anything other than text that represents a numeric value. If there is a question as to whether the argument x is a text value for an integer or is an actual integer value, using the `number()` function in the `[position==number(x)]` or `[number(x)]` predicate is a good way to ensure that the location path expression returns the expected result.

Sample 2.20 An Op Script Testing the [position()==x] Predicate

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
          <op-script-results> {
            var $rpc-config-req = <get-configuration>;
            var $configuration = jcs:invoke($rpc-config-req);

            var $number = 3;
            var $num = "3";
            var $txt = "three";

            for-each($configuration/interfaces/interface[$number]) {
              <output> "interface number " _ $number _ " is " _ name _ "!!";
            }

            <output> "*************************************";
            for-each($configuration/interfaces/interface[$num]) {
              <output> "interface number " _ $num _ " is " _ name _ "!!";
            }

            <output> "*************************************";

            for-each($configuration/interfaces/interface[number($num)]) {
              <output> "interface number " _ $num _ " is " _ name _ "!!";
            }

            <output> "*************************************";

            for-each($configuration/interfaces/interface[$txt]) {
              <output> "interface number " _ $txt _ " is " _ name _ "!!";
            }

            <output> "*************************************";

            for-each($configuration/interfaces/interface[number($txt)]) {
              <output> "interface number " _ $txt _ " is " _ name _ "!!";
            }

            <output> "FYI - number($txt) = " _ number($txt);

        }
}
```

And the output for this would be:

```
ps@r1> op proximity-position-test
interface number 3 is ge-1/0/1!!
*************************************
interface number 3 is ge-0/0/0!!
interface number 3 is ge-0/0/1!!
interface number 3 is ge-1/0/1!!
interface number 3 is ge-1/0/2!!
interface number 3 is lo0!!
*************************************
interface number 3 is ge-1/0/1!!
*************************************
interface number three is ge-0/0/0!!
interface number three is ge-0/0/1!!
interface number three is ge-1/0/1!!
interface number three is ge-1/0/2!!
interface number three is lo0!!
*************************************
FYI - number($txt) = NaN

ps@r1>
```

Backing Up for a Moment . .

Thus far this chapter examined current and context nodes, how to move *down* an XML hierarchy with location path expressions, and how to qualify our location path expressions with predicates. But what will happen, for instance, if the script is at a current node deep in the XML hierarchy and needs to get to a higher level? So far this chapter has covered only how to move *deeper* into the hierarchy.

Introducing the .. operator (two dots or periods). When used in a location path expression, this operator means *select the parent node of the context node*. An example of how it works follows.

Take a look at Sample 2.21. This is a commit script that gives a warning message for each interface that has an address with a network mask less than 24 (0-23).

Sample 2.21 A Junos Commit Script that Uses the .. Operator

1. Makes current/context node commit-script-input/configuration/interfaces/interface/unit/family/inet/address

2. Moves context node up four levels to interface so the <name> node resolves from the context of <interface>

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";


match configuration {
    for-each (interfaces/interface/unit/family/inet/address) {
        var $prefix = substring-after(name, "/");
        var $address = substring-before(name, "/");
        var $if-name = ../../../../name;
        var $unit-name = ../../../name;
        if ($prefix < 24) {
             <xnm:warning> {
             <message> {
             expr "The address of " _ $address _ " has a prefix of
\/";
             expr $prefix _ "\non interface " _ $if-name;
           expr " unit " _ $unit-name _ "\n";
     }
           }
       }
   }
}
```

And the output from the commit script in Sample 2.21 would be:

```
ps@r1# commit
warning: The address of 30.114.8.72 has a prefix of /22
on interface ge-1/0/0 unit 100

[edit]
ps@r1#
```

Examine the `$if-name` variable in Sample 2.21 and how it is defined. Within the `for-each` loop, the current node is `/commit-script-input/configuration/interfaces/interface/unit/family/inet/address`. The `$if-name` variable must be defined as being the contents of the text node within the `<name>` node that is the child of the `<interface>` node. In order to do this, the script must use a location path expression that can move up in the hierarchy to make `<interface>` the context node so `<name>` can properly resolve from the context node of `<interface>`. Dissecting the location path expression that defines `$if-name`, shows this:

..(context node is now `<inet>`)**/..**(context node is now `<family>`)**/..**(context node is now `<unit>`)**/..**(context node is now `<interface>`)/`name`.

To better illustrate this operation, Sample 2.22 shows a sample Junos XML hierarchy that the commit script in Sample 2.21 can act on.

Sample 2.22 A Sample Junos Hierarchy to Use with Script in Sample 2.21

The location path expression `../../../../name` moves the context node up four levels to `<interface>` so that `<name>` can properly resolve

Current node and initial context node within the `for-each` loop

```
<configuration>
  <interfaces>
    <interface>
      <name>ge-1/0/0</name>
      <vlan-tagging/>
      <unit>
        <name>100</name>
        <vlan-id>100</vlan-id>
        <family>
          <inet>
            <address>
              <name>30.114.8.72/22</name>
            </address>
          </inet>
        </family>
      </unit>
    </interface>
  </interfaces>
</configuration>
```

Try It Yourself: Writing a Location Path Expression Using the .. Operator and Predicates

Write an op script that looks for all *ge-* interfaces that contain a unit with vlan-id of 200 and returns the interface name(s) and logical unit number(s) back to the user.

```
The following ge interfaces contain vlan 200:
interface ge-0/1/0 unit 3
interface ge-1/0/1 unit 200
```

Here's another example of use of the `..` operator. The script in Sample 2.23 is a commit script that will enforce a maximum number of vlans permitted on any given interface configured for `vlan-tagging`. In Sample 2.23, examine the variable `$interface`. The `for-each(interfaces/interface/vlan-tagging)` statement sets the *current* and *context node* to `<vlan-tagging>`. Breaking down the location path expression that defines `$interface` shows this: **..***(context node is now* `<interface>`*)*/name.

Sample 2.23 Another Example Using the .. Operator

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";
match configuration {

     var $limit = 3;
     for-each (interfaces/interface/vlan-tagging) {
             var $vlancount = count(../unit);
             var $interface = (../name);




             if ($vlancount > $limit) {
             <xnm:warning> {
                <message> {
                    expr "The maximum number of vlans allowed on an interface is ";
                    expr $limit;
                    expr ".  You have configured ";
                    expr $vlancount;
                    expr " vlans on ";
                    expr $interface;
```

1. The `for-each` loop sets the current and context nodes to `/commit-script-input/configuration/interfaces/interface/vlan-tagging`

2. The location path expression `../name` moves the context node up one level to `<interface>`, then down one level to the `<name>` node

```
                }
            }
          }
      }

      var $all-vlans = count(interfaces/interface/vlan-tagging/../unit);
      var $num-ge = count(interfaces/interface[starts-with(name, 'ge')]);

      <xnm:warning> {
              <message> {
                    expr "\nFYI - The total # of vlans configured on all the interfaces is 
";
                    expr $all-vlans;
                    expr ".\n";
                  expr "The total number of 'ge' interfaces is " _ $num-ge _ "!";

              }
      }
}
```

And the output from the commit script in Sample 2.23 would be:

```
ps@r1# commit
warning: The maximum number of vlans allowed on an interface is 3.  You have configured 4 
vlans on ge-1/0/0
warning: FYI - The total # of vlans configured on all the interfaces is 6.
The total number of 'ge' interfaces is 4!
```

Try It Yourself: Variable Definitions Using Location Path Expressions

Examine the script in Sample 2.23.

What is the difference between the variables `$vlancount` and `$all-vlans`?

Why do those two variables need to have different location path expressions to define them?

Summary

Chapter 2 explored the tools to move up or down in the XML hierarchy (via the `..` and `/` operators, respectively), qualifying a location path expressions (via predicates) in order to select only the nodes of interest, XML data structures in scripts, how to understand a script's reference point in the XML hierarchy (current node), and, the point at which the script is acting on in the XML hierarchy (context node).

Chapter 3 examines how to further refine node selection using the 13 *axes*.

Chapter 3

Navigating Using Axes

What Is an XML Axis?

An object in the three-dimensional world can potentially move up/down, forward/backward, and side to side from a given position. When an object does this, it is moving along one or more axis (commonly called the *x*, *y*, and *z* axes in the three dimensional world). In the XML world, there are thirteen different *directions* in which to go from your current position (the context node). To put it another way, there are thirteen different groups of nodes possible to select from any given context node. For example, from a context node, the *descendant axis* points from a node down to its child nodes, its children's child nodes, its children's children's child nodes, etc. In Junos automation, to select all the element nodes in the descendant axis, use the term:

```
descendant::*
```

Note that this expression uses the wildcard operator * to select all nodes of the principal type. The wildcard * node test will be covered in depth later in the chapter. At this point, understand that in the expression above the * operator will return all nodes of the *element* type (no attribute or text nodes) on the *descendant* axis.

MORE? Refer back to Chapter 1 for a review of node types, including *element* nodes.

The expression `descendant::*` is written in the generic format:

```
first-thing::second-thing
```

Which can be read as:

all the 'second-thing' that are a(n) 'first-thing' of the context node

So, `descendant::*` means all element nodes that are a descendant of the context node.

Examine the simple op script in Sample 3.1 that outputs the names of interface `ge-2/0/1`'s descendant element nodes:

Sample 3.1 Simple Op Script Using Descendant Node

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";
```

```
match / {
         <op-script-results> {
           var $rpc-config-req = <get-configuration>;
           var $configuration = jcs:invoke($rpc-config-req);

           for-each($configuration/interfaces/interface[name == "ge-1/0/2"]) {
             for-each(descendant::*) {
             <output> "descendant node is " _ name() _ "!";
           }
         }
}
The output from the commit script in Sample 3.1 would be:
descendant node is name!
descendant node is vlan-tagging!
descendant node is unit!
descendant node is name!
descendant node is vlan-id!
descendant node is family!
descendant node is inet!
descendant node is address!
descendant node is name!
```

And the Junos XML hierarchy used in the op script in Sample 3.1:

```
ps@r1> show configuration interfaces ge-1/0/2 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.4R3/junos">
   <configuration junos:commit-seconds="1259324066" junos:commit-localtime="2009-11-27
12:14:26 UTC" junos:commit-user="ps">
         <interfaces>
            <interface>
               <name>ge-1/0/2</name>
               <vlan-tagging/>
               <unit>
                  <name>375</name>
                  <vlan-id>375</vlan-id>
                  <family>
                     <inet>
                        <address>
                           <name>10.0.10.2/30</name>
                        </address>
                     </inet>
                  </family>
               </unit>
            </interface>
         </interfaces>
   </configuration>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>

ps@r1>
```

NOTE The script in Sample 3.1 outputs the name of the selected nodes (using the `name()` operator) rather than the contents of the selected nodes. Using the `name()` operator to display the name of the selected node rather than displaying the entire contents of the selected node makes it easier to see which nodes the script is selecting. This technique can be a valuable troubleshooting tool if there is a question as to which nodes are being selected.

The *descendant* axis is just one of the thirteen axes listed in Table 3.1. The table shows the axis type (*forward* and *reverse* axes) discussed in the next section, lists the contents of each axis (which nodes each axis selects), and gives the generic syntax to use that axis in a script.

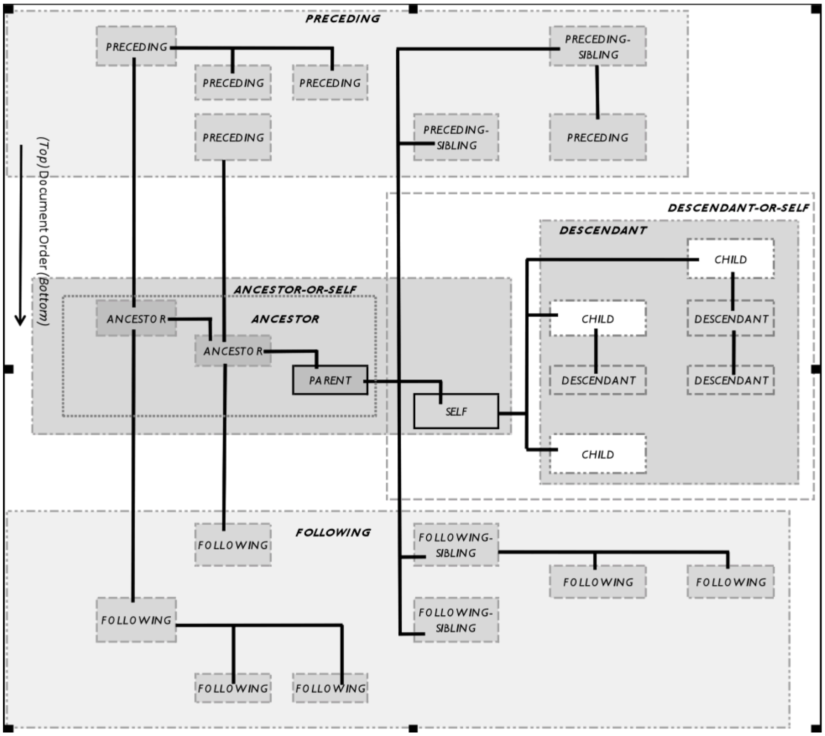
Table 3.1 The Thirteen XML Axes

Axis	Axis Type	Axis Contents	Generic Syntax
Child (default)	Forward	Children of the context node	child::
Parent	Forward	Parent of context node	parent::
Self	Forward	Context node	self::
Attribute	Forward	Attributes of context node; will typically be empty unless the context node is an element type node	attribute::
Ancestor	Reverse	Ancestors of context node	ancestor::
Ancestor-or-self	Reverse	Context node and ancestors of context node	ancestor-or-self::
Descendant	Forward	Descendants (child nodes, child's child nodes, etc.)	descendant::
Descendant-or-self	Forward	Context node and descendants of context node	descendant-or-self::
Preceding-sibling	Reverse	Preceding siblings of context node in document order	preceding-sibling::
Following-sibling	Forward	Following siblings of context node in document order	following-sibling::
Preceding	Reverse	All nodes that are before the context node in document order, excluding any ancestors, attribute nodes, and namespace nodes	preceding::
Following	Forward	All nodes that are after the context node in document order, excluding any descendants, attribute nodes, and namespace nodes	following::
Namespace	Forward	Returns defined namespaces for the context node if the context node is an element node	namespace::

NOTE In general, the uses for the *namespace* axis in SLAX for Junos automation are very minimal, and are especially so for a new to moderately experienced script writer. That being the case, any further explanation of the uses of the namespace axis is beyond the scope of this booklet.

Figure 3.1 illustrates the relationships of nodes to their respective axes. Start at the SELF node when reading it. Notice that a single node can be on more than one axis. For example, the PARENT node is on the *parent*, *ancestor*, and *ancestor-or-self* axes. Another example is the PRECEDING-SIBLING nodes, which lie on the *preceding* and *preceding-sibling* axes.

Figure 3.1 **Illustration of Relationship of the XML Axes (*Namespace* and *Attribute* Axes Not Shown)**



Forward and Reverse Axis Types

Examine the output for Sample 3.1 (on pages 52-53). The listed nodes from the script output are in the same order as the nodes themselves appear in the Junos XML hierarchy for Sample 3.1. This is the *document order*, or the order in which the nodes appear starting from the top of the XML document. An axis that can only contain the context node or nodes that are after the context node in document order is called a *forward axis*. Likewise, an axis that can only contain the context node or nodes that are before the context node in document order is called a *reverse axis*. Since the nodes selected by the *descendant* axis appear after the context node in document order (see the output and Junos XML hierarchy for Sample 3.1), the *descendant* axis is a *forward axis*.

Proximity position of a node within a node-set is the numbered position of the node in *document order* for a *forward* axis or in *reverse document order* for a *reverse* axis. The previous statement could use some clarification via an example. Examine the op script in Sample 3.2 and that op script's output following Sample 3.2. The script does the following:

- Sets the *context* node to `/$configuration/interfaces/interface[name == 'lo0']/unit[name == '0']`.
- Displays the element ancestor nodes for the context node in document order.
- Displays the element ancestor nodes in proximity positions 1 and 2 from the context node.
- Displays the element descendant nodes for the context node in document order.
- Displays the element descendant nodes in proximity positions 1 and 2 from the context node.

Sample 3.2 Op Script Showing *Forward* and *Reverse* Axis Examples

```
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
   <op-script-results> {
           var $rpc-config-req = <get-configuration groups="groups" inherit="inherit" 
changed="changed">;
         var $configuration = jcs:invoke($rpc-config-req);

           var $path = $configuration/interfaces/interface[name == ‘lo0’]/unit[name == 
‘0’];

           for-each($path) {
             for-each(ancestor::*) {
               <output> "ancestor is " _ name();
             }

             <output> "ancestor::*[1] = " _ name(ancestor::*[1]);
             <output> "ancestor::*[2] = " _ name(ancestor::*[2]);

             <output> "****************************************";

             for-each(descendant::*) {
               <output> "descendant is " _ name();
             }

             <output> "descendant::*[1] = " _ name(descendant::*[1]);
             <output> "descendant::*[2] = " _ name(descendant::*[2]);
           }
         }
}
```

NOTE Recall from Chapter 2 that `[x]` is an abbreviation for `[position() == x]`.

And the output from the commit script in Sample 3.2 is:

```
ps@r1> op proximity-position
ancestor is configuration
ancestor is interfaces
ancestor is interface
ancestor::*[1] = interface
ancestor::*[2] = interfaces
****************************************
descendant is name
descendant is family
descendant is inet
descendant is address
descendant is name
descendant::*[1] = name
descendant::*[2] = family

ps@r1>
```

And the sample hierarchy used in the op script in Sample 3.2 to act on:

```
<configuration>
  <interfaces>
    <interface>
      <name>lo0</name>
      <unit>
        <name>0</name>
        <family>
          <inet>
            <address>
              <name>10.200.7.1/32</name>
            </address>
          </inet>
        </family>
      </unit>
    </interface>
  </interfaces>
</configuration>
```

Notice that the nodes on the ancestor axis in proximity positions [1] (`ancestor::*[1]`) and [2] (`ancestor::*[2]`) are `<interface>` and `<interfaces>` respectively. They list out in reverse document order. So as the proximity positions increase in order, the nodes filling those positions list out in reverse document order (see the preceding sample hierarchy). This is because the *ancestor* axis is a *reverse axis*. The opposite is true for the descendant axis. As the proximity positions increase in order, the nodes filling those positions list out in document order. This is because the *descendant* axis is a *forward axis*.

NOTE A `for-each()` loop will always list out nodes in document order regardless of the type of axis (forward or reverse) it is acting on.

It is important to understand whether an axis is a forward or reverse axis if you are going to work with proximity positions. Failure to understand can result in the selection of the wrong node(s).

Axes and Location Path Expressions

Location path expressions can be used with axis identifiers to further select the desired nodes in the hierarchy. Look at the script in Sample 3.1 on page **52-53** again. If the requirement is to select only descendant nodes of interface `ge-1/0/2` named `vlan-id`, replace the expression:

```
for-each(descendant::*)
```

With:

```
for-each(descendant::vlan-id)
```

In plain English, the latter term means select all the descendant nodes of the context node (interface ge-1/0/2) node named `vlan-id`.

Or, if the requirement is to select descendant nodes of interface *ge-1/0/2* that have IPv4 `<address>` nodes, the expression is:

```
for-each(descendant::family/inet/address)
```

Node Tests

Element is the principal node type or all axes except for the *attribute* and *namespace* axes. The principal node type for the *attribute* axis is an *attribute* node, and the principal node type for the *namespace* axis is a *namespace* node. Table 3.2 highlights the principal node types.

Table 3.2 **Principal Node Types for Different Axes**

Axis	Principal Node Type
Attribute	attribute
Namespace	namespace
Any axis other than attribute or namespace	element

If, for example, you are interested in selecting the ancestor nodes for a given context node, there are a couple of options available. The syntax `ancestor::node()` will select an ancestor node of *any* type. In contrast, the syntax using the wildcard operator, `ancestor::*`, will select all ancestor nodes of the *principal* type for the given axis. In the case of the ancestor node, per Table 3.2, the principal node type is *element* (refer back to Chapter 1 for a review of the various node types). This means that `ancestor::*` selects the ancestors of the context node that are *element* node types.

This difference between the `node()` test and the wildcard * test may be best illustrated with a script example. Examine the op script in Sample 3.3. This script displays the names of the ancestor nodes of the *lo0* interface using both the `node()` and wildcard * tests. Sample output from the script follows it.

Sample 3.3 Ancestor Node Test Script: node() versus *

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
   <op-script-output> {
         var $rpc-config-req = <get-configuration groups="groups" inherit="inherit"
changed="changed">;
         var $configuration = jcs:invoke($rpc-config-req);

           for-each($configuration/interfaces/interface[name == "lo0"]) {
             <output> "***ancestor results using node() test***";
             for-each(ancestor::node()) {
               <output> "ancestor node(node() test)is " _ name();
             }

             <output> "****ancestor results using * test****";
             for-each(ancestor::*) {
               <output> "ancestor node (* test) is " _ name();
             }
           }
         }
}
```

And the output from the commit script in Sample 3.3:

```
ps@r1> op ancestor-node-test-name-vs-star
***ancestor results using node() test***
ancestor node(node() test)is
ancestor node(node() test)is configuration
ancestor node(node() test)is interfaces
****ancestor results using * test****
ancestor node (* test) is configuration
ancestor node (* test) is interfaces
```

Examine the script in Sample 3.3. The first section of the script displays the names of the ancestor nodes of the *lo0* interface using the `node()` test. The second section also returns the ancestor nodes of the *lo0* interface, but uses the wildcard * test. The results of the two methods are slightly different. The results using the `node()` test show three ancestors (the `root` node name shows up as blank) while the results of the wildcard * test show two. The `root` node does not show up in the wildcard * test results because, per Table 3.2, the principal node type for the ancestor axis is an *element* node. This means that the *ancestor::** test will return only ancestors of the *element* type (the root node is a node type of *root*). In contrast, the *ancestor::node()* test will return nodes of any type (*element* or *root*, in this example).

For reference, below is a snippet of the partial XML configuration hierarchy for *interface lo0*:

```
ps@r1> show configuration interfaces lo0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.3I0/junos">
   <configuration junos:commit-seconds="1264031058" junos:commit-
localtime="2010-01-20 23:44:18 UTC" junos:commit-user="tim">
         <interfaces>
            <interface>
               <name>lo0</name>
```

NOTE Running this script in Junos versions prior to 9.3S7, 9.4R4, 9.5R4, 9.6R3, 10.0R2, and 10.1R1 will display `<rpc-reply>` and `<junoscript>` as additional ancestor nodes and may display the *root* node last in the list. The presence of the `<rpc-reply>` and `<junoscript>` nodes is discussed in Chapter 2.

NOTE As explained in Chapter 2, the `<rpc-reply>` node, while technically part of the XML hierarchy of the returned results, should not concern you because it is not part of the configuration itself and you will mainly be working in the `<configuration>` node.

Try It Yourself: Using the Axes

Write a script that does the following:

1. Displays a count of all element nodes in the Junos device’s configuration.

2. Sets the current and context nodes to `interface[name == ‘lo0’].`

   a. Returns the count of element nodes on the ancestor axis.

   b. Returns the count of element nodes on the descendant axis.

   c. Returns the count of element nodes on the following axis.

   d. Returns the count of element nodes on the preceding axis.

   e. Returns the count of element nodes on the self axis.

3. Returns the sum of the values in step (2).

NOTE The *ancestor*, *descendant*, *following*, *preceding*, and *self* axes do not have any nodes in common and taken together they will return every element node in the configuration's XML hierarchy. The sum of the axes values listed in step (3) should match the value listed in step (1).

Axis Abbreviations

One special axis of note is the *child* axis. The child axis is the default axis, so the *child::* expression does not need to be present in a location path expression. For example, the op script in Sample 3.3 contains the expression:

```
($configuration/interfaces/interface[name == "lo0"])
```

This expression is short for:

```
($configuration/child::interfaces/child::interface[child::name
== "lo0"])
```

The `/descendant-or-self::node()/` syntax can be abbreviated with //. Sample 3.4 shows an op script with several examples of use of the *descendant-or-self* axis. The op script's output follows along with a sample XML hierarchy that the script in Sample 3.4 can act on.

Sample 3.4 Op Script Using Descendant-or-self Axis and Script Output

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {
          <op-script-output> {
            var $rpc-config-req = <get-configuration>;
            var $configuration = jcs:invoke($rpc-config-req);

            for-each($configuration/protocols/ospf) {
              <output> "initial context node is " _ name();

              <output> "*******************";

              for-each(.//self::*) {
                <output> "self node name is " _ name();
              }
```

Starting from context node, selects each node of any type on the `descendant-or-self axis` and from those nodes selects each principal node type of the `self axis`

```
              <output> "*******************";

              for-each(.//name) {
                <output> "selected name nodes are " _ name();
              }

              <output> "*******************";

              for-each(.//child::name) {
                <output> "selected name nodes v2 are " _ name();
              }

            }
          }
}
```

Starting from the context node, selects each node of any type on the descendant-or-self axis and from those nodes selects each <name> child node

And the output from the commit script in Sample 3.4:

```
ps@r1> op descendant-or-self-scriptv2
initial context node is ospf
*******************
self node name is ospf
self node name is area
self node name is name
self node name is interface
self node name is name
self node name is interface
self node name is name
self node name is passive
*******************
selected name nodes are name
selected name nodes are name
selected name nodes are name
*******************
selected name nodes v2 are name
selected name nodes v2 are name
selected name nodes v2 are name

ps@r1>
```

And the sample XML hierarchy for the script in Sample 3.4 to act on:

```
ps@r1> show configuration protocols ospf | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.3I0/junos">
   <configuration junos:commit-seconds="1265028487" junos:commit-localtime="2010-02-01
12:48:07 UTC" junos:commit-user="ps">
          <protocols>
             <ospf>
                <area>
                   <name>0.0.0.0</name>
```

```
                  <interface>
                     <name>ge-1/0/0.0</name>
                  </interface>
                  <interface>
                     <name>lo0.0</name>
                     <passive>
                     </passive>
                  </interface>
               </area>
            </ospf>
         </protocols>
   </configuration>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>

ps@r1>
```

There are other abbreviations that allow you to express some common verbose expressions more concisely. Table 3.3 shows these common expressions and their abbreviations, while Table 3.4 shows some examples of the abbreviations and explanations of each.

Table 3.3 **Syntax Abbreviations for Axes in Location Path Expressions**

Unabbreviated Syntax	Abbreviated Syntax
child::	*Can be omitted from location path expressions since* child *is the default axis*
parent::node()	.. *(two periods)*
self::node()	. *(period)*
self::node()/descendant-or-self::node()/	.//
/descendant-or-self::node()/	//
attribute::	@

Table 3.4 **Example Axis Abbreviations in XPath**

Abbreviation Example	Explanation
unit	Selects *unit* nodes that are children of the context node
unit/family	Selects the *family* nodes that are a child of the *unit* node(s) that are a child of the context node
unit[3]	Selects the 3rd *unit* child node of the context node in document order
unit[last()]	Selects the last *unit* child node of the context node in document order
ancestor::*[2]	Selects the 2nd ancestor node (in reverse document order) of the principal node type from the context node
.	Selects the context node
../unit	Selects the *unit* child node of the parent of the context node
interface[@inactive="inactive"]	Selects *interface* nodes of the context node with an attribute of *inactive* that has a value of *inactive*
interfaces/interface[@junos:group="fxp0"]	Selects *interface* nodes (that have an attribute of *junos:group* that has a value of *fxp0*) that are a child of node *interfaces*
//interface	Selects all *interface* nodes in the XML hierarchy
.//interface	Selects all *interface* nodes that are descendants of the context node

The /descendant-or-self::node()/, or its abbreviated expression //, provide the capability for a very powerful search. For example, *//unit* is short for */descendant-or-self::node()/child::unit*. This term will start at root, and look in the entire document for all nodes named unit. In complex XML hierarchies (such as a lengthy Junos device configuration), this process can be very resource intensive because it requires the script to examine every element node in the hierarchy and determine if it is named *unit*. Care should be taken to use this type of search sparingly. It is likely that you have a more specific idea of the <unit> nodes of interest and so you should attempt to more precisely specify the location. Avoid using a path such as:

```
//interface[starts-with(name, "ge-“)]/unit
```

This examines every node in the configuration to see if it is named *interface*. This search type is inefficient for two reasons:

1. It forces the script to examine every node in the XML hierarchy (the entire Junos configuration) to see if it is named `interface`.
2. It may return *interface* elements that are not of interest, such as interface elements found in the `configuration/protocols/ospf/area` hierarchy, which then forces the script to see if that `<interface>` node has a child node `<unit>`.

WARNING Excessive or unnecessary use of the `//` or `/descendant-or-self::node()/child::` syntax may cause a script to take longer to execute.

Try It Yourself: Using the Attribute and Parent Axes

Write an op script that identifies all nodes in a Junos XML configuration with an attribute of inactive and displays the name of each inactive node.

Compare the output from the script to the output from `show configuration | display xml | match inactive`

Hint: In order for the `$configuration` variable to contain the inactive nodes, define the rpc without the inherit attribute (ex: `var $rpc-config-req = <get-configuration>`)

Summary

Use XML axes to further refine the node selection in Junos automation, forward and reverse axis types, the difference between the node() and * (wildcard) node tests, how to incorporate the axes in location path expressions, common abbreviations employed when using the axes, and some common pitfalls associated with overuse of the // operator. Axes are a valuable tool that allow you to select specific nodes in a group that would otherwise be difficult to do without axes.

What to Do Next & Where to Go …

http://www.juniper.net/dayone

The PDF version of this booklet includes an additional Appendix.

http://www.juniper.net/automation

The Junos Automation home page, where plenty of useful resources are available including training class, recommended reading, and a script library - an online repository of scripts that can be used on Junos devices.

http://forums.juniper.net/jnet

The Juniper-sponsored J-Net Communities forum is dedicated to sharing information, best practices, and questions about Juniper products, technologies, and solutions. Register to participate at this free forum.

http://www.juniper.net/techpubs/en_US/junos/information-products/topic-collections/config-guide-automation/frameset.html

All Juniper-developed product documentation is freely accessible at this site, including the Junos API and Scripting Documentation.

http://www.juniper.net/us/en/products-services/technical-services/j-care/

Building on the Junos automation toolset, Juniper Networks Advanced Insight Solutions (AIS) introduces intelligent self-analysis capabilities directly into platforms run by Junos. AIS provides a comprehensive set of tools and technologies designed to enable Juniper Networks Technical Services with the automated delivery of tailored, proactive network intelligence and support services.

Appendix

Table of Location Path Operators

Table A-1 is a complete list of the location path operator types found in this booklet, along with examples and explanations for each.

Table A.1 **Location Path Operators**

Name `Code`	`Example` **Explanation**
Root /	`/` Selects the *root* node
Location Path Step /	`var $ospf = $configuration/protocols/ospf` Each / represents a step deeper into the XML hierarchy along an axis. The child axis is the default axis; if an axis is not specified then the step is in the direction of the child axis. The example above defines the `$ospf` variable as the node named `ospf` that is a child of the `<protocols>` node that is the child node of the `$configuration` variable.
Multiple Steps //	`//protocols` Abbreviation for `/descendant-or-self::node()/` The syntax in the example above starts at root and checks all nodes in the hierarchy, returning all nodes named `protocols`, no matter where they are in the XML hierarchy. This search can be resource intensive for larger Junos configurations and should be used sparingly.

Parent Axis
..
	`for-each(interfaces/interface/unit/family/inet/address) {`
	`    var $int-name = ../../../../name;`
	`}`
	
	Represents the parent axis. When used in conjunction with the location path step /, moves the context node up one step on the parent axis, effectively making the old parent node the new context node.
	
	In the example above, `$int-name` is defined by stepping the parent axis 4 times so that `name` is resolved from the context of `interface`.

Self Axis
`.`
	`.//interface`
	
	`for-each($configuration/interfaces/interface/name[contains(., "-1/")]) {`
	`        <output> "slot 1 has interface " _ . _ " configured";`
	`}`
	
	Represents the context node in a location path expression.
	
	The syntax in the first example searches for all nodes named `interface` starting from the context node.
	
	The syntax in the second example checks the contents of the context node in a predicate and then displays the contents of the context node when the `if()` statement evaluates to TRUE.

Predicates `[]`	`for-each($configuration/interfaces/interface[starts-with(name, "t3-“)])` Predicates qualify the node selection. They always appear between square brackets []. The predicate returns a TRUE or FALSE result. If the predicate expression evaluates to TRUE then the node is selected. The example qualifies the `<interface>` nodes selected to those whose `<name>` node's contents begin with `t3-`
Attribute Axis `@`	`for-each($interfaces/interface[@="inactive"])` The attribute axis contains the attributes of the context node. The example above searches for interface nodes with an attribute of *inactive*.
Wildcard `*`	`for-each(ancestor::*)` The wildcard operator * returns all nodes of the principal type for the context node. The example above will return the `<configuration>` and `<interfaces>` element nodes if the context node is `<interface>` (an element node).
Node Test `node()`	`for-each(ancestor::node())` The `node()` test returns all nodes of any type for the context node. The example above will return the / *(root)*, `<configuration>`, and `<interfaces>` nodes if the context node is `<interface>`.

Child axis *(default axis)*	`interfaces/interface` The child axis is the default axis. The example above finds child nodes of `<interfaces>` that are named `interface`.
Ancestor axis `ancestor::`	`ancestor::interface` Returns ancestor nodes of context node. The example selects all `<interface>` element nodes that are an ancestor of the context axis.
Ancestor-or-self axis `ancestor-or-self::`	`ancestor-or-self::family` Returns context node and ancestors of context node. The example selects all `<family>` nodes that are the context node or ancestors of the context node.
Descendant axis `descendant::`	`descendant::family` Returns all descendants of context node. The example selects all `<family>` nodes that are descendants of the context node.
Descendant-or-self `descendant-or-self::`	`descendant-or-self::interface` Returns the context node and all descendants. The example returns all `<interface>` nodes that are the context node or descendants of the context node.
Preceding-sibling axis `preceding-sibling::`	`../preceding-sibling::name` Returns all sibling nodes that precede the context node in document order. The example selects `<name>` nodes that are preceding-siblings of the parent of the context node.
Following-sibling axis `following-sibling::`	`following-sibling::node()` Selects sibling nodes that are after the context node in document order. The example selects any following-sibling nodes of the context node.

Preceding axis preceding::	preceding::node() Returns nodes that precede the context node in document order excluding ancestors, attribute nodes, and namespace nodes. The example selects all preceding nodes, regardless of node type, of the context node.
Following axis following::	following::interface Returns nodes that are after the context node in document order excluding descendants, attribute nodes, and namespace nodes. The example selects <interface> nodes that follow the child of the context node in document order as long as the node is not a descendant of the context node, an attribute node, or a namespace node.

Answers to Try It Yourself Exercises

Chapter 1

Try It Yourself: Viewing the XML Hierarchy

Run the command show configuration interfaces lo0:

```
ps@r1> show configuration interfaces lo0
unit 0 {
     family inet {
        address 10.200.7.1/32;
    }
}
```

Run the command show configuration interfaces lo0 | display xml:

```
ps@r1> show configuration interfaces lo0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.3I0/
junos">
    <configuration junos:commit-seconds="1265669494"
junos:commit-localtime="2010-02-08 22:51:34 UTC" junos:commit-
user="ps">
            <interfaces>
            <interface>
               <name>lo0</name>
               <unit>
                  <name>0</name>
                  <family>
                     <inet>
                        <address>
                           <name>10.200.7.1/32</name>
                        </address>
                     </inet>
                  </family>
               </unit>
            </interface>
         </interfaces>
   </configuration>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>

ps@r1>
```

Compare the output of the two commands.

Do the two versions of output contain the same information?
Both versions contain the same information. The difference is how the information is organized.

Which version of output is easier to read?
The output from show configuration interfaces lo0 is easier to read.

Which version of output is easier to use to describe where to find the interface's IP address?
The output from show configuration interfaces lo0 | display xml is easier to use to describe how to get to the IP address. The way to the IP address information can be described by navigating through the XML hierarchy using the XML tags.

Try It Yourself: An Operational Command's XML Output

Run the command `show interface lo0`:

```
ps@r1> show interfaces lo0
Physical interface: lo0, Enabled, Physical link is Up
  Interface index: 6, SNMP ifIndex: 6
  Type: Loopback, MTU: Unlimited
  Device flags   : Present Running Loopback
  Interface flags: SNMP-Traps
  Link flags     : None
  Last flapped   : Never
    Input packets : 0
    Output packets: 0

  Logical interface lo0.0 (Index 72) (SNMP ifIndex 16)
    Flags: SNMP-Traps Encapsulation: Unspecified
    Input packets : 0
    Output packets: 0
    Protocol inet, MTU: Unlimited
      Flags: None
      Addresses, Flags: Is-Default Is-Primary
        Local: 10.200.7.1

  Logical interface lo0.16384 (Index 69) (SNMP ifIndex 21)
    Flags: SNMP-Traps Encapsulation: Unspecified
    Input packets : 0
    Output packets: 0
    Protocol inet, MTU: Unlimited
      Flags: None
      Addresses
        Local: 127.0.0.1

  Logical interface lo0.16385 (Index 70) (SNMP ifIndex 22)
    Flags: SNMP-Traps Encapsulation: Unspecified
    Input packets : 0
    Output packets: 0
    Protocol inet, MTU: Unlimited
      Flags: None

  Logical interface lo0.32768 (Index 71) (SNMP ifIndex 123)
    Flags: Encapsulation: Unspecified
    Input packets : 0
    Output packets: 0

ps@r1>
```

Run the command `show interface lo0 | display xml`:

```
ps@r1> show interfaces lo0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.3I0/
junos">
    <interface-information xmlns="http://xml.juniper.net/
junos/9.3I0/junos-interface" junos:style="normal">
        <physical-interface>
            <name>lo0</name>
            <admin-status junos:format="Enabled">up</admin-
status>
            <oper-status>up</oper-status>
```

```
          <local-index>6</local-index>
          <snmp-index>6</snmp-index>
          <if-type>Loopback</if-type>
          <mtu>Unlimited</mtu>
          <if-device-flags>
             <ifdf-present/>
             <ifdf-running/>
             <ifdf-loopback/>
          </if-device-flags>
          <if-config-flags>
             <iff-snmp-traps/>
          </if-config-flags>
          <if-media-flags>
             <ifmf-none/>
          </if-media-flags>
          <interface-flapped junos:seconds="0">Never</interface-
flapped>
          <traffic-statistics junos:style="brief">
             <input-packets>0</input-packets>
             <output-packets>0</output-packets>
          </traffic-statistics>
          <logical-interface>
             <name>lo0.0</name>
             <local-index>72</local-index>
             <snmp-index>16</snmp-index>
             <if-config-flags>
                <iff-snmp-traps/>
             </if-config-flags>
             <encapsulation>Unspecified</encapsulation>
             <traffic-statistics junos:style="brief">
                <input-packets>0</input-packets>
                <output-packets>0</output-packets>
             </traffic-statistics>
             <address-family>
                <address-family-name>inet</address-family-name>
                <mtu>Unlimited</mtu>
                <address-family-flags>
                   <ifff-none/>
                </address-family-flags>
                <interface-address>
                   <ifa-flags>
                      <ifaf-current-default/>
                      <ifaf-current-primary/>
                   </ifa-flags>
                   <ifa-local>10.200.7.1</ifa-local>
                </interface-address>
             </address-family>
          </logical-interface>
          <logical-interface>
             <name>lo0.16384</name>
```

```
   <local-index>69</local-index>
   <snmp-index>21</snmp-index>
   <if-config-flags>
      <iff-snmp-traps/>
   </if-config-flags>
   <encapsulation>Unspecified</encapsulation>
   <traffic-statistics junos:style="brief">
      <input-packets>0</input-packets>
      <output-packets>0</output-packets>
   </traffic-statistics>
   <address-family>
      <address-family-name>inet</address-family-name>
      <mtu>Unlimited</mtu>
      <address-family-flags>
         <ifff-none/>
      </address-family-flags>
      <interface-address heading="Addresses">
         <ifa-local>127.0.0.1</ifa-local>
      </interface-address>
   </address-family>
</logical-interface>
<logical-interface>
   <name>lo0.16385</name>
   <local-index>70</local-index>
   <snmp-index>22</snmp-index>
   <if-config-flags>
      <iff-snmp-traps/>
   </if-config-flags>
   <encapsulation>Unspecified</encapsulation>
   <traffic-statistics junos:style="brief">
      <input-packets>0</input-packets>
      <output-packets>0</output-packets>
   </traffic-statistics>
   <address-family>
      <address-family-name>inet</address-family-name>
      <mtu>Unlimited</mtu>
      <address-family-flags>
         <ifff-none/>
      </address-family-flags>
   </address-family>
</logical-interface>
<logical-interface>
   <name>lo0.32768</name>
   <local-index>71</local-index>
   <snmp-index>123</snmp-index>
   <encapsulation>Unspecified</encapsulation>
   <traffic-statistics junos:style="brief">
      <input-packets>0</input-packets>
      <output-packets>0</output-packets>
   </traffic-statistics>
</logical-interface>
```

```
        </physical-interface>
    </interface-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>

ps@r1>
```

Compare the output of the two commands. Is the same information contained in each version of output? *The two commands contain essentially the same information but are organized differently.*

What is the location path expression to get to the *physical interface name* for the lo0 interface in the XML output? *Given the output, the path to reach the physical interface name is* `/rpc-reply/interface-information/physical-interface/name`.

Chapter 2

Try It Yourself: Viewing Commit Script Input

To view the actual input for a commit script for yourself, configure the following:

1. Configure `traceoptions` under `[system scripts commit]` to flag input. The file name can be of your choosing:

```
ps@script-compy> show configuration system scripts commit
traceoptions {
    file commit-scripts;
    flag input;
}
```

2. Clear the configured log file (this clears out any extraneous log entries that may be in the file so it is easier to view the script input).

3. Configure a commit script to run.

4. Perform a commit.

5. Run the following CLI command:

```
ps@r1> show log commit-scripts
Nov 19 22:49:11 script-compy clear-log[14466]: logfile cleared
Nov 19 22:49:13 cscript script processing begins
Nov 19 22:49:14 reading commit script configuration
Nov 19 22:49:14 testing commit script configuration
Nov 19 22:49:14 commit script input
Nov 19 22:49:14 begin dump
<?xml version="1.0"?>
<commit-script-input xmlns:junos="http://xml.juniper.net/
junos/*/junos">

<configuration junos:changed-seconds="1258666164"
junos:changed-localtime="2009-11-19 21:29:24 UTC">
<version>9.4R3.5</version>
<system>
. . .
   . . .
```

Try It Yourself: Writing a Location Path Expression Using the .. Operator and Predicates

Write an op script that looks for all *ge-* interfaces that contain a unit with vlan-id of 200 and returns the interface name(s) and logical unit number(s) back to the user.

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {

   <op-script-results> {

    var $rpc-config-req = <get-configuration>;
    var $configuration = jcs:invoke($rpc-config-req);

    <output> "The following ge interfaces contain vlan 200:";

    for-each($configuration/interfaces/
interface[contains(name, ‘ge-’)]/vlan-tagging/../unit[vlan-id
== ‘200’]) {
        <output> "interface " _ ../name _ " unit " _ name;
    }
  }
}
```

```
The following ge interfaces contain vlan 200:
interface ge-0/1/0 unit 3
interface ge-1/0/1 unit 200
```

Try It Yourself: Variable Definitions Using Location Path Expressions

Examine the script in Sample 2.17.

What is the difference between the variables `$vlancount` and `$all-vlans`?

> *`$vlancount` is defined from the context node `<vlan-tagging>`, and represents the number of units configured on the parent node of `<vlan-tagging>`. `$all-vlans` is defined from the context node `<configuration>` and represents the number of units configured on all the interfaces with vlan-tagging enabled on the entire Junos device.*

Why do those two variables need to have different location path expressions to define them?

> *The variables $vlancount and $all-vlans must be defined with different location path expressions because each serves a different purpose: one counts all the configured units under each interface configured for vlan-tagging, while the other counts all configured units under any interface with vlan-tagging enabled.*

Chapter 3

Try It Yourself: Using the Axes

Write a script that does the following:

1. Displays a count of all element nodes in the Junos device's configuration.

2. Sets the current and context nodes to `interface[name == ‘lo0’]`.

   a. Returns the count of element nodes on the ancestor axis.

   b. Returns the count of element nodes on the descendant axis.

   c. Returns the count of element nodes on the following axis.

   d. Returns the count of element nodes on the preceding axis.

   e. Returns the count of element nodes on the self axis.

3. Returns the sum of the values in step (2).

Note: The ancestor, descendant, following, preceding, and self axes do not have any nodes in common, and taken together they will return every element node in the configuration's XML hierarchy. The sum of the axes values listed in step (3) should match the value listed in step (1).

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";


import "../import/junos.xsl";

match / {

   <op-script-results> {
               var $rpc-config-req = <get-configuration>;
        var $configuration = jcs:invoke($rpc-config-req);

               for-each($configuration/interfaces/interface[name == ‘lo0’]) {
                       var $self-nodes = count(self::*);
                       var $preceding-nodes = count(preceding::*);
                       var $following-nodes = count(following::*);
                       var $desc-nodes = count(descendant::*);
                       var $ancestor-nodes = count(ancestor::*);
                       var $all-nodes = $self-nodes + $preceding-nodes + $following-
nodes + $desc-nodes + $ancestor-nodes;

                       <output> "the count for all nodes is " _ $all-nodes;
               }
               <output> "****";
               <output> "there are " _ count($configuration/descendant-or-self::*) _ "
element nodes";

        }
}

ps@r1> op axes-test-scriptv4
the count for all nodes is 927
****
there are 927 element nodes

ps@r1>
```

Try It Yourself: Using the attribute and Parent Axes

Write an op script that identifies all nodes in a Junos XML configuration with an attribute of inactive and displays the name of each inactive node.

Compare the output from the script to the output from `show configuration | display xml | match inactive`

Hint: In order for the `$configuration` variable to contain the inactive nodes, define the rpc without the inherit attribute (ex: `var $rpc-config-req = <get-configuration>`).

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";

import "../import/junos.xsl";

match / {

   <op-script-results> {

               var $rpc-config-req = <get-configuration>;
               var $configuration = jcs:invoke($rpc-config-req);

               if($configuration//@inactive) {
                       for-each($configuration//@inactive) {
                              <output> "the node " _ name(..) _ " is inactive";
                       }
               } else {
                       <output> "no inactive nodes";
               }
        }
}
```

```
ps@r1> op inactive-nodes
the node authentication-order is inactive
the node file is inactive
the node file is inactive
the node file is inactive
the node trap-group is inactive
the node bgp is inactive
the node routing-instances is inactive

ps@r1>
```