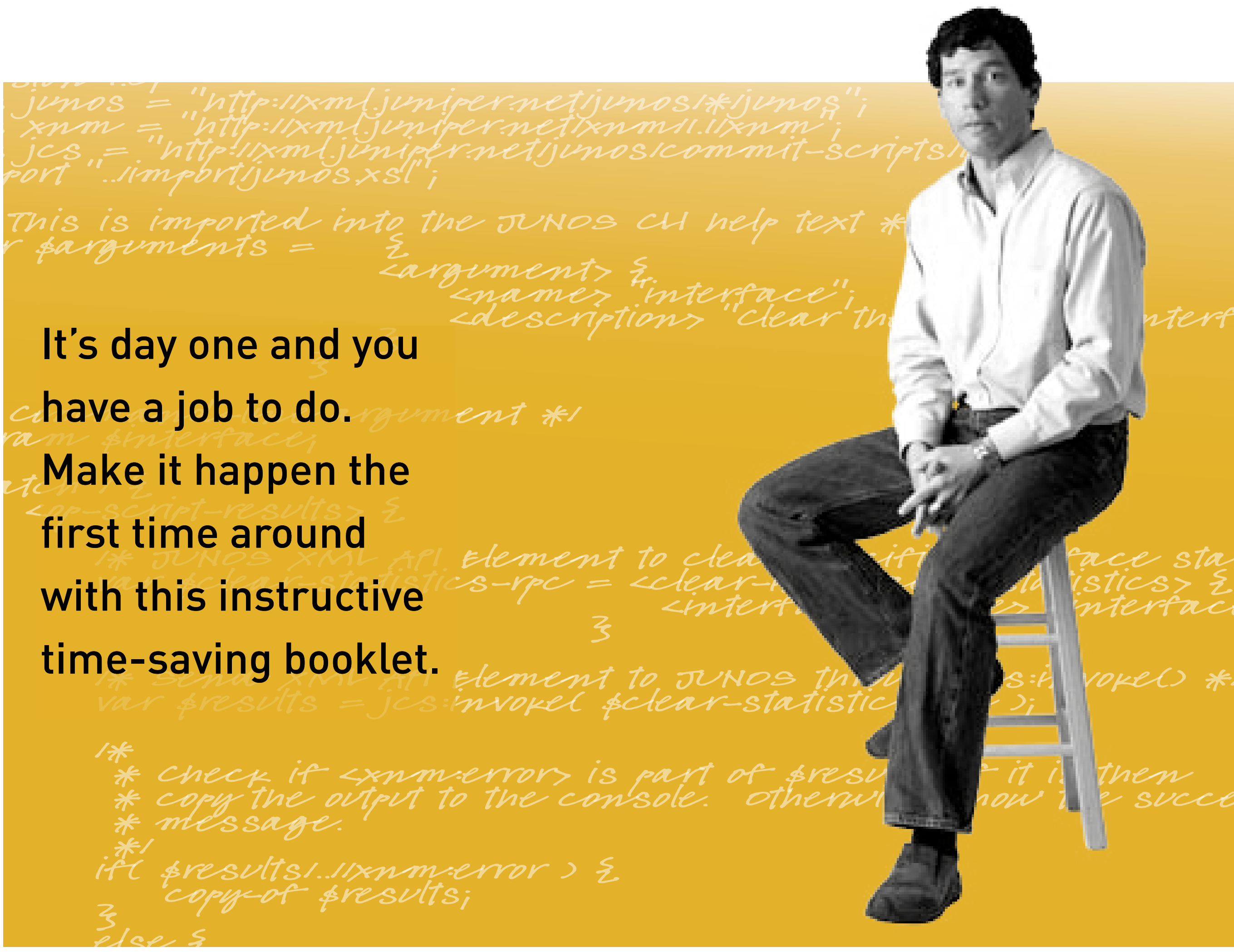
JUNOS® Software Automation Series

Day One: Applying JUNOS Automation



It's day one and you have a job to do. Make it happen the first time around with this instructive time-saving booklet.

by Curtis Call

"Just the information you need to get started on day one."

JUNOS® Software Automation Series #1

Day One: Applying JUNOS Automation

As an organization continues to work with JUNOS Software they will build a knowledge reservoir of best practices and lessons learned, a body of intelligence that can be available 24x7 to help the network run optimally. This is the possibility that JUNOS automation provides. It allows organizations to automate their accumulated intelligence through scripts which automatically control JUNOS devices according to the desired best practices.

JUNOS automation is a standard part of the JUNOS operating system available on Juniper Networks platforms including routers, switches, and security devices. This booklet introduces JUNOS automation and demonstrates how to take advantage of its potential. It also explains how to use Operation (op) scripts, one type of JUNOS automation script.

"JUNOS automation technology provides a rich portfolio of toolsets that are extremely powerful yet simple to adopt. This book demonstrates that in very little time you too can create solutions for many challenging network management tasks."

Lixun Qi, Lead IP Engineer, T-Systems North America Inc.

Day One: Applying JUNOS Automation shows you how to:

- Understand how the JUNOS automation tools work.
- Explain where to use the different JUNOS script types.
- Use reference scripts from this book and Juniper's script library.
- Interpret the XML data structures used by JUNOS devices.
- Communicate with JUNOS through the JUNOS XML API.

7100109

Juniper® NETWORKS

JUNOS® Software Automation Series

Day One: Applying JUNOS Automation

By Curtis Call

Juniper® NETWORKS

Day One Series Editor: Cathy Gadecki
Writer: Curtis Call
Copyeditor and Proofing: Nancy Koerbel
Literary Management: Ames & Eames, Inc.
Printed in the USA by Vervante Corporation.

Version History: v1 September 2009
2 3 4 5 6 7 8 9 10

About the Authors
Curtis Call is a Systems Engineer at Juniper Networks. He is JNCIE-M #43 and has eight 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. The literary manager, Patrick Ames worked with me to find the fight outlet for this material and Nancy Koerbel fine-tuned my writing. The Day One Series Editor Cathy Gadecki was instrumental in bringing this project to fruition and helped me position the content to be more instructional. Roy Lee, the JUNOS automation Product Line Manager, reviewed the manuscript several times and always found ways to clarify the presenation.
Thank you all.

A free PDF version of this booklet is available at: www.juniper.net/dayone.

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 the first day, day one.

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

What This Booklet Offers You

This first booklet in our JUNOS Software Automation series helps you to automate operations tasks in your devices. Read this booklet to learn how to use the JUNOS automation scripting toolset and how to write your first operation scripts.

When you're done with this booklet, you'll be able to:

- ✓ Understand how the JUNOS automation tools work.
- ✓ Explain where to use the different JUNOS script types.
- ✓ Use reference scripts from this book and Juniper's script library.
- ✓ Interpret the XML data structures used by JUNOS devices.
- ✓ Communicate with JUNOS through the JUNOS XML API.
- ✓ Ease how you write XML data structures using the SLAX XML abbreviated format.
- ✓ Read SLAX scripts and understand the operations they perform.
- ✓ Create your own customized operation scripts.

What You Need to Know Before Reading

Before reading this booklet, you should be familiar with the basic administrative functions of JUNOS Software. 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 Software Fundamental series, as well as the training materials available on the Fast Track portal, can help to provide this background (see the last page of this booklet for these and other references).

Other things that you will find helpful as you explore the pages of this booklet:

- ✓ Having access to a JUNOS device while reading this booklet is very useful. A number of practice examples which reinforce the concepts being taught are included. Most of these examples require creating or modifying a script and then running it on a JUNOS device in order to see and understand the effect.
- ✓ The best way to edit SLAX 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 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.

Supplemental Appendix

If you're reading the print edition of this booklet, there's another 20+ pages in the PDF supplemental appendix. Go to **www.juniper.net/dayone** and download the free PDF version of this booklet, *Day One: Applying JUNOS Automation*. The supplemental appendix is included in that PDF edition.

How to Give Us Feedback

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

Chapter 1

Introducing JUNOS Automation

Computer networks continue to improve - promising higher speeds, more capabilities, and increased reliability. Yet enhanced functionality carries with it an increase in complexity, as more technologies have to coexist and work together. This tradeoff presents a challenge to network operators who want the advantages of new opportunities but still need to keep their networks as simple as possible in order to minimize operating costs and prevent errors.

Deploying JUNOS devices within a network can reduce the level of complexity that would otherwise be present. This benefit comes from the ability to use the same operating system to control routers, switches, and security devices. Instead of having to train staff to support multiple operating systems for each type of device, only a single operating system has to be learned and maintained. This decreases the overall complexity of the network.

As an organization continues to work with JUNOS it will build a knowledge reservoir of best practices and lessons learned. Imagine if this accumulated experience could always be available to help the network run optimally. Imagine if every configuration change, every system event, and every troubleshooting step could take advantage of the organization's gathered knowledge and make use of it. Enter JUNOS automation. It allows organizations to automate their pooled intelligence through scripts that automatically control JUNOS devices according to the desired best practices.

JUNOS automation is a standard part of the JUNOS Software operating system available on all JUNOS platforms, including routers, switches, and security devices. This booklet introduces JUNOS automation and demonstrates how to take advantage of its potential. It also explains how to use operation scripts, one type of JUNOS automation script.

JUNOS automation enables an organization to embed its wealth of knowledge and experience of operations directly into its JUNOS devices:

- **Business rules automation:** compliance checks can be enforced. Change management can help to avert human error.
- **Provisioning automation**: complex configurations can be abstracted and simplified. Errors can be automatically corrected.
- **Operations automation**: customized commands and outputs can be created to streamline tasks and ease troubleshooting.
- **Event automation**: responses can be pre-defined for events allowing the device to monitor itself and react as desired.

What JUNOS Automation Can Do

JUNOS automation is a powerful suite of tools for automating the methods and procedures of operating a network. Automation can not only save your team time, it also helps to establish high performance operation of the network and to manage greater scale in the network by simplifying complex tasks.

The tool sets let you automate a majority of the commands used within the JUNOS command-line, further control the commit process, as well as automate the response to defined events. JUNOS includes three types of automation scripts, each providing different types of functionality for automation:

- Operation (op) scripts instruct JUNOS of actions to take whenever the script is called through the command-line or by another script.
- Event scripts instruct JUNOS of actions to take in response to an occurrence in a monitored set of events.
- Commit scripts instruct JUNOS of actions to take during the commit process of activating configuration changes.

MORE? To see examples of each type of script go to the online script library at www.juniper.net/scriptlibrary.

Operation (Op) Scripts

This booklet helps you to write your first op scripts. Op scripts are used in operational mode to create custom commands and to change configurations. They execute whenever called upon, either by an operator who simply enters a command request in the CLI or from within an event script (see below).

Tailor made show commands are the most common form of op scripts. An op script written for this objective gathers data from multiple show commands, processes it, and then outputs the desired information to the screen.

Another common form of op script is an automated configuration change. These op scripts perform controlled configuration changes based on supplied input from command line arguments, interactive prompts, or JUNOS show commands. The advantage of this approach is that you can code the structure of the change into the script itself. This mitigates human error and allows users with less expertise the ability to change the configuration in controlled ways.

You can also create op scripts to iteratively narrow the potential cause of network problems. These scripts run an operational mode command, process the output, determine the next appropriate action, and repeat the process until the source of the problem is determined and reported to the CLI. Op scripts can thereby give operators a running start that is immensely valuable during troubleshooting.

By uncovering the root cause, or at least helping operators to quickly shorten the list of possible causes, op scripts can speed the time to resolution. Rapid problem diagnosis is crucial during an outage; it is not uncommon for an operations team to spend hours diagnosing a problem that ultimately takes only a few minutes to repair.

Event Scripts

Event scripts are triggered automatically in response to an event (or events) such as a syslog message, SNMP trap, or timer. You can use event scripts to gather troubleshooting information in reaction to a system event, or to automatically change the configuration due to networking changes or the current time of day.

A typical event script anticipates a scenario such as: “If interface X and VPN Y go down, execute op script XYZ and log a customized message.” By watching for events specified by each organization—warning parameters that signal potential problems such as a line card crash, routing failure, or memory overload—operations teams can respond more quickly with corrective actions.

The event scripts work by processing new events. When the event process of the JUNOS Software receives events, a set of event scripts can instruct it to select and correlate specific events. An event script can also perform a set of actions, such as invoking a JUNOS command or an op script, creating a log file from the command output, and uploading the file to a given destination.

In this way, operators can better control the series of operations events from the first leading indicators. Event scripts and op scripts work together to automate early warning systems that not only detect emerging problems, but can also take immediate steps to restore normal operations. By capturing more directly relevant information faster and taking action, automation scripts can give operations teams more options earlier.

MORE? After you learn about scripting basics in this booklet, download *Day One: Automating JUNOS Operations* to learn more about using op and event scripts. Check for availability at www.juniper.net/dayone/.

Commit Scripts

Commit scripts instruct JUNOS during the commit process of configuration changes. When the user performs a commit operation, JUNOS executes each commit script in turn, passing the candidate configuration through the defined commit scripts. This allows the script to fail the commit when user-defined errors are present, to provide warnings to the committing user on the console, to log messages to the syslog, or to change the configuration automatically. As examples, you could use a commit script to enforce physical card placement and wiring standards, or for a specified logical configuration.

The true power of commit scripts becomes evident when they are used as macros. Macros greatly simplify the task of complex configurations by taking basic configuration variables as input (such as the local interface, the VPN ID, and the Site ID), and then using these to create a complete set of configuration statements (such as a VPLS interface). By limiting user input only to necessary variables, macros can ensure consistency in the configuration of a particular interface, protocol, etc. across the network.

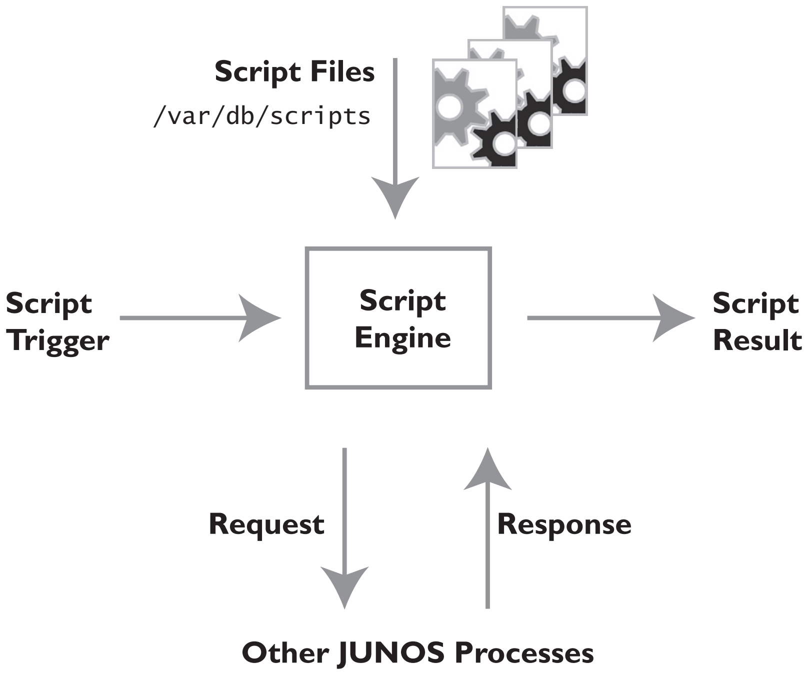
MORE? Download *Day One: Automating JUNOS Configuration* to learn more about using commit scripts. Check for availability at www.juniper.net/dayone.

How JUNOS Automation Works

JUNOS automation scripts provide a sequenced set of steps (or conditional steps) that JUNOS takes when it processes the script. JUNOS can process only those scripts specifically included as being a part of the device configuration. Only specifically permitted users have the permission to add a script to the device configuration.

Figure 1.1 outlines the basic flow of script processing. JUNOS stores scripts in predetermined `/var/db/scripts/` directories. JUNOS begins the processing of a script as a result of a trigger, defined by the type of script. For example, the trigger for processing all commit scripts is the `commit` command in configuration mode. A script engine within the JUNOS Software then processes the script line-by-line, taking the specified actions. These actions can include requests to other JUNOS processes. When the script engine completes the processing of the script, it sends the results to the output specified by the script.

Figure 1.1 The Flow of Script Processing



NOTE The script engine uses XML (eXtensible Markup Language) to read the script and communicate with other JUNOS processes. The management process daemon, known as `mgd`, of JUNOS Software includes the primary script engine for processing scripts. The event process daemon, known as eventd, also includes a script engine for monitoring events. *The Configuration and Diagnostic Automation Guid*e includes further details about how JUNOS processes scripts. Find the guide along with other the JUNOS Software documentation at www.juniper.net/tech-pubs/.

Scripting Languages

JUNOS automation scripts can be written in either of two scripting languages: XSLT or SLAX. XSLT is a standardized language designed to convert one XML document into another. While XSLT can be used to write JUNOS automation scripts, its original purpose of document conversion and the fact that it's written in XML makes it a less comfortable choice for most people.

NOTE XSLT stands for eXtensible Stylesheet Language Transformations. SLAX stands for Stylesheet Language Alternative syntaX.

Juniper developers created SLAX to provide a more user-friendly and intuitive method in which to write JUNOS scripts than XSLT. SLAX has a more readable syntax. And, it feels more natural to anyone who is familiar with reading JUNOS configurations or writing programs in the C or Perl programming languages.

This booklet focuses solely on teaching the SLAX language, as XSLT knowledge is not necessary to take advantage of JUNOS automation.

MORE? For information on how to use XSLT to write JUNOS scripts see the *Configuration and Diagnostic Automation Guide* within the JUNOS Software documentation at www.juniper.net/techpubs/.

Using JUNOS Automation with Other Systems

JUNOS automation complements existing network automation systems. Existing systems offer substantial benefits for change management, provisioning, and monitoring, but their usefulness is limited when it comes to detecting and diagnosing configuration and network problems.

Automated change management systems can only identify problems after the fact, as these packages collect information about system conditions reactively, by polling the device at predefined intervals. JUNOS automation is unique in that it provides immediate, on-box problem detection and resolution. The automation scripts are always available, always alert to potential issues, and always ready to initiate repair.

TIP Building on the JUNOS automation toolset, Juniper Networks Advanced Insight Solutions (AIS) introduces intelligent self-analysis capabilities directly into platforms run by JUNOS Software. 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. For more information visit the Juniper Networks services web page at http://www.juniper.net/us/en/products-services/technical-services/j-care/.

XML Basics

JUNOS automation scripts communicate with their host device using the XML language. While it's somewhat of a dry topic, a basic understanding of how XML is used in JUNOS Software is thereby a necessary first step in learning how to apply JUNOS scripts. This section gives you just the brief XML background that you need for writing your own scripts.

Fortunately, the SLAX language greatly simplifies how one reads and uses XML data structures. The next section explores how SLAX abstracts the described XML data structures for greater ease of use.

Displaying XML

The defined JUNOS XML API (Application Programming Interface) provides methods for JUNOS scripts to make requests. These requests can instruct other JUNOS processes to retrieve particular data or perform specific actions (see Figure 1.1). JUNOS performs the requested operation and returns the XML results to the script engine for further processing of the script.

As an example of the XML results that a JUNOS script can use, take a look at the configuration below expressed in XML mode:

```
user@JUNOS> show configuration routing-options | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.6I0/junos">
  <configuration junos:commit-seconds="1238100702" junos:commit-localtime="2009-03-26
13:51:42 PDT" junos:commit-user="user">
     <routing-options>
       <route-distinguisher-id>192.168.1.1</route-distinguisher-id>
       <autonomous-system>
         <as-number>65535</as-number>
       </autonomous-system>
     </routing-options>
  </configuration>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

At first glance this output can appear confusing, but the intuitive structure makes it simple to understand. Notice the `rpc-reply` mentioned in the first line of output. This shows the output is a reply from JUNOS providing the requested XML data.

The next line indicates that this is configuration information, and the following line begins the familiar `routing-options` configuration hierarchy. Just as in a normal configuration, the `routing-options` hierarchy contains the `route-distinguisher-id` and `autonomous-system` configurations. The XML form uses the same hierarchical approach, making it easy to understand and simple to compare against the text configuration.

The output above includes examples of key concepts necessary to understand how to communicate using the JUNOS XML API: elements, attributes, namespaces, and nodes.

Try It Yourself: Viewing JUNOS Configuration in XML

Show the following configuration hierarchy levels in XML on a JUNOS device:

```
(e.g. show configuration system | display xml)

[system]
[interfaces]
[protocols]
```

Elements

An *element* is the basic unit of information within an XML data structure. Elements can contain data such as a text string or number, or they can contain other elements. An element that contains another element is the parent of the enclosed child element. Likewise, the inner element is the child of the containing element. This creates a hierarchy, which is inherent in the XML structure, similar to the familiar JUNOS configuration hierarchy.

Elements are written by using start and end tags that provide the boundaries of the element. A tag contains the element name enclosed within < > arrows. The output above shows examples of tags, such as `<routing-options>`, a tag for the `routing-options` element. The output lists `<routing-options>` twice, once as the start tag and once as the end tag.

All the text within the start and end tags is an element's data. Both tags include the element name; however, the end tag also includes a / before the name, for example `</routing-options>`. If an element is empty, meaning it has no data or child elements, then it can be expressed using a single tag with a / following the element name, for example `<exten-sive/>`.

Here is an example XML configuration hierarchy showing four separate elements:

```
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
    <disable/>
  </interface>
</interfaces>
```

The `<interfaces>` element is the parent element of `<interface>`, which is the parent element of the `<name>` and `<disable>` elements. The `<name>` element contains the text data `ge-0/0/0`. The `<disable>` element, however, contains no data or child elements . This is why it is expressed as an empty tag instead of a start and end tag pair. Yet its presence in the XML data structure communicates that this interface has been disabled.

Attributes

Elements can include additional information in the form of *attributes*. This information is expressed by including the attribute name and value within the start tag:

```
user@JUNOS> show configuration routing-options | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.6I0/junos">
  <configuration junos:commit-seconds="1238100702" junos:commit-localtime="2009-03-26
13:51:42 PDT" junos:commit-user="user">
<snip>
```

Both the `<rpc-reply>` and `<configuration>` elements have attributes defined. For example, the `<configuration>` element has three attributes: `junos:commit-seconds`, `junos:commit-local-time`, and `junos:commit-user`. Here, the three attributes of the `<configuration>` element provide additional details about the last commit.

XML expresses the attribute value by including an equal sign (=) following the attribute name and providing the value within quotation marks as shown above. If an element has multiple attributes, they are included within the start tag separated by spaces.

Namespaces

In the last example the three attributes defined for the `<configuration>` element all started with the same word. In this case the `junos` portion of the attribute name fulfills a specific purpose: it indicates the *namespace* of the attribute.

NOTE A namespace prevents confusion between elements using the same name for different purposes. For example, there could be a `commit-seconds` attribute used by multiple computing devices, but when it is included in the `junos` namespace it becomes `junos:commit-seconds.` What the attribute indicates is now explicitly known.

More precisely, the `junos` name is actually a placeholder for the full namespace, which is `http://xml.juniper.net/junos/9.6I0/junos`. XML uses a URL for namespaces to ensure each is unique and to prevent namespace collisions. Fortunately XML and SLAX include ways to simplify the assignment of URLs to namespaces.

Defining a Namespace

Writing out the full URL-based namespace for every attribute or element can become overly verbose and tedious. For this reason XML enables the creation of a placeholder:

```
<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/9.6I0/junos”>
```

The single attribute of `<rpc-reply>` fulfills a special purpose. `xmlns` defines a XML namespace. The example declares that the `http://xml.juniper.net/junos/9.6I0/junos` namespace is the reference of `junos`. This assignment takes effect for the `<rpc-reply>` element and all of its child elements.

Using Namespaces in SLAX

JUNOS further simplifies the use of namespaces when working with SLAX scripts. Rather than using the exact JUNOS version (9.6 in the above example), JUNOS replaces the version number with a * when providing the XML data structure to the script. In this way a script can be written without reference to the exact namespace used on the JUNOS device.

With this simplification, the only steps that a SLAX script writer must follow to use namespaces correctly are:

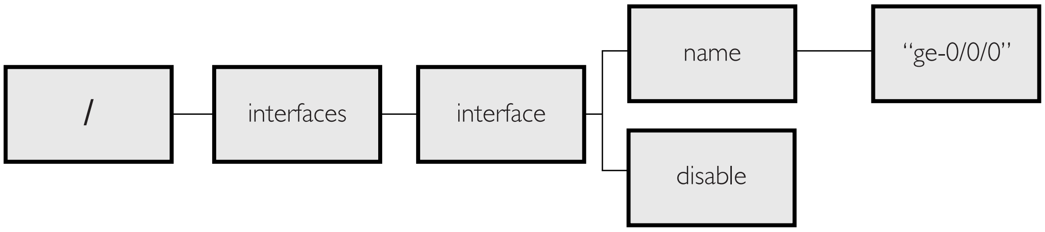
1. Copy the standard boilerplate (explained in Chapter 2) into the script.
2. Prepend the namespace placeholder (`junos, jcs, xnm`, etc.) correctly to the element or attribute name (if they have been assigned a namespace).

Nodes

When SLAX parses a XML data structure it reads it as a tree of nodes. Every element, attribute, and text data becomes a separate node on the tree. As an example, Figure 1.2 shows how SLAX would assemble all the nodes for the following configuration:

```
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
    <disable/>
  </interface>
</interfaces>
```

Figure 1.2 SLAX Tree of Nodes Example



Note that each node in Figure 1.2 is in the correct hierarchical position of the tree. In SLAX, every node tree contains a root node at its base, representing data to computers, with its syntax expressed in Figure 1.2 as `/`. Next the four element nodes appear according to their hierarchy. Lastly, a text node holds the text contents of the `<name>` element. With the XML data structure expressed in tree form, it is possible to traverse the tree from parent to child, and from sibling to sibling, in order to retrieve the necessary data.

MORE? For more details on XML you can look at http://www.w3schools.com/xml/. It is one of many online XML tutorials.

SLAX Abbreviated XML Format

Compare this XML data structure:

```
<interfaces>
  <interface>
    <name>ge-0/0/0</name>
    <disable/>
  </interface>
</interfaces>
```

To the actual configuration it represents:

```
interfaces {
  interface ge-0/0/0 {
    disable;
  }
}
```

The XML data structure is harder to read and more time-consuming to write. While XML's structure makes it very consistent and useful for representing data, its syntax is not ideal to read or manually enter. This stems from the necessity of using start and end tags for each element.

The SLAX language therefore uses an abbreviated format to describe XML data structures. This format is more congruent with the JUNOS configuration style:

```
<interfaces> {
  <interface> {
    <name> “ge-0/0/0”;
    <disable>;
  }
}
```

This is the same XML data structure as shown in the XML format example, yet in SLAX it appears more similar to the actual configuration it represents. Note one difference between the actual configuration text and its representation in the SLAX abbreviated XML format: the identifier for configuration objects appears as the first child element within an element called `<name>`, rather than appearing in the same configuration line.

As an example, `ge-0/0/0` is in line with `interface` in the normal configuration text, yet it is assigned to the `<name>` child element of the `<interface>` element in the SLAX format.

To achieve the simplification, the SLAX abbreviated XML format uses only the start tags; the end tags are no longer required. Instead, SLAX expresses the boundary of the element in one of three ways:

- If the element contains child elements then curly braces `{ }` contain the child elements (the same method used to indicate hierarchy in JUNOS).
- If the element contains data then the data is written within quotation marks (" ") and the line is terminated with a semi-colon (`;`) (similar to JUNOS configurations).
- A single start tag terminated by a semi-colon (`;`) represents empty elements with no children or text data.

Try It Yourself: Writing XML in the SLAX Abbreviated Format

Rewrite the following configuration using the SLAX abbreviated XML format:

```
system {
  host-name r1;
  login {
    message "Unauthorized access prohibited.";
  }
}
```

Chapter 2

Writing Your First Script

JUNOS automation scripts can automate many operation steps in JUNOS. This chapter provides the first glimpse of how to write a script, load it on a JUNOS device, and enable it in the configuration.

Hello World

The functionality of this first op script is very simple: when run, it displays the text "Hello World!" on the console. To run the op script, an administrator simply enters the script file name at the CLI prompt. The complete code for the Hello World script follows:

```
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> {
    <output> "Hello World!";
  }
}
```

Here is the output shown by the Hello World op script:

```
user@JUNOS> op hello-world
Hello World!
```

How to load and run the Hello World op script:

To run this op script on a device, take the following steps.

1. Save the code into a text file called `hello-world.slax`.

ALERT! All SLAX script filenames must end with the `.slax` extension.

2. Copy the script file into the `/var/db/scripts/op` directory on the JUNOS device.

3. Before you can run the script, you must enable it within the JUNOS configuration. Explicit configuration is a security precaution that prevents unauthorized scripts from being executed on the JUNOS device. Only super-users, users with the `all` permission bit, or users that have specifically been given the `maintenance` permission bit are permitted to enable or disable JUNOS scripts in the configuration. The command to enable an op script is `set system scripts op file <filename>.` So for the Hello World op script, enter:

```
set system scripts op file hello-world.slax
```

NOTE Devices with multiple routing-engines must have the script file copied into the `/var/db/scripts/op` directory on all routing-engines. The script must be enabled within the configuration of each routing-engine as well. Typically this configuration is done automatically through configuration synchronization. However, if the configurations are not synched, then the configuration must be entered manually into all routing-engines.

4. Execute the script with the `op` command followed by the script file name (without the `.slax` filename extension). For example:

```
user@JUNOS> op hello-world
```

SLAX Syntax Rules

The SLAX scripting language has a set of basic syntax rules. Chapter 1 provided some of these rules in the section on SLAX abbreviated XML format. Since JUNOS scripts contain XML elements and data structures, scripts must follow these relevant formatting rules.

The rest of the SLAX syntax rules are very similar to the JUNOS configuration syntax rules. For example, code blocks and line termination within SLAX scripts are done in the same manner as in JUNOS configuration, and the formatting of strings and comments in SLAX is also comparable to JUNOS.

Code Blocks

A JUNOS configuration indicates hierarchy through the use of curly braces { }:

```
interfaces {
  interface ge-0/0/0 {
    disable;
  }
}
```

In the above example, the `interfaces` configuration hierarchy contains the `interface ge-0/0/0` hierarchy, which contains `disable`. The entire hierarchical relationship is clearly defined with the use of curly braces.

The SLAX scripting language follows a similar style, enclosing distinct code blocks within curly braces to indicate their hierarchy and bounds. Review this portion of the configuration from the Hello World script:

```
match / {
  <op-script-results> {
    <output> "Hello World!";
  }
}
```

Curly braces bound the `match /` code block. This provides a clear boundary indicating exactly where the code block starts, where it stops, and what code it contains.

Line Termination

The following lines end with a semi-colon:

```
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";
```

Each of these is an example of an individual statement. Individual statements are not part of a code-block. SLAX terminates individual statements with a semi-colon (the same as in JUNOS configuration). The semi-colon tells the script engine in JUNOS that the end of the line has been reached.

String Values

A string is a sequence of text characters. "`Hello World!`"and "`../import/junos.xsl`" are examples of strings in the Hello World script. Scripts must always enclose string values within quotes. In this way the script engine knows that the text is intended as a string value.

This method is very similar to how JUNOS handles strings in a configuration. The difference: in a JUNOS configuration, quotation marks are generally only required when the text includes a space; while in a SLAX script, quotation marks are always required whether a space is present or not.

NOTE SLAX allows the use of either single quotes '`String Value`' or double quotes "`String Value`", but the character used to open the string must also be used to close it.

Adding Comments

The regular use of comments within a script is very helpful and highly recommended. Comments provide insight into the logic of the script and the expectations it is working under. This can be beneficial to those that did not write the script and are unfamiliar with the design decisions that influenced it. Comments can also be useful for the script author who might need to revise the script months later.

Comments in SLAX scripts are written within the delimiters `/*` and `*/` such as:

```
 /* This is a comment. */
```

This syntax is similar to how comments appear in a JUNOS configuration when configuration commands are annotated.

NOTE In JUNOS comments are indicated in two ways, either within /* and */ or following a #. SLAX scripts only support the delimiters /* and */.

Comments can be included anywhere within your script:

```
match / {
  <op-script-results> {
    /* Display this string on the console */
    <output> “Hello World!”;
  }
}
```

Create multi-line comments by entering the terminating */ on a separate line from the starting /*:

```
/*
 * This is a simple script which outputs “Hello World!” to the console.
 */
```

Try It Yourself: Adding Comments to the Hello World Script

Make the following modifications to the Hello World script:

1. Add a multi-line comment at the beginning that describes the purpose of the script.

2. Add an additional comment before the `<output> “Hello World!”;` line to state that it is writing to the console.

After making the two modifications, replace the prior version of `hello-world.slax` on your JUNOS device with the new version. Execute the script again and verify that the new comments did not change the operation.

Understanding the Result Tree

JUNOS automation requires a communication method so that scripts can instruct JUNOS to perform desired actions. For example, the Hello World script causes JUNOS to display “Hello World!” on the console. In the Hello World script, the `<output>` element within the result tree provides this request.

Using the result tree is the simplest way for a script to provide instructions for JUNOS. The result tree is a XML data structure created by the processing of the script and delivered to the script engine after the script terminates. During operation, the script specifies the XML elements to include in the result tree. Once the script has finished, JUNOS follows the instructions of the completed result tree.

Writing to the Result Tree

Writing to the result tree within a script is easy. XML elements are simply embedded within the SLAX script in the abbreviated XML format. When the script engine arrives at a line that consists of a XML element it automatically attaches that element into the result tree.

NOTE Some XML elements take effect within the script itself and are not written into the result tree. These are special purpose elements such as `<xsl:sort>` and `<xsl:message>`. They constitute the exception to the rule; typically all embedded XML elements are written to the result tree.

Processing these lines creates the result tree in the Hello World script:

```
match / {
  <op-script-results> {
    <output> "Hello World!";
  }
}
```

`<op-script-results>` is an XML element with a child element of `<output>`. When the script engine begins executing the script it reaches this XML data structure, recognizes these as XML elements, and writes them to the result tree as:

```
<op-script-results> {
  <output> "Hello World!";
}
```

The parent element in the example above is `<op-script-results>`. This is always the top-level element in an op script result tree. This element indicates to JUNOS that instructions are coming from an op script. There is no action performed by the `<op-script-results>` element, it simply contains the child elements. It is the child elements that provide instructions for JUNOS to process.

The `<output>` element is the most common element found in the result tree of op scripts. As the name implies, it outputs an associated string. Specifically, it instructs JUNOS to display the string to the console followed by a line-feed. A script can include multiple `<output>` elements, with each string appearing on a different line:

If you wish to include line-feeds within your text string then use the `\n` escape character: `<output> "First Line\nSecond Line";`

```
<op-script-results> {
  <output> "Hello World!";
  <output> "I'm Home!";
}
```

Results in the following output:

```
Hello World!
I'm Home!
```

Try It Yourself: Adding Additional Output to the Hello World Script

Modify the Hello World script by adding two additional lines of output to the console above the "Hello World!" string.

Replace the prior version of `hello-world.slax` on your JUNOS device with the changed version. Execute the script again and see the effect the new `<output>` elements have on the script output.

Importing Script Code

The Hello World script example includes the following line:

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

This specific statement loads all the code from the `/var/db/scripts/import/junos.xsl` script file into your op script prior to execution. Importing allows the use of common code within multiple scripts without having to copy and paste the actual text from one script to the other. The `junos.xsl` script file is included as part of the standard JUNOS distribution. It contains default templates and parameters. Your scripts should always include the above line to import this file.

For information on the contents of `junos.xsl` see *The Configuration and Diagnostic Automation Guide* at www.juniper.net/techpubs/.

The Main Template

When writing a script, all code and result tree elements must be included within a code structure known as a template. When the script engine in JUNOS first executes a script, it searches the script file for the main template. The script engine then begins executing the instructions and writing the included result tree elements. For op scripts, the main template is `match /`.

Note in the Hello World script below: the presence of the `match /` template, the curly braces { } enclosing the code block, and the XML elements to add to the result tree included within the block.

```
match / {
  <op-script-results> {
    <output> "Hello World!";
  }
}
```

Using the Op Script Boilerplate

When writing JUNOS op scripts, work from the standard 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 / {
  <op-script-results> {

   /* Your script code goes here */

  }
}
```

The boilerplate simplifies script writing by providing the needed namespace URLs (see Chapter 1) along with other components. Copy and paste the boilerplate and add your script code within it. The boilerplate includes the following components:

- **version:** while version 1.0 is currently the only available version of the SLAX language, the `version` line is required at the beginning of all JUNOS scripts.
- **ns:** a `ns` statement defines a namespace prefix and its associated namespace URL. The following three namespaces must be included in all JUNOS scripts:

```
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”;
```

It is easiest to just copy and paste these namespaces into each new script as part of the boilerplate rather than trying to type them out by hand.

The boilerplate includes the `<op-script-results>` element to simplify writing of op scripts. SLAX statements can be included within the `<op-script-results>` code block without interfering with the created result tree. The script engine can differentiate between statements to execute and XML elements to add to the tree.

- **import:** the import statement is used to import code from one script into the current script. As the `junos.xsl` script contains useful default templates and parameters, all scripts should import this file. The `import “../import/junos.xsl”;` line from the boilerplate is all a script needs to accomplish this.
- **match /:** this code block is the main template of the op script. In the standard boilerplate it includes the `<op-script-results>` result tree element.

Try It Yourself: Writing Your Own Script Using the Boilerplate

Using the configuration boilerplate, create a new op script that outputs three separate lines of text to the console. Copy this script to your JUNOS device and enable it. Now you can verify it by executing it from the command-line.

Chapter 3

Understanding SLAX Language Fundamentals

The last chapter covered the syntax rules of the SLAX scripting language as well as the boilerplate used when creating a new op script. It also explored the Hello World op script and demonstrated how to write text to the console. This chapter digs deeper into the fundamentals of the SLAX language and further explains its capabilities.

1. The `jcs`, `xnm`, and `junos` namespaces are reserved. Do not use any of these namespaces when creating variables, parameters, elements, or templates.

2. Do not start any names with "`junos`".

While the following serve as guidelines, they are also best practices that let your scripts conform to JUNOS configuration naming standards as well as to official JUNOS scripts.

1. Write variables, parameters, elements, and templates entirely in lowercase.

2. Separate multiple words with a dash (-), for example: `gig-interface`.

Variables

In the SLAX language, a variable is a reference to an assigned value. The variable name is used within the script, and the script engine substitutes the value in its place when it executes the script. SLAX variables are immutable; they cannot be changed. This might seem strange to those who are accustomed to other programming languages, but in SLAX variables always refer to the value to which they were first assigned.

Data Types

There are five data types defined in the SLAX scripting language:

- **string:** a sequence of text characters, for example "`Hello World!`".
- **number:** numbers are stored as floating points so decimals are permitted.
- **boolean:** used for conditional operations; evaluated as either true or false.
- **result tree fragment:** a portion of the result tree. By default, all XML elements that are embedded in a script are written to the result tree. It is possible, however, to redirect this XML data to a variable instead. The variable stores the data as an unparsed XML data

structure, as such no additional data can be extracted from it. The script can only use the unparsed data to write to the result tree later or to convert to a node-set or string.

- **node-set:** an unordered set of XML nodes. A node-set consists of parsed XML data, so information can be extracted from it. Typically, node-sets are the result of a query to JUNOS for information, a location path, or a converted result tree fragment.

Declaring Variables

Variables are all declared using the `var` statement. Variable names are always preceded by a dollar sign:

```
var $example-string = “Example”;
```

The data type of the variable is automatically determined based on the assigned value. Here are examples of how to declare variables in each of the different data types:

- **string:** `var $example-string = “Example”;`
- **number:** `var $example-number = 15;`
- **boolean:** `var $example-boolean = ( 15 == 15 );`
- **result tree fragment:**

```
var $example-rtf = {
  <system> {
    <host-name> “R1”;
  }
}
```

- **node-set:** `var $example-node-set = jcs:invoke(“get-interface-information”);`

BEST PRACTICE To allow your scripts to be compatible with future JUNOS script functionality, always follow these rules when naming variables, parameters, elements, and templates.

Using Variables

Once declared, a script can use the variable to reference the represented value. When using variables their full case-sensitive name must be used, including the preceding dollar sign:

```
match / {
  <op-script-results> {
    var $router-name = “R1”;
    <output> $router-name;
  }
}
```

The above example shows the main template of an op script. This script declares a variable of `$router-name` with a string value of "`R1`" assigned to it. The script then includes this variable as the content of the `<output>` result tree element that writes "`R1`" to the console.

Scope of Variables

Variables are only usable within a limited scope. A scope is the code hierarchy in which a variable is declared. Each variable can be used within its own scope, as well as in other specific scopes that also fall within the declared scope.

The following are the main types of scopes for variables:

- **global variable:** refers to any variable declared outside of all templates. Global variables can be referenced anywhere within the script.
- **template variable:** refers to variables defined within templates, such as the main template, and have a scope of only their own template. Template variables cannot be used outside of their own template.

NOTE The script code can declare variables of the same name both globally and within a template, but only one or the other is usable at a time. The template variable overrides the global variable within the template that assigns the template variable.

More specific scopes are also possible. If a variable is declared within the code block of either an `if` or `for-each` statement (which are discussed later), then it is only usable within that code block; it cannot be referenced outside of it.

Global Variables

Here is a variation of the Hello World script where the string is defined as a global variable:

```
/* hello-world.slax */
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”;

/* This is a Global Variable */
var $first-string = “Hello World!”;

match / {
  <op-script-results> {

    /* This is a variable with template scope */
    var $second-string = “Goodbye World!”;

            /* Output both variables to the console */
    <output> $first-string;
    <output> $second-string;
  }
}
```

In the above example, both the global variable `$first-string` and the template variable (sometimes called a local variable) `$second-string` are available for use within the main template. However, if additional templates are added to the script (as is discussed later in this chapter), only the global variable `$first-string` can be used within these.

NOTE Notice that the `var` statement is declared within the `<op-script-results>` XML element. The SLAX processor is able to determine that this is a line of script code rather than an XML element, and it does not try to write it to the result tree. It is common to interleave SLAX code and result tree elements in this manner within JUNOS scripts.

Operators

SLAX contains a wide variety of operators to enhance script operation. These operators enable the script to perform mathematical operations, compare values, convert data, and create complex expressions. Table 3.1 summarizes the operators available in SLAX.

Table 3.1 **SLAX Operators**

Name ... **Code**	**Example ...** **Explanation**
Addition `+`	`var $example = 1 + 1;` Assigns the value of 1 + 1 to the `$example` variable.
Subtraction, Negation `-`	`var $example = 1 - 1;` Assigns the value of 1 - 1 to the `$example` variable and changes the sign of a number from positive to negative or from negative to positive.
Multiplication `*`	`<output> 5 * 10;` Results in the value 50 being written to the console.
Division `div`	`<output> $bit-count div 8;` Divides the bits by eight, returning the byte count, and displays the result on the console (requires that the variable `$bit-count` has been initialized).
Modulo `mod`	`<output> 10 mod 3;` Returns the division remainder of two numbers. In this example the expression writes 1 to the console.
Equals `==`	`$mtu == 1500` If the value assigned to `$mtu` is 1500 then the expression resolves to true, otherwise it returns false (requires that `$mtu` has been initialized).
Does not equal `!=`	`$mtu != 1500` If `$mtu` equals 1500 then the result is false, otherwise it returns false (requires that `$mtu` has been initialized)
Less than `<`	`$hop-count < 15` Returns true if the left value is less than the right value, otherwise it returns false (requires that `$mtu` has been initialized).
Less than or equal to `<=`	`$hop-count <= 14` Returns true if the left value is less than the right value or if the two values are the same, otherwise it returns false (requires that `$mtu` has been initialized).
Greater than `>`	`$hop-count > 0` Returns true if the left value is greater than the right value, otherwise it returns false (requires that `$mtu` has been initialized).

Greater than or equal to `>=`	`$hop-count >= 1` Returns true if the left value is greater than the right value or if they are the same, otherwise it returns false.
Parenthesis `( )`	`var $result = ( $byte-count * 8 ) + 150;` Used to create complex expressions. Parenthesis function the same way as in a mathematical expression, with the expression within the parenthesis evaluated first. Parenthesis can be nested with the innermost set of parenthesis evaluated first, then the next set, and so on.
And `&&`	`$byte-count > 500000 && $byte-count < 1000000` The `&&` (and) operator combines two expressions to get one result. If either of the two expressions evaluates to false then the combined expression evaluates to false.
Or `||`	`$mtu-size != 1500 || $mtu-size > 2000` The `||` (or) operator combines two expressions to get one result. If either of the two expressions evaluates to true then the combined expression evaluates to true.
String concatenation `_`	`var $combined-string = $host-name _ " is located at " _ $location;` The underscore _ is used to concatenate multiple strings together (note that strings cannot be combined using the + operator in SLAX). In the example if `$host-name` is "r1" and `$location` is "HQ" then the value of `$combined-string` is "r1 is located at HQ".
Node-Set Union `|`	`var $all-interface-nodes = $fe-interface-nodes | $ge-interface-nodes;` The `|` operator creates a union of two node-sets. All the nodes from one set combine with the nodes in the second set. This is useful when a script needs to perform a similar operation over XML nodes that are pulled from multiple sources.
Result Tree Fragment to Node-Set Conversion `:=`	`var $new-node-set := $rtf-variable;` A result tree fragment contains an unparsed XML data structure. It is not possible to retrieve any of the embedded XML information from this data type, so the `:=` conversion operator was created. This operator converts a variable from a result tree fragment into a node-set. The script can then tell JUNOS to search the node-set for the appropriate information and extract it. Only JUNOS 9.2 and beyond supports this operator (see note next page).

NOTE There is no operator for "not" as there is in other programming languages. Instead, there is a not() function that returns the opposite boolean value of its argument.

NOTE The `:=` operator is only supported in JUNOS 9.2 and beyond. If you are using an earlier version you can use the node-set() extension function to convert a result tree fragment into a node-set. The node-set() function requires that the "`http://xmlsoft.org/XSLT/namespace`" namespace be declared, and the assigned prefix prepended to the function name. For example, if you assign the namespace to `ext` (`ns ext = "http://xmlsoft.org/XSLT/namespace";`), then you call the function as `ext:node-set(): var $node-set-variable = ext:node-set( $rtf-variable );`

Data Type Conversion

Typically it is not necessary to explicitly convert from one data type into another. The primary exception to this rule is converting from result tree fragment to node-set, but otherwise most conversions occur automatically.

When the script engine comes to an operator or a statement, and the associated date type is not of the correct type, the script engine attempts to automatically convert it. As an example, when the addition operator is encountered, the two arguments are converted into numbers.

The conversion process works in the following way, based on the original data type:

- **string:** strings which consist entirely of appropriate characters for numeric content are converted into the equivalent number, otherwise they are converted to NaN (not a number). When converting to boolean, empty strings convert to false, and non-empty strings convert to true.
- **number:** numbers are converted to strings by converting each digit into the appropriate character. The numeric value zero is converted to the boolean value false, all other numeric values are converted to the boolean value true.
- **boolean:** when converted to strings, booleans become either "true" or "false." The boolean value false is converted to the numeric value of 0. True is converted to the numeric value of 1.

- **node-set:** a node-set is converted into a string by returning the string value of the first node in the node-set. The string value is the text contents of the node as well as all its child nodes. A node-set converts into a number in a similar fashion. Empty node-sets are converted to the boolean value of false, node-sets with one or more nodes are converted to the boolean value of true.
- **result tree fragment:** result tree fragments are converted to strings by returning all the text content within the XML data structure.

The following script example shows the automatic conversion process where a string is converted to the needed number data type:

```
/* convert.slax */
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> {

    /* String variable */
    var $numeric-string = “-700”;
    /* Number variable */
    var $number = 100;

    /* Output the addition of the two variables to the console */
    <output> $numeric-string + $number;
  }
}
```

When executed, this script displays the number -600 on the console. The script engine automatically converts the string “-700” within the script into the equivalent number -700. The result of the expression of -700 + 100 is then shown.

Try It Yourself: Working with Operators

Create a new script including two variables that are assigned numeric values. Add a third variable and assign it the product of the first two variables. Display the value of the third variable on the console.

Parameters

Parameters are variables whose value is assigned by an external source. Other than their declaration they share the same rules as variables, and you can use them in a similar way.

Default Parameters

Every script begins with six parameters predefined, which are declared within `junos.xsl`. When `junos.xsl` is imported as part of the standard boilerplate, these parameters are imported as well and can be used within the script.

The default parameters provide commonly-used information for scripts:

- **`$product`:** contains the name of the local JUNOS device model
- **`$user`:** is assigned to the name of the user that executed the script
- **`$hostname`**: stores the local hostname of the JUNOS device
- **`$script`**: contains the name of the script that is currently executing
- **`$localtime`**: stores the local time when the script was executed using the following format: `Tue Jan 20 14:07:33 2009`
- **`$localtime-iso`**: provides a different format of local time: `2009-01-20 14:07:33 PST`

NOTE In JUNOS versions prior to 9.6 `$localtime-iso` is named `$localtime_iso`. The old name format will continue to be supported in 9.6 in a deprecated fashion.

Global Parameters

Parameters whose value is set by JUNOS at script initialization must be defined as global parameters. The default parameters listed above are examples of global parameters. To declare a global parameter, use the `param` statement and provide a name for the parameter. As with variables, parameter names always require a preceding "`$`".

```
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";

/* This is a global parameter */
param $interface;
```

A script can assign a default value to a global parameter. This provides a fallback value in the event that JUNOS does not give a value to the parameter. If no default value is declared and none is assigned during script processing, then the parameter defaults to an empty string. Here is an example where the `$interface` parameter defaults to "`fxp0`":

```
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";

/* This is a global parameter with a default value */
param $interface = "fxp0";
```

Command-line Arguments

Global parameters within op scripts are typically used to pass command-line arguments from JUNOS to the op script. This technique greatly increases the versatility of a script, as scripts can be written to respond differently based on the arguments provided.

Creating Command-line Arguments

Command-line arguments are always expressed as name and value pairs. When a user executes an op script and includes command-line arguments, JUNOS searches the script for a global parameter of the same name (excluding the dollar sign `$`) and assigns the value from the command-line argument to the matching parameter.

As an example, assume a user entered the following command:

```
user@JUNOS> op show-interface interface fe-0/0/0.0
```

Based on the command-line entry, JUNOS searches for a global parameter named `$interface`. If the parameter is present, then JUNOS assigns it the string value of "`fe-0/0/0.0`".

Here is an example of a script that utilizes command-line arguments:

```
/* combine-strings.slax */
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»;

/* Command-line arguments */
param $string1;
param $string2;

match / {
  <op-script-results> {

    /* Output the command-line arguments to the console */
    <output> $user _ «: Here are your combined strings: « _ $string1 _ $string2;

  }
}
```

This script shows an example of both default parameter use and command-line arguments. `$user` is a default parameter assigned to the name of the user running the script. `$string1` and `$string2` are global parameters which are populated through command-line arguments. Assume the script executed using the following command-line:

```
user@JUNOS> op combine-strings string1 "Hello" string2 " World!"
user: Here are your combined strings: Hello World!
```

As the example shows, the strings "`Hello`" and "` World!`" were properly assigned to the `$string1` and `$string2` parameters and the `$user` parameter correctly identified the username of "`user`".

Op Script Help

Remembering command-line argument names can be burdensome. It is more user-friendly to provide a quick reference as to which command-line arguments are available. JUNOS provides a way for op scripts to create additions to the JUNOS CLI help system that remind users which arguments are available for the op script.

By default, JUNOS users can enter a ? after a command to see the available completions. If a ? is entered following the `op script` command the following is displayed:

```
user@JUNOS> op combine-strings ?
Possible completions:
 <[Enter]>     Execute this command
 <name>       Argument name
 detail       Display detailed output
 |        Pipe through a command
```

This display does not give any hint of what command-line arguments can be provided to the script. The solution is to use a special purpose global variable named `$arguments` within the op script. JUNOS automatically looks for this variable when building the help contents for an op script. By following the format of the XML data structure, it is possible to add command-line arguments to the help text.

The structure required for how it uses the `$arguments` variable is:

```
<argument> {
  <name> “ArgumentName”;
  <description> “Argument description”;

}
```

Take a look at the edited script to see how it is used:

```
/* combine-strings.slax */
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»;

/* This variable defines the CLI help text */
var $arguments = {
  <argument> {
    <name> «string1»;
    <description> «The first string»;
  }
  <argument> {
    <name> «string2»;
    <description> «The second string»;
  }
}

/* Command-line arguments */
param $string1;
param $string2;

match / {
  <op-script-results> {

    /* Output the command-line arguments to the console */
    <output> $user _ «: Here are your combined strings: « _ $string1 _ $string2;

  }
}
```

Now, notice the difference when a user invokes the JUNOS CLI help:

```
user@JUNOS> op combine-strings ?
Possible completions:
 <[Enter]>     Execute this command
 <name>       Argument name
 detail       Display detailed output
 string1      The first string
 string2      The second string
 |        Pipe through a command
```

Try It Yourself: Working with Command-line Arguments

Create a new script with a command-line argument that accepts a number from the user. Include the `$arguments` global variable so that the CLI help output includes the command line argument. Perform a mathematical operation on the command-line argument and output the result to the console. Execute the op script a few times, with a different number provided on the command-line to verify that the result changes.

Conditional If Statements

The script examples to this point have been fairly basic. Conditional code execution allows more flexible functionality with the SLAX `if` statement. Using the `if` statement instructs the script engine to execute segments of code only when certain conditions are met. This means that scripts can react to values instead of only operating on them.

If Statements

The `if` statement consists of two parts: a boolean expression and a conditional code block:

```
if( $mtu == 1500 ) {
  <output> “Jumbo Frames are not enabled”;
}
```

In the above example the boolean expression is `$mtu == 1500`. It is expressed within parenthesis immediately following the `if` statement. When the expression evaluates to true then the conditional code block of the `if` statement is executed. When the expression evaluates to false then the script engine skips to the end of the statements code block and continue processing from that point. As an example, if the `$mtu` variable is assigned to the value 1500 then `<output> “Jumbo Frames are not enabled”` is written to the result tree, otherwise this string does not appear in the console output.

Else If and Else Statements

Additional possibilities can be expressed by adding `else if` and/or `else` statements:

```
if( $interface == “fxp0” ) {
  <output> “Out of Band Management”;
}
else if( $interface == “lo0” ) {
  <output> “Loopback Interface”;
}
else {
  <output> “Other”;
}
```

In this case, the script engine checks each boolean expression sequentially. The first expression that evaluates to true has its code block executed. If neither the `if` nor the `else if` evaluate to true, then the `else` code block is executed (when present). In all cases, the script engine executes a maximum of one conditional code block. If multiple boolean expressions evaluate to true, the script engine only applies the first.

Conditional Operation Example

Here is a script that shows conditional operation. It outputs the value of one default parameter, with the desired parameter being chosen through a command-line argument:

```
/* output-parameter.slax */
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»;

/* This shows the parameter argument in the CLI help output */
var $arguments = {
  <argument> {
    <name> «parameter»;
    <description> «Enter name of parameter, e.g. $user»;
  }
}

/* Command-line argument */
param $parameter;

match / {
  <op-script-results> {
    /* Output the selected parameter to the console */
    if( $parameter == «$user» ) {
      <output> «$user = « _ $user;
    }
    else if( $parameter == «$hostname» ) {
      <output> «$hostname = « _ $hostname;
    }
    else if( $parameter == «$product» ) {
      <output> «$product = « _ $product;
    }
    else if( $parameter == «$script» ) {
      <output> «$script = « _ $script;
    }
    else if( $parameter == «$localtime» ) {
      <output> «$localtime = « _ $localtime;
    }
    else if( $parameter == «$localtime-iso» ) {
      <output> «$localtime-iso = « _ $localtime-iso;
    }
    /* If nothing matches then give this message instead */
    else {
      <output> «This is not a valid default parameter: « _ $parameter;
    }
  }
}
```

Now let's see this script in action:

```
user@JUNOS> op output-parameter parameter $user
$user = user

user@JUNOS> op output-parameter parameter $script
$script = output-parameter.slax
```

Conditional Variable Assignment

One common use for the `if` statement is to conditionally assign variable values. Because a variable's initially assigned value cannot be reassigned, it is prudent to be very selective in the value bound to a variable. This can be done by declaring that a variable must have its value assigned by an `if` statement:

```
var $user-type = {
  if( $user == “john” ) {
    expr “operator”;
  }
  else if( $user == “tom” ) {
    expr “admin”;
  }
  else {
    expr “unknown”;
  }
}
```

Observe the following points in the above code. First, in order to conditionally assign a variable the entire `if` statement and any attached `else if` and `else` statements must all be enclosed within curly braces { }. Second, note the use of a new statement `expr,` which is used to write text to the result tree. What the code above actually does is write a conditionally selected string to the result tree. But this string is redirected to the `$user-type` variable making it a result tree fragment.

Suppose the `$user` default parameter equals "`john`" then the "`operator`" string is written to the variable `$user-type` as a result tree fragment. Having a data type of result tree fragment in place of a string does not cause any problems because the script engine automatically converts the data type result tree fragment to a string whenever necessary. Because of this, the script can treat the result tree fragment as if it was just a normal string variable.

Try It Yourself: Conditionally Assigning Variable Values

Create a new script with a command-line argument that can be set to either + or - , signifying the mathematical operation to perform. Create a variable that is assigned conditionally, based on the value of the command-line argument. If the command-line argument is specified as a + then two values should be added together and assigned to the variable. If the command-line argument is specified as a - then subtraction should be performed between the two values and assigned to the variable. Output the result to the console.

Named Templates

The examples provided so far have included only the main template. This is appropriate given their simple nature. However, as the complexity of a script increases it becomes more advantageous to modularize it by removing some of the functionality from the main template and placing the code into named templates instead.

A named template is a segment of code which can be called from anywhere in the script by referring to its name. When this occurs, the script engine shifts its focus to the code within the selected template until the script completes. Then the script engine returns to the calling template and continues processing from where it left the script.

Named templates can greatly enhance scripts in the following ways:

- **code re-use:** if a particular code stanza has to be repeated multiple times throughout your script then it makes sense to place it within a named template instead. This reduces the size of your script and simplifies changes.
- **self-documentation:** named templates with descriptive names clarify the script actions. A script that is written in this way is simpler to read and understand than one in which all the operations are performed in a single large main template.
- **recursion:** a useful capability of named templates is their ability to call themselves. By looping through a section of code as many times as necessary, you can program the script to reach a specific end goal.

Named Templates Syntax

To create a named template you use the `template` statement, give it a name, and provide its curly brace enclosed code block:

```
template example-template {
  /* Template code goes here */
}
```

To call a template use the `call` statement and include the name of the desired template:

```
call example-template;
```

Here is an example of a script that uses a named template:

```
/* show-user.slax */
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> {

    /* Call the display-user template */
    call display-user;

  }
}

/* This template outputs the username to the console */
template display-user {
  <output> «Your user name is « _ $user;
}
```

In this example, the main template calls the `display-user` named template. The called template is then executed, causing the `<output>` element to be written to the result tree along with its included string. Although this is a simple example, it highlights a significant fact about named templates: any elements that are written to the result tree by a named template are inserted at the point the template is called.

The main template code writes the `<op-script-results>` element to the result tree, but before doing so it calls the `display-user` template. The `display-user` template then provides the instructions to include the `<output>` element as a child element of `<op-script-results>`. The final result tree sent to JUNOS is:

```
<op-script-results> {
  <output> "Your user name is " _ $user;
}
```

Template Parameters

Template parameters are similar to global parameters – their value is set outside of the template. However, rather than being set by JUNOS they are set by the script code when it codes the named template.

To declare template parameters, include them within parenthesis following the template name:

```
template display-user( $full-name ) {
  /* Template code goes here */
}
```

A default value can be provided in the same way as a global parameter. If the script code provides no default value, and the template code does not assign a parameter value, when the template is called, then the parameter is set to an empty string.

```
template display-user( $full-name = "John Doe" ) {
  /* Template code goes here */
}
```

NOTE An alternate, more verbose, method to declare parameters is to use the `param` statement in the lines immediately following the template name.

Parameter values are assigned within the same statement that calls the named template. The assignments are made by name, not by position:

```
call display-user( $full-name = "Jane Doe" );
```

This instructs the script engine to call the `display-user` template and to give its `$full-name` parameter a value of "Jane Doe".

NOTE An alternate, more verbose, method to assign template parameter values when calling a named template is to use the `with` statement in the following manner:

```
call display-user {
  with $full-name = “Jane Doe”;
}
```

This script example shows how to use template parameters:

```
/* show-time.slax */
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> {

    /* Call the display-time template and set the $format parameter */
    call display-time( $format = “normal”);
  }
}

/* Output the localtime to the console in either iso (default) or normal format */
template display-time( $format = “iso” ) {
  if( $format == “iso” ) {
    <output> “The iso time is “ _ $localtime_iso;
  }
  else {
    <output> “The time is “ _ $localtime;
  }
}
```

In this example, the `display-time` template shows the execution time of the script in either the default ISO format or the normal format. When the script is called the `$format` parameter is set to “normal” resulting in the normal time being displayed on the console.

This op script would be more useful if it allowed the user to choose the format. The following adds a command-line argument for `$format` that lets the user do so:

```
/* show-time.slax */
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”;
```

```
/* This is imported into the JUNOS CLI help text */
var $arguments = {
  <argument> {
    <name> "desired-format";
    <description> "Choose either iso or normal";
  }
}

/* Command-line argument */
param $desired-format;

match / {
  <op-script-results> {

    /* Call the display-time template and set the $format parameter */
    call display-time( $format = $desired-format );
  }
}

/* Output the localtime to the console in either iso (default) or normal format */
template display-time( $format = "iso" ) {
  if( $format == "iso" ) {
    <output> "The iso time is " _ $localtime_iso;
  }
  else {
    <output> "The time is " _ $localtime;
  }
}
```

This modified script includes a new `desired-format` command-line argument, allowing the user to choose in which format to display the time. When the `display-time` template is called the `$format` parameter is set to the value of the `$desired-format` global parameter.

Here is an example of the output:

```
user@JUNOS> op show-time display-format iso
The iso time is 2009-05-12 21:01:10 PDT

user@JUNOS> op show-time display-format normal
The time is Tue May 12 21:01:13 2009
```

NOTE If the calling template includes variables or parameters with the same name as the parameter of the called template, then the parameter can be listed without applying an assignment. The script engine automatically sets it to the value of the variable or parameter within the calling template. For example:

```
var $display-string = "Example String";
call show-string( $display-string );
```

SHORT CUT There is an easier way to write the script on the previous page. Remember that global parameters and variables are accessible in the entire script, not just in the main template. This means that it is not necessary to pass the `$format` value to the `display-time` template as a template parameter. Instead, the `display-time` template can simply use the global parameter. The Appendix included in the PDF version of this booklet provides an example that uses the global parameter. The script operates in the same manner as the preceding one, but in this case the named template accesses the global parameter instead of relying on the main template to pass the format as a template parameter.

Redirecting the Result Tree

Creating templates that perform a subset of the script code is useful for code modularization and self-documentation. Additionally, there are many times when the calling template can process a desired result from the called named template. In this scenario, the called template is designed to perform a specific operation and then return the result. Named templates (unlike functions in other languages) do not have a direct mechanism for returning values; however, they are able to write to the result tree, and it is possible to redirect the result tree to a variable. Doing this allows called templates to effectively return a string value to the calling template by writing to the result tree. The calling template then redirects the result tree output into a variable. Here is an example of how to do this:

```
/* show-day.slax */
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> {

    /* get-day-of-week returns the day – assign it to $day-of-week */
    var $day-of-week = { call get-day-of-week(); }

    /* Output day string to the console */
    <output> $day-of-week;
  }
}
```

```
/* Extract the day of week string from the $localtime global parameter */
template get-day-of-week {
  /* Write the first three characters of the $localtime to the result tree */
  expr substring( $localtime, 1, 3 );
}
```

The `get-day-of-week` template extracts the day string from `$localtime` and writes it to the result tree. The calling template can then redirect the result tree output and store it within a variable instead. As seen in the script, the syntax used to accomplish this is similar to the syntax used for conditional variable assignment. In both cases the script requires the use of curly braces around the code, which writes to the result tree:

```
var $day-of-week = { call get-day-of-week(); }
```

By extracting the day string from `$localtime` and writing it to the result tree through a named template, the script can call this named template multiple times. The named template can also be easily copied from one script into another that requires similar functionality. To retrieve the day string the script takes advantage of the `substring()` function within the `get-day-of-week` template. The last section of this chapter covers using functions.

Try It Yourself: Working with Named Templates

Create a new script that contains a named template. The template should write a string to the result tree. Redirect this into a variable in the calling template and output the variable value to the console.

Functions

Functions are coded procedures within the script engine. A script can invoke functions, which take arguments, perform a specific action, and return the result. This process might sound similar to a named template, but there are large differences between the two:

- A named template is actual script code, whereas a function is part of the underlying JUNOS operating system itself.
- Values are provided to named templates through the use of parameters, which are assigned by name, but functions use arguments where a precise order is mandated.
- Functions actually return results, whereas named templates can only write to the result tree and have that result tree fragment redirected to a variable in the calling function.

The syntax of functions differs from that of named templates as well. The `call` statement is not used; only the function name is provided and the required arguments specified within parenthesis:

```
expr substring( $localtime, 1, 3 );
```

Functions return values, as an example the substring() function returns a string. The above code writes the string value to the result tree. The script code could assign the string value to a variable instead using the following syntax:

```
var $day-string = substring( $localtime, 1, 3 );
```

Note the difference between assigning a value to a variable from a named template versus from a function:

```
var $day-of-week = { call get-day-of-week(); }
```

String Functions

String functions are used regularly within scripts. Table 3.2 lists some of the most common and useful of the string functions.

Table 3.2 String Functions

Function	**Example** **Explanation**
`substring( string-value, starting-index, length )`	`var $hello-string = substring( “Hello World”, 1, 5 );` Takes a starting string and returns the substring that begins at the specified index and extends for the given length. This example results in the `$hello-string` being assigned the string value “`Hello`”. In SLAX, indexes always begin with 1.
`substring-before( string-value-1, string-value-2 )`	`var $hello-string = substring-before( “Hello World”, “ “ );` Returns a substring of `string-value-1`, but in this case the size of the substring is determined by the location of `string-value-2` within `string-value-1`. The function returns the entire portion of `string-value-1` up to `string-value-2`. The example results in `$hello-string` being assigned the string value “`Hello`”.
`substring-after( string-value-1, string-value-2 )`	`var $world-string = substring-after( “Hello World”, “ “ );` Returns the portion of `string-value-1` which comes after `string-value-2` [`i.e.,`the reverse logic of `substring-before()`]. This example sets `$world-string` to the string value “`World`”.

`contains( string-value-1, string-value-2 )`	`if( contains( $interface-name, “ge-” ) ) {`
	`  <output> “The interface is Gigabit-Ethernet”;`
	`}`
	
	Returns a boolean value of true or false. If `string-value-1` contains `string-value-2` then it returns true, otherwise it returns false. The example shows the `contains()` function. This code uses it to determine if an interface is a ge or not based on the presence of the second string “`ge-`” in the string `$interface-name`. If the returned value is true then a string is written to the result tree.

`starts-with( string-value-1, string-value-2 )`	`if( starts-with( $interface-name, “ge-” ) ) {`
	`  <output> “The interface is Gigabit-Ethernet”;`
	`}`
	
	Returns a boolean value of true if `string-value-1` begins with `string-value-2`, otherwise it returns false. In the example, if `$interface-name` begins with “`ge-`” then a string is written to the result tree.

`string-length( string-value )`	`expr string-length(“ospf”);` Returns the number of characters within the string. The example causes the value 4 to be written to the result tree.
`translate( string-value, from-string, to-string )`	`var $new-string = translate( $string, “abcdefghijklmnopqr stuvwxyz”, “ABCDEFGHIJKLMNOPQRSTUVWXYZ” );` The `translate()` function translates the characters within the `string-value`, where the function translates any matching characters within `from-string` to their corresponding characters in `to-string` and returns the result. This example translates a string into upper-case.

MORE? This booklet only covers a portion of the available functions. Additional functions are discussed in the *Configuration and Diagnostic Automation Guide* at www.juniper.net/techpubs.

jcs:printf()

Printing unformatted output to the screen is sufficient for some scripts, but in many cases it is more user-friendly to have formatted script output where each line follows the same column spacing. The `jcs:printf()` function is used in op scripts for this purpose. It returns a string based on the formatting instructions and values provided in the arguments.

NOTE The `jcs:printf()` function does not output directly to the console, it only returns a formatted string. This string can then be output to the console as desired.

The syntax for `jcs:printf()` is the following:

```
jcs:printf( “expression”, value1, value2, ..., valuex );
```

The string expression contains embedded placeholders that indicate where each value should be inserted, as well as the format in which they should be placed. Here is an example:

```
<output> “123456789012345678901234567890”;
<output> jcs:printf(“%-10s%-10s”,“OSPF”,“ISIS” );
<output> jcs:printf(“%10s%10s”, “OSPF”, “ISIS” );
```

There are two embedded placeholders in the expression of each of these `jcs:printf()` function calls. An embedded placeholder is indicated by a `%` followed by any flags, the width of the field, and the value type. In the first case:

```
<output> jcs:printf(“%-10s%-10s”, “OSPF”, “ISIS” );
```

The `%-10s` indicates that a string value is inserted in a 10 space column that is left justified. This is repeated twice, once for “`OSPF`” and once for “`ISIS`”. In the second case:

```
<output> jcs:printf(“%10s%10s”, “OSPF”, “ISIS” );
```

The `jcs:printf()` function supports most standard printf formats which can be found in any guide to the printf function, used by C and other languages. There are also some JUNOS specific formats that can be found in the *Configuration and Diagnostic Automation Guide* within the JUNOS Software Documentation at www.juniper.net/techpubs.

The size and number of fields are the same but they lack the – flag, which causes them to be right justified. Here is the output that is displayed based on the three lines of code above:

```
user@JUNOS> op show-pretty-output
123456789012345678901234567890
   OSPF  ISIS
         OSPF      ISIS
```

Through proper usage of the `jcs:printf()` function an op script can produce output that looks just as well formatted as the normal JUNOS operational commands.

Try It Yourself: Working with Functions

Create a new script with a variable assigned to the value “Juniper Networks”. Output the following to the console on separate lines:

1. The variable value
2. The variable value - right justified in a 20 space field
3. The string length of the variable
4. The substring before the space
5. The string converted entirely into uppercase

Chapter 4

Communicating with JUNOS

The last two chapters covered most of the necessary SLAX syntax and statements used to build functional scripts, but the real potential of JUNOS automation is accomplished through direct communication with JUNOS devices. This chapter discusses the process that allows the script to interact with a JUNOS device, including retrieval of operational information, writing to the syslog, and working with the configuration.

Invoking Operational Commands

Most operational commands that can be executed manually on the JUNOS CLI prompt can also be invoked within an automation script. JUNOS processes the commands in the same manner as if they were run from the CLI. However, JUNOS generates their output in XML and provides it to the script. The script can then parse the results and perform any desired actions based on the gathered information.

JUNOS XML API

Invoking operational commands in JUNOS is possible through use of the JUNOS XML API. The script code sends API Elements to JUNOS, which then processes the received elements, performs the associated actions, and returns the results to the script.

Many operational commands are mapped directly to an API Element. For example, the "`clear interfaces statistics`" CLI command is mapped to `<clear-interfaces-statistics>`. To see the list of mapping between CLI commands and API Elements consult the *JUNOS XML Operational API Reference Guide* (some examples are in Table 4.1). Operational commands that do not have a specific API Element can be executed by using the `<command>` element with the CLI command as its text content:

```
<command> "show route";
```

Table 4.1 **API Element Examples**

API Element	CLI Command
`<get-configuration>`	`show configuration`
`<get-isis-adjacency-information>`	`show isis adjacency`
`<get-ospf-interface-information>`	`show ospf neighbor`
`<get-bgp-neighbor-information>`	`show bgp neighbor`
`<get-chassis-inventory>`	`show chassis hardware`
`<get-interface-information>`	`show interfaces`
`<get-route-information>`	`show route`
`<get-software-information>`	`show version`

jcs:invoke()

The Script code sends API elements to JUNOS by calling the `jcs:invoke()` function and providing the element as an argument. The API Element can be expressed either through a result tree fragment variable, or it can be a string containing the API Element's name. Any results from `jcs:invoke()` are returned as a node-set and should be assigned to a variable:

```
var $clear-statistics-rpc = <clear-interfaces-statistics-all>;
var $results = jcs:invoke( $clear-statistics-rpc );

or
var $results = jcs:invoke( "clear-interfaces-statistics-all" );
```

In both cases, the `clear interfaces statistics all` command is invoked, and any XML results are assigned to the `$results` variable. The difference between specifying the API Element in XML versus providing the string name is that XML attributes, text content, and child elements can only be included when the API Element is expressed as an XML data structure assigned to a variable that is passed to `jcs:invoke()`. Shown here where the interface to be cleared is provided as a command-line argument:

```
/* clear-statistics.slax */
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";

/* This is imported into the JUNOS CLI help text */
var $arguments =  {
          <argument> {
            <name> "interface";
            <description> "Clear the specified interface statistics";
          }
        }

/* Command-line argument */
param $interface;

match / {
  <op-script-results> {

    /* JUNOS XML API Element to clear specific interface statistics */
    var $clear-statistics-rpc = <clear-interfaces-statistics> {
                <interface-name> $interface;
              }

    /* Send XML API Element to JUNOS through jcs:invoke() */
    var $results = jcs:invoke( $clear-statistics-rpc );

    /* Copy XML contents of $results to the result tree */
    copy-of $results;

  }
}
```

In this example, the `<clear-interfaces-statistics>` API Element is used to clear the statistics of the interface that is specified within the command-line of the op script. The API Element is sent to JUNOS by the `jcs:invoke()` function and the results of the operation are stored in the `$results` variable.

Finally, a new statement can be seen in the script: `copy-of`. The `copy-of` statement is used to copy the contents of a result tree fragment variable or a node-set to the result tree. The script code above does this to communicate any error messages. If `<clear-interfaces-statistics>` executes successfully then no result is provided, but if there is an error then a `<xnm:error>` element is returned with a `<message>` child element. `<xnm:error>` is a valid op script result tree element, so copying it to the result tree causes the included `<message>` to be displayed to the console as an error message:

```
user@JUNOS> op clear-statistics interface ge-13/0/0
error: device ge-13/0/0 not found
```

NOTE The JUNOS device processes the received API Elements from `jcs:invoke()` immediately rather than waiting for the script to terminate as with the result tree.

Try It Yourself: Invoking JUNOS operational commands

Following the example of the clear-statistics op script shown in this section, create an op script that reboots the system. (Hint: The XML API Element needed is `<request-reboot>`).

Retrieving Data

In the last section, the clear statistics script used the `copy-of` statement to send the XML contents of the `$results` variable to the result tree. But what exactly were those contents?

The simplest way to know what XML results are returned from `jcs:invoke()` is to run the command manually with `| display xml` appended:

```
user@JUNOS> op clear-statistics interface ge-13/0/0 | display xml
<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/9.0R4/junos”>
  <xnm:error xmlns=”http://xml.juniper.net/xnm/1.1/xnm”>
    <source-daemon xmlns=””>
      ifinfo
    </source-daemon>
    <message xmlns=»»>
      device ge-13/0/0 not found
    </message>
  </xnm:error>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

When `jcs:invoke()` returns its XML result, it assigns the child element of `<rpc-reply>` to the `$results` variable. In this case that is `<xnm:error>`. In the clear statistics script the returned XML data structure is just copied into the result tree. But consider the following modification, which allows a success message to be displayed when no error occurs:

```
/* clear-statistics.slax */
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”;

/* This is imported into the JUNOS CLI help text */
var $arguments =  {
            <argument> {
              <name> “interface”;
              <description> “Clear the specified interface statistics”;
            }
          }

/* Command-line argument */
param $interface;

match / {
  <op-script-results> {

    /* JUNOS XML API Element to clear specific interface statistics */
    var $clear-statistics-rpc = <clear-interfaces-statistics> {
                <interface-name> $interface;
              }

    /* Send XML API Element to JUNOS through jcs:invoke() */
    var $results = jcs:invoke( $clear-statistics-rpc );

    /*
     * Check if <xnm:error> is part of $results, if it is then
     * copy the output to the console. Otherwise show the success
     * message.
     */
    if( $results/..//xnm:error ) {
      copy-of $results;
    }
    else {
      <output> “Statistics Cleared”;
    }
  }
}
```

The change here is the addition of an `if` statement with the following test: `$results/..//xnm:error`. This is an example of a location path that pulls data from the XML data structure and assigns it to the `$results` variable. If a `<xnm:error>` element is present in the results then the expression evaluates to true and the XML contents of `$results` are copied to the result tree. Otherwise "`Statistics Cleared`" is output to the console showing that the op script has successfully cleared the interface statistics.

Location Paths

Location paths are used to extract data from an XML structure. They follow a fixed syntax that dictates which nodes should be retrieved and included within the output. The results of a location path are communicated as a node-set that can be assigned to a variable or used within a SLAX statement such as `if` or `for-each`.

This example helps better illustrate the purpose and usage of location paths. Consider the following XML data structure:

```
<system-uptime-information xmlns="http://xml.juniper.net/junos/9.0R4/junos">
   <current-time>
     <date-time junos:seconds="1242673659">2009-05-18 12:07:39 PDT</date-time>
   </current-time>
   <system-booted-time>
     <date-time junos:seconds="1242424838">2009-05-15 15:00:38 PDT</date-time>
     <time-length junos:seconds="248821">2d 21:07</time-length>
   </system-booted-time>
   <protocols-started-time>
     <date-time junos:seconds="1242424912">2009-05-15 15:01:52 PDT</date-time>
     <time-length junos:seconds="248747">2d 21:05</time-length>
   </protocols-started-time>
   <last-configured-time>
     <date-time junos:seconds="1242424900">2009-05-15 15:01:40 PDT</date-time>
     <time-length junos:seconds="248759">2d 21:05</time-length>
     <user>root</user>
   </last-configured-time>
   <uptime-information>
     <date-time junos:seconds="1242673659">12:07PM</date-time>
     <up-time junos:seconds="248851">2 days, 21:07</up-time>
     <active-user-count junos:format="2 users">2</active-user-count>
     <load-average-1>0.00</load-average-1>
     <load-average-5>0.00</load-average-5>
     <load-average-15>0.00</load-average-15>
   </uptime-information>
  </system-uptime-information>
```

This is the XML output of the operational command `show system uptime` (the API Element is `<get-system-uptime-information>`). When this command is executed by `jcs:invoke()` the `<system-uptime-information>` element is returned as the XML result. Once this XML data has been retrieved and placed into a node-set variable it is possible for location paths to extract the embedded information.

Using location paths is similar to locating files within a file system. The search starts at a particular reference point and the use of the forward slash / indicates that the search path will move to a child of the current reference point. As an example, assume that `$result` is the name of the node-set variable which has been assigned the above XML data structure. If the boot-time value is desired then it can be retrieved with the following location-path:

```
$results/system-booted-time/date-time
```

This element node could be assigned to another variable:

```
var $date-time = $results/system-booted-time/date-time;
```

Or the text value of the date-time element node could be output to the console:

```
<output> $results/system-booted-time/date-time;
```

Reviewing the location path shown above, `$results` has a default reference point of the `<system-uptime-information>` element node. This is the starting context node from which all location paths defined using this variable are based.

TIP An easy way to verify the starting reference point of a variable is to use the `name()` function. For example: `<output> name( $results );` would display the XML element name to the console.

The / indicates a location path step. When taking a step, the default action is to look at the child of the context node. The location path example specifies `<system-booted-time>` so all the child nodes of `<system-uptime-information>` that are named `<system-booted-time>` are selected (in this case, only one). Next the second / indicates another location path step. The context node within the location path changes at each step, now it has become the `<system-booted-time>` element. Once again the default step is used to search for a child element named `<date-time>` (there is only one in this output). This is the end of the location path so the `<date-time>` element node is returned. Examples of location paths using the prior XML data structure:

Was the router booted this year?

```
If( contains( $results/system-booted-time/date-time, “2009” ) ) {
  /* conditional code goes here... */
}
```

How many users are currently online?

```
var $user-count = $results/uptime-information/active-user-count;
```

NOTE A // can be used instead of a / if the location path should match on the desired child node, no matter how deep into the hierarchy it appears. For example `$results//active-user-count` would return the same output as `$results/uptime-information/active-user-count`.

Parent Axis

By default, the child axis is used for each location path step, but what if the script needs to go in the opposite direction?

Assume the following has been set in the starting-template:

```
var $results = jcs:invoke( “get-system-uptime-information” );
var $date-time = $results/system-booted-time/date-time;
call display-boot-time( $date-time );
```

The `display-boot-time` template has been called with its `$date-time` parameter set to the `$date-time` variable in the calling template, which consists of a single element node `<date-time>`. But the `display-boot-time` template is intended to display both the `<date-time>` text value as well as its sibling element `<time-length>`. In order to do this, there must be some way to retrieve `<time-length>` while starting with `<date-time>` as a reference point.

The solution is to use the parent axis instead of the default child axis. Here is an example of how the template could be written to output both of the element node values:

```
template display-boot-time( $date-time ) {
  <output> “Here is the boot time: “ _ $date-time;
  <output> “Here is the time length: “ _ $date-time/../time-length;
}
```

This template outputs both the `<date-time>` element node value to the console and the `<time-length>` element node value. The `$date-time` parameter is initialized to the `<date-time>` node so the first output line can be printed by referring to the variable itself. For the second line a new location path operator is introduced: the `..` parent axis abbreviation. Just like with a file system `cd ..` command the `..` causes the location path to search the parent of the context node rather than its children. So the path goes from `<date-time>` to the `<system-booted-time>` parent. A following / indicates another step, this time using the default child axis, after which the element node `<time-length>` is selected.

Attribute Axis

The child and parent axis are the most commonly used axes within location paths. The next most needed axis is the attribute axis. It is through this axis that attribute nodes can be extracted from XML data structures.

Take a look at the `<system-uptime-information>` XML output again and note the number of `junos:seconds` attributes that are included. For every time output there is also a corresponding `junos:seconds` attribute applied to the element. This attribute contains the millisecond value as an alternative to the complete date/time string.

If this millisecond value is desired instead of the more verbose string then it can be retrieved through the attribute axis, which is specified by using the `@` abbreviation:

```
var $results = jcs:invoke( “get-system-uptime-information” );
var $date-time = $results/system-booted-time/date-time/@junos:seconds;
<output> “Boot time milliseconds: “ _ $date-time;
```

In this case an additional location path step has been added. Rather than stopping at the date-time node, the location path goes further and retrieves the `junos:seconds` attribute node by using the attribute axis as denoted by the `@`. As a result, the `$date-time` variable is assigned to the milliseconds value, which is then output to the console.

MORE? There are thirteen different axes that can appear in a location path but most are rarely used. To read about the other location path axes consult the XPATH specification: http://www.w3.org/TR/xpath#axes/.

Predicates

It is often necessary to be more selective about which nodes are extracted than is shown in the prior examples. If there are multiple occurrences of a node of the same name then location paths, based on the syntax above, would return all of them instead of only a single node. If the goal is to process all of the returned nodes in the same manner then that might be the desired behavior. But if the intent is to only extract a specific node among identically named nodes then a predicate must be used to indicate this within the location path.

For example, the output of `show interface terse` displays all the interfaces available on the JUNOS device. The corresponding `<get-interface-information>` API Element returns the same list of interfaces. That might be what a script needs, or it might not. It's important to be aware of the default behavior – which is to return all of the nodes with the same name – and to know how to sharpen the location path using predicates when a more specific query is desired.

Predicates are filters that prevent non-matching nodes from being included in the location path result. The predicate expression is enclosed within brackets `[ ]` for example:

```
var $get-interface-rpc = <get-interface-information> {
            <terse>;
          }
var $results = jcs:invoke( $get-interface-rpc );
var $ge-node = $results/physical-interface[name==”ge-1/0/0”];
```

Here is the XML output of `<get-interface-information> <terse>`:

```
<interface-information>
   <physical-interface>
     <name>ge-1/0/0</name>
     <admin-status>up</admin-status>
     <oper-status>up</oper-status>
     <logical-interface>
       <name>ge-1/0/0.0</name>
       <admin-status>up</admin-status>
       <oper-status>up</oper-status>
       <address-family>
         <address-family-name>inet</address-family-name>
         <interface-address>
           <ifa-local junos:emit=”emit”>1.1.1.1/24</ifa-local>
         </interface-address>
       </address-family>
       <address-family>
         <address-family-name>iso</address-family-name>
       </address-family>
       <address-family>
         <address-family-name junos:emit=”emit”>mpls</address-family-name>
       </address-family>
     </logical-interface>
   </physical-interface>
   ... repeated for every interface
</interface-information>
```

If a script does not specify a predicate, the location path `$results/physical-interface` returns a node-set with the `<physical-interface>` element nodes for ALL the interfaces in the JUNOS device. But because the script includes the `[name==”ge-1/0/0”]` predicate, the location path is instructed to only include the `<physical-interface>` element nodes in the returned node-set if they have a child named `<name>` whose value is “`ge-1/0/0`”. As a result, the location path only returns a single <physical-interface> element node: the `ge-1/0/0` interface.Multiple predicates can be included within a location path. When more than one predicate is present they are read from left to right. The expressions within all of the predicates must evaluate to true or the currently compared node is not included in the location path result.

This op script shows how predicates can be used in location paths to sharpen the results. Processing the script displays the admin status of only a single interface on the console:

```
/* show-admin-status.slax */
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";

/* This is imported into the JUNOS CLI help text */
var $arguments =  {
            <argument> {
              <name> "interface";
              <description> "Show admin status of interface";
            }
          }

/* Command-line argument */
param $interface;

match / {
  <op-script-results> {

    /* JUNOS XML API Element for show interface terse */
    var $get-interface-rpc = <get-interface-information> {
                <terse>;
              }

    /* Retrieve the results of the API request */
    var $results = jcs:invoke( $get-interface-rpc );

    /* Assign all matching nodes from the results to the $admin-status variable */
    var $admin-status = $results/physical-interface[name==$interface]/admin-status;

    /* Output the interface admin status to the console */
    <output> "The admin status of " _ $interface _ " is " _ $admin-status;
  }
}
```

The op script allows the interface to be chosen through a command-line argument. Then, after retrieving the `<get-interface-information>` output from the `jcs:invoke()` function, the admin status of only the selected interface is extracted through a location path with a predicate included. The output of the script shows:

```
user@JUNOS> op show-admin-status interface ge-1/0/0
The admin status of ge-1/0/0 is up
```

Location Path Operators, Abbreviations, and Wildcards

Table 4.2 lists the operators used within location paths.

Table 4.2 Location Path Operators

Name Code	Example Explanation
Location Path Step `/`	`var $host-name = $results/system/host-name;` Each `/` represents one step in the XML hierarchy. The direction of the step depends on the axis in use, with the default being the child axis.
Multiple Steps `//`	`var $host-name = $results//host-name;` The `//` skips over multiple steps in the hierarchy. This example is the same as the previous example but the `<system>` element node is skipped since the `//` operator will search through zero or more steps. If there were `<host-name>` element nodes in other hierarchy levels they would be returned by this location path as well.
Parent Axis `..`	`var $errors = $results/..//xnm:error;` The `..` is an abbreviation for the parent axis. It indicates that the parent axis should be searched instead of the default child axis.
Attribute Axis `@`	`var $changed = $configuration//@changed;` The `@` sign is an abbreviation for the attribute axis, indicating that JUNOS should search the attribute axis instead of the default child axis.
Wildcard Match `*`	`var $user-children = $configuration/system/login/user/*;` The wildcard matches all nodes along the given axis (by default the child axis). In this example it would match all of the child nodes for all `<user>` element nodes.
Predicates `[ ]`	`var $ge-interface = $configuration/interfaces/interface[ starts-with(name, “ge” ) ];` Predicates are bounded by [ ]. If their expression evaluates to false then the node is not included in the location path result.
Context Node `.`	`var $ge-interface = $configuration/interfaces/interface/name[ starts-with(., “ge”) ];` A period can be used within a predicate to indicate that the expression should use the context node value. In this example the `<name>` element node itself is evaluated by the `starts-with()` function since the `.` is used as an argument.

Try It Yourself: Retrieving Information from JUNOS

Create a script similar to the `show-admin-status.slax` example script above, but instead of the Admin Status report the MTU of a physical interface to the screen. The interface to be displayed should be selected through a command-line argument.

Looping with For-each

It can be necessary to loop through several returned nodes within a node-set as a result of location paths. The `for-each` statement does this. It instructs the script engine to execute a code block for the first node and then loop back through the code block for every node until the node list is exhausted, at which point the script engine continues past the `for-each` statement.

The syntax of a `for-each` loop is similar to an `if` statement but the contents within its parenthesis consist of a location path or node-set variable rather than a conditional expression:

```
for-each( $results/physical-interface/mtu ) {
  /* looped code */
}
```

The `$results/physical-interface/mtu` location path is evaluated and a list of nodes returned. The `for-each` code block is then executed for each node in the node list. Here is a script that displays the mtu of all physical interfaces in the router:

```
/* show-mtu.slax */
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> {

    /* Send JUNOS XML API Element via jcs:invoke */
    var $results = jcs:invoke( “get-interface-information” );

    /* Create node list based on location path, loop through each node */
    for-each( $results/physical-interface/mtu ) {

      /* Output the MTU for all interfaces that don’t have Unlimited MTU */
      if( . != “Unlimited” ) {
        <output> “The MTU for “ _ ../name _ “ is “ _ .;
      }
    }
  }
}
```

After the `<get-interface-information>` API Element is used to retrieve the desired information, a location path based on the returned XML data is provided to the `for-each` statement. This location path provides a result of all the `<mtu>` element nodes that are children of a `<physical-interface>` element node. In other words, all the `<mtu>` nodes of physical interfaces are returned.

This node list is submitted to the `for-each` loop and each time through the loop the mtu for the interface is output to the screen. An `if` statement specifies that interfaces without a typical MTU (their MTU is listed as "Unlimited") are not shown.

NOTE This same behavior is enforced through a location path predicate instead: `$results/physical-interface[ mtu != "Unlimited" ]`

The code example uses the `.` abbreviation to refer to the context node. Each time through the loop the context node changes as a new node is selected from the node list. The conditional expression `. != "Unlimited"` is testing if the current `<mtu>` element node has a value of "Unlimited" or not. When it does, then its MTU is not shown on the console.

With the addition of the `for-each` statement a new reference point, besides variables, is now available for location paths. Each iteration through the loop works with a different context node. This context node is the reference point for any location paths that are not based on a variable. Look again at this line from the script:

```
<output> "The MTU for " _ ../name _ " is " _ .;
```

The `.` at the end of the sentence refers to the current `<mtu>` element node. Concatenating the element node to a string causes its value to be included. Also, note how the `../name` location path is included without any attached variable. Instead, the `../name` location path is compared against the `<mtu>` context node. This causes the path to search first the parent `<physical-interface>` node of the `<mtu>` node and then to select its `<name>` child node. The result is that the text name of the physical interface is included in the output string.

Try It Yourself: Retrieving Information from JUNOS

Create a script that displays the logical interface MTU of all interfaces within your JUNOS device.

Interactive Input

The previous chapter demonstrated how input can be given to the op script through command-line arguments. Beginning in JUNOS 9.4 it is also possible to accept this input interactively within the script itself by using the `jcs:get-input()` function. In JUNOS 9.6 the `jcs:get-secret()` function was also added to prompt for input while hiding the entered answer from the user.

jcs:get-input() / jcs:input()

The `jcs:get-input()` function (known as `jcs:input()` in JUNOS 9.4 and 9.5) causes a prompt to be displayed on the console while the script engine pauses for the user to type a response that is terminated by the enter key. A prompt string is specified as the only argument to the function, the entered answer is returned as a string:

```
var $user-input = jcs:get-input(“Enter your favorite protocol: 
“);
```

NOTE `jcs:input()` requires JUNOS 9.4 or above. Starting in JUNOS 9.6, use `jcs:get-input()` instead.

A good use for `jcs:get-input()` is to gather needed information from missing command-line arguments. As an example, here is a modified script that displays the admin status of an interface. It has a single command-line argument of `interface`. If the command-line argument is missing then the script uses `jcs:get-input()` to learn the interface value:

```
/* show-admin-status.slax */
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”;

/* This is imported into the JUNOS CLI help text */
var $arguments =  {
           <argument> {
             <name> “interface”;
             <description> “Show admin status of interface”;
           }
         }

/* Command-line argument */
param $interface;

match / {
  <op-script-results> {
    /* JUNOS XML API Element for show interface terse */
    var $get-interface-rpc = <get-interface-information> {
                <terse>;
              }

    /* Retrieve API results through jcs:invoke */
    var $results = jcs:invoke( $get-interface-rpc );

    /* Use the command-line argument if provided otherwise ask user through jcs:get-
input() */
    var $interface-value = {
              if( string-length( $interface ) > 0 ) {
                expr $interface;
              }
              else {
                expr jcs:get-input("Enter interface: ");
              }
            }

    /* Locate the matching node and assign to $admin-status */
    var $admin-status = $results/physical-interface[name==$interface-value]/admin-
status;

    /* Output the interface and status to the console */
    <output> "The admin status of " _ $interface-value _ " is " _ $admin-status;
  }
}
```

The difference between the new script and the original version is the addition of the `$interface-value` conditionally-set variable. If the `$interface` parameter has been entered through the command-line (as indicated by having a string-length greater than `0`) then its entered value is used. Otherwise, the `jcs:get-input()` function is invoked with a prompt of "`Enter interface:`". The user types in the interface name and presses enter, after which the inputted string is assigned to the `$interface-value` variable. This new variable is now used by the following lines to correctly retrieve the interface admin status from the `$results` variable and to print the interface name to the console.

```
user@JUNOS> op show-admin-status interface fxp0
The admin status of fxp0 is up

user@JUNOS> op show-admin-status
Enter interface: fxp0
The admin status of fxp0 is up
```

TIP The result tree is only processed after a script is executed, but the `jcs:get-input()` and `jcs:get-secret()` functions run as part of the script processing. This might cause your output to appear in the incorrect order because any use of the `<output>` result tree element is only displayed after the script terminates. An alternative method of writing to the console, which occurs during script processing, is to use the `jcs:output()` function. Here is an example :note that it must always be preceded by the `expr` statement: `expr jcs:output("Hello World!");`

jcs:get-secret()

The `jcs:get-secret()` function works in the same way as `jcs:get-input()` except that the user input is not echoed to the screen. This makes the function ideal when the user must enter sensitive information such as passwords.

ALERT! jcs:get-secret() requires JUNOS 9.6 or later.

Try It Yourself: Interacting with the User

Modify your script displaying the MTU of a single physical interface. Add a check to see if the command-line argument for the interface has been entered. If it has not, then request the information from the user through the jcs:get-input() function.

Writing to the Syslog

JUNOS scripts can write messages to the syslog by using the `jcs:syslog()` function. This requires two arguments. The first is a string that defines the facility and severity at which the message should be logged. The second is the message string to be logged.

A single string expresses the facility and severity with a period inbetween, for example "external.error" or "daemon.info". The available facilities and severities are listed in Tables 4.3 and 4.4.

Table 4.3 Syslog Facilities

Facility String	Description
`auth`	Authorization system
`change`	Configuration change log
`conflict`	Configuration conflict log
`daemon`	Various system processes
`external`	Local external applications
`firewall`	Firewall filtering system
`ftp`	FTP process
`interact`	Commands executed via CLI
`pfe`	Packet forwarding engine
`user`	User processes

Table 4.4 **Syslog Severity Levels**

Severity	Description
`alert`	Conditions that require immediate correction
`crit`	Critical conditions
`debug`	Debug messages
`emerg / panic`	Panic conditions
`err /error`	Error conditions
`info`	Informational messages
`notice`	Non-error conditions that require special handling
`warn / warning`	Warning messages

When including the `jcs:syslog()` function in a script it must be preceded by the `expr` statement. While there is no result returned to write to the result tree, SLAX requires that scripts try to do something with function results, even if they are always blank:

```
expr jcs:syslog("external.warn", "The script changed the configuration" );
```

The above code would result in the following being logged to the syslog:

```
May 21 21:45:13 JUNOS cscript: %EXTERNAL-4: The script changed the configuration
```

Syslog messages from JUNOS scripts always begin with `cscript:`. The %EXTERNAL-4 is present because explicit-priority has been activated in the configuration for the syslog file. It shows that the message was logged by the correct facility at the desired severity.

Here is an example script that allows any desired syslog message to be logged from the CLI. The facility, severity, and message are all entered through command-line arguments:

```
/* log-message.slax */
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»;

/* This is imported into the JUNOS CLI help text */
var $arguments = {
  <argument> {
    <name> «facility»;
    <description> «Facility for the syslog message»;
  }
  <argument> {
    <name> «severity»;
    <description> «Severity level of the syslog message»;
  }
  <argument> {
    <name> «message»;
    <description> «Message to send to syslog»;
  }
}

/* Command-line arguments */
param $facility;
param $severity;
param $message;

match / {
  <op-script-results> {

/* Assemble the facility-severity string */
    var $facility-severity = $facility _ «.» _ $severity;

/* Log message to syslog */
    expr jcs:syslog( $facility-severity, $message );
  }
}
```

If the log-message.slax script is run with the following arguments:

```
user@JUNOS> op log-message facility user severity notice message "Test Message"
```

Then it logs this to the syslog (with explicit-priority set):

```
May 21 21:50:18 JUNOS cscript: %USER-5: Test Message
```

NOTE `jcs:syslog()` requires JUNOS 9.0 or later.

Try It Yourself: Writing to the Syslog

Create an op script that logs the user name, script, product, and hostname to the syslog from the user facility with a severity level of info.

Reading the Configuration

JUNOS scripts can retrieve the current configuration by sending the `<get-configuration>` API Element to `jcs:invoke()`:

```
var $configuration = jcs:invoke( "get-configuration" );
```

There are a number of attributes available for `<get-configuration>`, the most useful of which is `database`. It can be set to either `committed` or `candidate`, and it indicates which configuration database should be returned. The candidate database is returned by default if the `database` attribute is missing.

```
var $rpc = <get-configuration database=”committed”>;
var $committed-configuration = jcs:invoke( $rpc );
```

MORE? Additional attributes is used with the `<get-configuration>` API Element. Consult the *JUNOScript API Guide* in the JUNOS Software Documentation at www.juniper.net/techpubs/ for more information.

The entire configuration is returned in XML format enclosed within a parent `<configuration>` element. To see the XML structure of the configuration on a JUNOS device, append `| display xml` to the `show configuration` command:

```
user@JUNOS> show configuration | display xml
<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/9.6I0/
junos”>
  <configuration ...attributes were cut... >
      <version>9.6I0 [builder]</version>
      <groups>
        <name>re0</name>
        <system>
          <host-name>JUNOS</host-name>
          <time-zone>America/Los_Angeles</time-zone>
      <snip>
```

The `<get-configuration>` API Element does not have to return the entire configuration. When the script requires only portions of the configuration then specify those hierarchies within `<get-configura-tion>`:

```
var $rpc = <get-configuration database=”committed”> {
        <configuration> {
          <protocols> {
            <bgp>;
          }
          <policy-options>;
        }
      }
```

As shown above, to request a subset of the configuration, enclose the desired hierarchies within a `<configuration>` child element of the `<get-configuration>` API Element. The example above only retrieves the `bgp` and `policy-options` hierarchy levels.

Configuration settings can be retrieved through location paths in the same way as operational results. This example shows how to do it:

```
/* show-name-servers.slax */
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> {

    /* JUNOS XML API Element to retrieve the configuration */
    var $config-rpc =  <get-configuration> {
             <configuration> {
               <system>;
             }
           }

    /* Request configuration and assign to $config variable */
    var $config = jcs:invoke( $config-rpc );

    /* Extract the name-servers from the config and assign to variable */
    var $name-servers = $config/system/name-server;

    /*
     * If no name-servers are present then output message, otherwise
     * output all the name-server names/addresses.
     */
    if( jcs:empty( $name-servers ) ) {
      <output> "There are no name servers defined.";
    }
    else {
      <output> "Here are the name-servers:";
      for-each( $name-servers ) {
        <output> ./name;
      }
    }
  }
}
```

The script begins by requesting the current configuration of the system hierarchy. The `$config` variable has the `<configuration>` element node as its reference point, so the first step of the location path is the main hierarchy level (in the example: `system`). The script then loads all the `<name-server>` element nodes from the system hierarchy into a node-set variable `$name-servers`. What happens next depends on the value of `$name-servers`. The `jcs:empty()` function returns true if its node-set argument is empty, otherwise it will return false. In the script code, if the `$name-servers` variable is empty, so `jcs:empty()` returns true, then the output string will express that there are no name-servers. Otherwise, if name-servers are present in the configuration then a `for-each` statement will loop through every element node within the `$name-servers` variable and output the address of the name-server. Here is the output:

```
user@JUNOS> op show-name-servers
Here are the name-servers:
192.168.16.10
192.168.35.10
```

This second example looks at the `routing-options` hierarchy level. It checks to see if an autonomous-system number has been configured and reports if one is present or not. It also reports all static routes that have been configured:

```
/* show-routing-options.slax */
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> {

    /* API Element to retrieve the routing-options configuration in XML */
    var $config-rpc =  <get-configuration> {
            <configuration> {
              <routing-options>;
            }
          }

    /* Send API Element to JUNOS via jcs:invoke and retrieve config in XML format */
    var $config = jcs:invoke( $config-rpc );

    /* Extract the ASN from the configuration using a location path */
    var $asn = $config/routing-options/autonomous-system/as-number;

    /* If the ASN is defined then output it to the console */
    if( jcs:empty( $asn ) ) {
      <output> “There is no ASN defined.”;
    }
    else {
      <output> “The ASN is: “ _ $asn;
    }

    /* Extract all the static routes from the configuration using a location path */
    var $static-routes = $config/routing-options/static/route;

    /* If there are static routes present then loop through and display them */
    if( jcs:empty( $static-routes ) ) {
      <output> “There are no static routes.”;
    }
    else {
      for-each( $static-routes ) {
        <output> “There is a static route to “ _ name _ “ with next-hop “ _ next-hop;
      }
    }
  }
```

If the `routing-options` configuration looks like this:

```
<configuration>
    <routing-options>
      <static>
        <route>
          <name>10.1.0.0/16</name>
          <next-hop>192.168.1.1</next-hop>
        </route>
        <route>
          <name>10.2.0.0/16</name>
          <next-hop>192.168.1.1</next-hop>
        </route>
      </static>
      <autonomous-system>
        <as-number>65500</as-number>
      </autonomous-system>
    </routing-options>
</configuration>
```

Then this will be the output from the script:

```
user@JUNOS> op show-routing-options
The ASN is: 65500
There is a static route to 10.1.0.0/16 with next-hop 192.168.1.1
There is a static route to 10.2.0.0/16 with next-hop 192.168.1.1
```

Try It Yourself: Reading the JUNOS Configuration

Create an op script that reads the configuration and outputs all the syslog file names to the console.

Changing the Configuration

JUNOS scripts are capable of changing, as well as reading, the configuration. When used in this manner op scripts can perform structured configuration changes, allowing users with less JUNOS familiarity to safely alter the configuration. As an example, consider a provisioning script that could ask relevant questions for a new connection and then automatically commit the needed configuration. This decreases the complexity in adding new customers even when sophisticated policies are in place, and it guarantees that all customers are added to the configuration using the established guidelines.

To change the configuration use the `jcs:load-configuration` template. This template is included in the junos.xsl default import file and is available for use by all op scripts (in JUNOS 9.3 and later). Before using `jcs:load-configuration` a connection must be opened. This is needed because multiple steps must occur for a configuration change to be successful:

1. The configuration database is locked.
2. The configuration changes are loaded.
3. The configuration is committed.
4. The configuration database is unlocked.

All of these actions must be performed in a sequence with no interference from other users or scripts. JUNOS uses connections to ensure that the steps are performed together rather than in isolation. These connections are first created by the `jcs:open()` function, which returns a connection identifier. When the script is ready to make a configuration change it passes this connection to the `jcs:load-configuration` template. After the changes have been completed, the connection should be closed by using the `jcs:close()` function.

NOTE The `jcs:open()` and `jcs:close()` functions and the `jcs:load-configuration` template are only available in JUNOS 9.3 and beyond.

Configuration changes made with the `jcs:load-configuration` template are made in exclusive configuration mode. If any users are currently editing the configuration or if the database is currently locked then the attempted changes fail. The `jcs:load-configuration` template first locks the database; it then loads the configuration change, performs a commit, and unlocks the database.

The script must provide the connection, and the configuration changes the `jcs:load-configuration` template through its parameters. The change is expressed in SLAX abbreviated XML format and is enclosed within a `<configuration>` element. The entire hierarchy level must be included for the change, which is merged into the existing configuration:

```
<configuration> {
  <routing-options> {
    <static> {
      <route> {
        <name> "10.3.0.0/16";
        <next-hop> "192.168.1.1";
      }
    }
  }
}
```

The above XML structure could be used to add a new static route for 10.3.0.0/16.

The template parameters used by `jcs:load-configuration` are `$connection` for the connection and `$configuration` for the configuration change. An example of the steps needed to make a configuration change can be seen in the following:

```
/* The configuration change must be defined */
var $configuration-change = <configuration> {
             <routing-options> {
               <static> {
                  <route> {
                    <name> “10.3.0.0/16”;
                    <next-hop> “192.168.1.1”;
                  }
               }
             }
           }

/* A connection must be opened */
var $connection = jcs:open();

/*
 * The connection and change are set as parameters to the jcs:load-configuration
 * template which performs the change. The := operator is used to ensure that the
 * $results variable is a node-set rather than a result tree fragment.
 */
var $results := { call jcs:load-configuration( $connection, $configuration =
$configuration-change ); }

/* Check for errors – report them if they occurred */
if( $results//xnm:error ) {
  for-each( $results//xnm:error ) {
    <output> message;
  }
}

/* The connection is closed */
var $close-results = jcs:close($connection);
```

NOTE The location path used to check for errors from templates like `jcs:load-configuration` is different than the location path for errors from functions like `jcs:invoke()`. With `jcs:load-configuration` the location path is: “`$results//xnm:error`”. With `jcs:invoke()` the location path is “`$results/..//xnm:error`”. (Both of these examples assume that the variable name used is `$results`.)

MORE? Find more examples of op scripts – including a script for configuring static routes, a script that extracts a policy chain, a script that lets user self-serve their local account by changing their password, and a script to change the default behavior of standard operational mode commands – in the Appendix included in the PDF version of this booklet.

What to Do Next & Where to Go …

http://www.juniper.net/dayone

The PDF version of this booklet contains a supplemental Appendix with more information and sample answers to the *Try It Yourself* sections. Check to see if the other booklets in this series are availabe here as well.

http://www.juniper.net/scriptlibrary/

The JUNOS Script Library is 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/us/en/training/elearning/junos_scripting.html

This course teaches you how use JUNOS automation scripting to reduce network downtime, reduce configuration complexity, automate common tasks, and decrease the time to problem resolution.

http://www.juniper.net/techpubs

All Juniper-developed product documentation is freely accessible at this site. The JUNOS API and Scripting Documentation can be found within.

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 Software. AIS provides a comprehensive set of tools and technologies designed to enable the automated delivery of tailored, proactive network intelligence and support services.

Appendix

Supplemental JUNOS Automation Information

This *Applying JUNOS Automation* booklet shows the potential that JUNOS automation offers to enhance the availability and efficiency of a network. It explained the foundation of the SLAX scripting language, as well as the mechanisms used to communicate with JUNOS and to affect the configuration of the JUNOS device.

This Appendix supplements the information previously discussed by providing five additional op scripts as well as example solutions to the *Try It Yourself* sections.

Op Script Examples

This first section of the Appendix provides five additional scripts, which highlight the possibilities that op scripts provide and makes use of the lessons learned in this volume. Extensive comments are included within the scripts to provide documentation on their structure.

Display Time

In this first example, the `display-time` template alters the earlier example of Chapter 3 by providing an easier way for the user to select the ISO format or the normal format.

In the prior example the `desired-format` command-line argument was added, allowing the user to choose what format in which to display the time. When the `display-time` template is called the `$format` parameter is set to the value of the `$desired-format` global parameter.

There is an easier way to write the script to let the user choose the format. Remember that global parameters and variables are accessible in the entire script, not just in the main template. This means that it is not necessary to pass the `$format` value to the `display-time` template as a template parameter. Instead, the `display-time` template can simply use the global parameter.

The script operates in the same manner as the script in Chapter 3, but in this case the named template accesses the global parameter instead of relying on the main template to pass the format as a template parameter:

```
/* show-time.slax */
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";

/* This is imported into the JUNOS CLI help text */
var $arguments = {
  <argument> {
    <name> "display-format";
    <description> "Choose either iso or normal";
  }
}

/* Command-line argument */
param $display-format;

match / {
  <op-script-results> {

    /* Call the display-time template */
    call display-time;

  }
}

/* Output the localtime to the console in either iso (default) or normal format */
template display-time {
  if( $display-format == "iso" ) {
    <output> "The iso time is " _ $localtime_iso;
  }
  else {
    <output> "The time is " _ $localtime;
  }
}
```

The output is the same as provided by the example of Chapter 3:

```
user@JUNOS> op show-time display-format iso
The iso time is 2009-05-12 21:01:10 PDT

user@JUNOS> op show-time display-format normal
The time is Tue May 12 21:01:13 2009
```

Static route

The next script is used to add a static route. First the `jcs:get-input()` function is used to learn the static route and then it is used again to learn the next-hop. Once these two values are known the script instructs configuration changes, then loads and commits the change:

```
/* add-route.slax */
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> {

    /* Ask for the static route */
    var $static-route = jcs:get-input(«Enter the static route: «);

    /* Ask for the next-hop */
    var $next-hop = jcs:get-input(«Enter the next-hop: «);

    /* Create the configuration change */
    var $configuration = <configuration> {
                 <routing-options> {
                   <static> {
                     <route> {
                       <name> $static-route;
                       <next-hop> $next-hop;
                     }
                   }
                 }
               }

    /* Open a connection */
    var $connection = jcs:open();

    /* Call jcs:load-configuration and provide the connection and configuration
     * change to make.
     */
    var $results := { call jcs:load-configuration( $connection, $configuration ); }

    /* Check for errors - report them if they occurred */
    if( $results//xnm:error ) {
      for-each( $results//xnm:error ) {
        <output> message;
      }
    }

    /* If there are no errors then report success */
    if( jcs:empty( $results//xnm:error ) ) {
      <output> «Committed without errors.»;
    }

    /* Close the connection */
    var $close-results = jcs:close($connection);
  }
}
```

Here is an example of the `add-route op` script in operation:

```
user@JUNOS> **op add-route**
Enter the static route: 10.6.0.0/16
Enter the next-hop: 192.168.1.4
Committed without errors.
```

And here is the configuration after the scripted change (two routes were already present):

```
user@JUNOS> show configuration routing-options
static {
  route 10.1.0.0/16 next-hop 192.168.1.1;
  route 10.2.0.0/16 next-hop 192.168.1.1;
  route 10.6.0.0/16 next-hop 192.168.1.4;
}
autonomous-system 65500;
```

show-bgp-policy

This is an example of a custom show command that was created to enhance JUNOS operation. The goal of this script is to provide an easy way to peruse the complete policy chain of a BGP peer. With multiple BGP policies in an import or export policy chain, it can become troublesome to determine exactly when or if a prefix is accepted/rejected as the policies typically do not appear in the desired order within the `policy-options` configuration.

This script extracts the policy chain for the selected peer in the import or export direction. It then retrieves the configuration text for each policy, and displays it on the console in sequential order. Now a user only has to execute the op script and he can then read the complete policy chain from start to finish.

This shows the output of the script when asked to report the import policies for peer 10.0.0.1:

```
user@JUNOS> op show-bgp-policy neighbor 10.0.0.1 direction import
BGP Neighbor: 10.0.0.1 in group EBGP
Import Policies: block-private set-local-pref accept-by-community
Policy: block-private
  policy-statement block-private {
    from {
      route-filter 192.168.0.0/16 orlonger;
      route-filter 10.0.0.0/8 orlonger;
      route-filter 172.16.0.0/12 orlonger;
    }
    then reject;
  }
Policy: set-local-pref
  policy-statement set-local-pref {
    term default {
      then {
        local-preference 50;
      }
    }
    term pref-75 {
      from community pref-75;
      then {
        local-preference 75;
      }
    }
    term pref-100 {
      from community pref-100;
      then {
        local-preference 100;
      }
    }
  }
Policy: accept-by-community
  policy-statement accept-by-community {
    term accept {
      from community from-64500;
      then accept;
    }
    term reject {
      then reject;
    }
  }
```

An additional feature of this script is that a user can use the `database` command-line argument to select between the *committed* configuration and the *candidate* configuration. The committed configuration is used by default. However, the option to view the policy chain in the candidate configuration is useful to verify the BGP policy configuration prior to committing any policy changes.

The script code for `show-bgp-policy.slax` follows:

```
/*
 * This op script displays the full chain of policy configuration for a given
 * neighbor and import/export direction. Either the candidate or the committed
 * database can be used, with the committed database used by default.
 *
 * Usage: op show-bgp-policy neighbor 10.0.0.1 direction import
 *
 */
```

```
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”;

/*
 * The $arguments global variable is a special variable which JUNOS reads to build
 * the CLI help for the script. The command-line arguments will appear within the help
 * message along with their description if the following format is followed.
 */
var $arguments = {
  <argument> {
    <name> “neighbor”;
    <description> “Required: The BGP neighbor address”;
  }
  <argument> {
    <name> “direction”;
    <description> “Required: Policy direction, either import or export”;
  }
  <argument> {
    <name> “database”;
    <description> “Optional: Specify either the [committed] or candidate configuration
database”;
  }
}

/* These global parameters are assigned based on their corresponding command-line
arguments */
param $neighbor;
param $direction;
param $database=”committed”;

match / {
  <op-script-results> {

    /*
     * The script first sanity checks the $neighbor and $direction command-line
arguments. If they
     * have not been set correctly then the script exits by using the <xsl:message>
command element
     * and specifying that the script should end by setting the terminate attribute to
“yes”. Note
     * that <xsl:message> is not a result tree element, it is one of the few elements
that is
     * processed immediately rather than written to the result tree.
     */
    if( jcs:empty( $neighbor ) or jcs:empty( $direction ) or
      ( $direction != “import” and $direction != “export” ) ) {
      <xsl:message terminate=”yes”> “The neighbor address and policy direction must be 
specified.”;
    }

    /*
     * The database should be set to either “committed” or “candidate”, if not then exit 
the script
     * with an error
     */
    if( $database != “committed” and $database != “candidate” ) {
      <xsl:message terminate=”yes”> “The database is not set correctly.”;
    }

    /*
     * This API element is used to retrieve the configuration. Either the candidate or 
the committed
     * configuration can be used. The choice is based on the $database command-line 
argument. In
     * addition the inherit attribute has been set. This is used to request that the 
configuration be
     * retrieved in inherited mode meaning that all configuration from configuration 
groups will be
     * inherited into its proper hierarchy level. This is done so that the script has an 
accurate view
     * of the current BGP policy configuration.
     */
    var $get-bgp-rpc = <get-configuration database=$database inherit=”inherit”> {
            <configuration> {
              <protocols> {
                <bgp>;
              }
            }
          }

    /*
     * The assembled API element is sent to JUNOS through jcs:invoke and the XML 
response is stored in
     * $bgp-config
     */
    var $bgp-config = jcs:invoke( $get-bgp-rpc );

    /*
     * The BGP neighbor is extracted from the configuration through a location path. The 
last()
     * function is used to guarantee that only one element node will be returned. It 
returns true
     * only if the node is the last in the node list so only one node can be selected and 
assigned to
     * $bgp-neighbor.
     */
```

```
   var $bgp-neighbor = $bgp-config/protocols/bgp//neighbor[name == $neighbor
][last()];

   /*
    * Error check, if the $bgp-neighbor is missing than jcs:empty() will return true
and the script
    * will be terminated with an error message.
    */
   if( jcs:empty( $bgp-neighbor ) ) {
     <xsl:message terminate="yes"> "BGP Neighbor " _ $neighbor _ " isn't configured.";
   }

   /*
    * Begin the output. The BGP neighbor will be shown first as well as the BGP group it
is in.
    */
   <output> "BGP Neighbor: " _ $neighbor _ " in group " _ $bgp-neighbor/../name;

   /*
    * The BGP policy list will now be retrieved. To do this the jcs:first-of() function
is used.
    * This is necessary because a BGP peer's policy can be configured at up to three
different
    * hierarchy levels: the neighbor level, the group level, or the bgp level. The peer
uses the
    * policy configuration at the most specific level. jcs:first-of() works by checking
multiple
    * node-set arguments. The first node-set that is not empty will be returned. So in
this case
    * three separate location paths are provided (each of which results in a node-set).
The first
    * location path refers to any policies at the neighbor level, the second pulls
policies at the
    * group level, and the last pulls policies at the bgp level. The most specific
location that has
    * policies will be returned and assigned to the $policy-list variable.
    */
   var $policy-list = jcs:first-of( $bgp-neighbor/*[name()==$direction],
                $bgp-neighbor/../*[name()==$direction],
                $bgp-neighbor/../../*[name()==$direction] );

   /*
    * Error check, if there are no policies then the script can terminate.
    */
   if( jcs:empty( $policy-list ) ) {
     <xsl:message terminate="yes"> "There are no " _ $direction _ " policies for " _
$neighbor;
   }

   /*
    * The policy chain is now output to the console. This is done all within one line by
```

```
writing
     * the text strings to a single <output> element through the expr statement.
     */
    <output> {
      if( $direction == “import” ) {
        expr “Import Policies:”;
      }
      else {
        expr “Export Policies:”;
      }
      for-each( $policy-list ) {
        expr “ “ _ .;
      }
    }

    /* A for-each will now loop through each policy individually to allow them to be
displayed */
    for-each( $policy-list ) {

      /* Output the policy name */
      <output> “\nPolicy: “ _ .;

      /*
       * The script must retrieve the text version of each policy one by one. By default the
       * returned configuration is always in XML format. To see the configuration in text
format
       * use the format attribute and set it to “text”.
       */
      var $get-policy-rpc = <get-configuration format=”text” database=$database
inherit=”inherit”> {
                   <configuration> {
                     <policy-options> {
                       <policy-statement> {
                         <name> .;
                       }
                     }
                   }
                 }

      /* Send assembled API element to JUNOS through jcs:invoke(); */
      var $policy-text = jcs:invoke( $get-policy-rpc );

      /*
       * The returned configuration will include the entire hierarchy including the policy-
options
       * statement and enclosing brackets. This is extra clutter that is not needed so it is
removed
       * by using the substring-after and substring functions to remove all the unnecessary
       * characters. The complete text policy is then output to the console.
       */
      var $cropped-text =substring-after( $policy-text, "policy-options {" );
      <output> substring( $cropped-text, 1, string-length( $cropped-text )-2 );
    }
  }
}
```

change-password

This example shows a script that performs an automated configuration change. This script allows a user to self-serve their local account by changing their password from the command-line. The minimum version required is JUNOS 9.6 as the `jcs:get-secret()` function is used to query the user for his new password.

Here is an example of the output of the script:

```
user@JUNOS> op change-password
Enter the new password:
Reenter the new password:
Password changed.
```

Here is the script code for `change-password.slax`:

```
/*
 * This op script changes the password for the current user. The password is
 * learned using jcs:get-secret() for security purposes so the minimum JUNOS version
 * is 9.6
 */
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> {

    /*
     * Query the user for the new password. Use jcs:get-secret() so that the password
     * is not shown on the screen. Ask twice and compare the entries. Terminate the
     * script if they are not the same.
     */
    var $new-password = jcs:get-secret(«Enter the new password: «);
    var $new-password2 = jcs:get-secret(«Reenter the new password: «);

    if( $new-password != $new-password2 ) {
      <xsl:message terminate=»yes»> «The passwords do not match.»;
    }
    if( string-length( $new-password ) == 0 ) {
      <xsl:message terminate=»yes»> «The password is blank.»;
    }

    /*
     * Assemble the configuration change needed to change the password. Pull in the
     * user name from the $user default global parameter.
     */
     var $configuration = {
      <configuration> {
        <system> {
          <login> {
            <user> {
              <name> $user;
              <authentication> {
                <plain-text-password-value> $new-password;
              }
            }
          }
        }
      }
    }

    /* Open a connection */
    var $connection = jcs:open();

    /*
     * Send the configuration change and connection to jcs:load-configuration, it will 
load
     * the change and commit it.
     */
     var $result := { call jcs:load-configuration($connection, $configuration); }

     /* Check for commit errors - report them if found */
     if( $result//xnm:error ) {
      <output> jcs:output(«Error changing the password»);
      for-each( $result//xnm:error ) {
        <output> message;
      }
    }
    else {
      <output> «Password changed.»;
    }

    /* Close the connection */
    var $close-results = jcs:close( $connection );
  }
}
```

safe-bgp-clear

This last example shows how scripts can wrap around standard operational mode commands to alter their default behavior. When users enter `clear bgp neighbor` then *all* BGP peering sessions are restarted. This script makes the command safer by requiring users to confirm that they really do want to restart all their sessions. The minimum version required is JUNOS 9.6 because the `jcs:get-input()` function is used to perform confirmation.

Here is an example of the output of the script:

```
user@JUNOS> op safe-bgp-clear peer all
This will clear ALL BGP sessions
Are you sure? (yes/[no]): yes
Cleared 2 connections
```

Here is the script code for `safe-bgp-clear.slax`:

```
/*
 * This script provides a safe version of the “clear bgp neighbor” command. That command
 * allows an operator to accidently clear all BGP neighbors when no address is
specified.
 * This script requires “peer all” to be specified in order to clear all BGP neighbors,
 * and it requires confirmation from the operator that they do indeed wish to clear all
 * of their BGP peers.
 *
 * Minimum JUNOS version is 9.6 due to the jcs:get-input() function. (This can be
performed
 * in JUNOS 9.4 and 9.5 by using the deprecated jcs:input() function.)
 */

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”;

/*
 * The $arguments global variable is a special variable which JUNOS reads to build
 * the CLI help for the script. The command-line arguments will appear within the help
 * along with their description as long as the following format is followed.
 */
var $arguments = {
  <argument> {
    <name> “peer”;
    <description> “Neighbor address to drop or ‘all’ for all sessions”;
  }
}
```

```
/* This global parameter will have its value set based on the matching command-line
argument */
param $peer;

match / {
  <op-script-results> {

    /*
     * The script requires that a peer be specified. If the user did not enter a peer on
the
     * command-line then display an error and exit.
     */
    if( string-length( $peer ) == 0 ) {
      <xsl:message terminate=”yes”> “You must specify which BGP peer you want to
clear.”;
    }

    /*
     * If peer ‘all’ was entered on the command-line then the user will need to confirm
the choice
     * before clearing all the BGP peers.
     */
    if( $peer == “all” ) {

      /*
       * Request confirmation from user before clearing all the BGP peers. In JUNOS 9.4 &
9.5
       * jcs:input() can be used instead.
       */
      var $prompt = “This will clear ALL BGP sessions\nAre you sure? (yes/[no]): “;
      var $response = jcs:get-input( $prompt );

      if( $response == “yes” ) {
                  /*
         * The user confirmed they wish to clear all sessions so call the execute-command
         * template.
         */
        call execute-command();
      }
      else {
        /*
         * If they decided not to clear all the sessions then display that choice to the
         * console.
         */
        <output> “Clear all cancelled”;
      }
    }
    else {
      /* There is no need to confirm the clearing of a specific peer, just execute the
clear
       * command by calling the execute-command template.
       */
      call execute-command();
    }
  }
}

/*
 * This template will invoke the clear bgp neighbor command and display any output to
the console.
 */
template execute-command() {

  /* There is no mapped API Element for clear bgp neighbor so the <command> element will
be used */
  var $command = {
    /* If all is specified then just do the normal command, otherwise include the peer
address */
    if( $peer == “all” ) {
      <command> “clear bgp neighbor”;
    }
    else {
      <command> “clear bgp neighbor “ _ $peer;
    }
  }

  /* Execute the command and retrieve the results */
  var $results = jcs:invoke( $command );

  /* Copy output to the result tree so that it will be displayed on the console */
  copy-of $results;
}
```

Try It Yourself Sample Solutions

This section of the Appendix provides sample solutions for each of the *Try It Yourself* sections in Chapters 1 through 4.

Chapter 1: Try It Yourself: Viewing JUNOS Configuration in XML

Show the following configuration hierarchy levels in XML on a JUNOS device:

```
(e.g. show configuration system | display xml)

[system]
[interfaces]
[protocols]
```

```
user@JUNOS> show configuration system | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.0R4/junos">
  <configuration junos:commit-seconds="1248125252" junos:commit-localtime="2009-07-20
14:27:32 PDT" junos:commit-user="user">
      <system>
        <host-name>JUNOS</host-name>
        <login>
          <user>
            <name>user</name>
            <authentication>
              <encrypted-password>pVFST7cOl4Hu2</encrypted-password>
            </authentication>
          </user>
        </login>
      </system>
  </configuration>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>

user@JUNOS> show configuration interfaces | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/9.0R4/junos">
  <configuration junos:commit-seconds="1248125420" junos:commit-localtime="2009-07-20
14:30:20 PDT" junos:commit-user="user">
      <interfaces>
        <interface>
          <name>lo0</name>
          <unit>
            <name>0</name>
            <family>
```

```
              <inet>
                <address>
                  <name>10.3.3.3/32</name>
                </address>
              </inet>
              <iso>
                <address>
                  <name>47.0000.3333.3333.3333.00</name>
                </address>
              </iso>
            </family>
          </unit>
        </interface>
      </interfaces>
  </configuration>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>

user@JUNOS> **show configuration protocols | display xml**
<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/9.0R4/junos”>
  <configuration junos:commit-seconds=”1248125504” junos:commit-localtime=”2009-07-20
14:31:44 PDT” junos:commit-user=”user”>
      <protocols>
        <ospf>
          <area>
            <name>0.0.0.0</name>
            <interface>
              <name>ge-1/0/0.0</name>
            </interface>
          </area>
        </ospf>
        <pim>
          <interface>
            <name>ge-1/0/0.0</name>
          </interface>
        </pim>
      </protocols>
  </configuration>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

Chapter 1:
Try It Yourself: Writing XML in the SLAX Abbreviated Format

Rewrite the following configuration using the SLAX abbreviated XML format:

```
system {
  host-name r1;
  login {
     message “Unauthorized access prohibited.”;
  }
}
```

```
<system> {
  <host-name> “r1”;
  <login> {
    <message> “Unauthorized access prohibited.”;
  }
}
```

Chapter 2:
Try It Yourself: Adding Comments to the Hello World Script

Make the following modifications to the Hello World script:

1. Add a multi-line comment at the beginning that describes the purpose of the script.
2. Add an additional comment before the `<output> “Hello World!”`; line which states that the script is writing to the console.

After the two modifications have been made, replace the prior version of hello-world.slax on your JUNOS device with the new version. Execute the script again and verify that the new comments did not change the operation.

```
/*
 * hello-world.slax - This script will output “Hello World!” to the console.
 */
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> {
    /* This writes the string to the console. */
    <output> “Hello World!”;
  }
}
```

Chapter 2:
Try It Yourself: Adding Additional Output to the Hello World Script

Modify the Hello World script once again. This time, add two additional lines of output to the console above the `Hello World!` string.

Replace the prior version of hello-world.slax on your JUNOS device with the changed version. Execute the script again and see the affect the new <output> elements have on the script output.

```
/*
 * hello-world.slax – This script will output “Hello World!” to the console.
 */
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> {
    /* This writes the string to the console. */
    <output> “I have a message for you.”;
    <output> “Here is the message:”;
    <output> “Hello World!”;
  }
}
```

Chapter 2:
Try It Yourself: Writing Your Own Script Using the Boilerplate

Using the configuration boilerplate, create a new op script that outputs three separate lines of text to the console. Copy this script to your JUNOS device and enable it. Now you can verify it by executing it from the command-line.

```
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> {
    <output> “One”;
    <output> “Two”;
    <output> “Three”;
  }
}
```

Chapter 3:
Try It Yourself: Working with Operators

Create a new script including two variables that are assigned numeric values. Add a third variable and assign it the product of the first two variables. Display the value of the third variable on the console.

```
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 $first-variable = 100;
    var $second-variable = 5;
    var $third-variable = $first-variable * $second-variable;
    <output> “Result: “ _ $third-variable;
  }
}
```

Chapter 3:
Try It Yourself: Working with Command-line Arguments

Create a new script with a command-line argument that accepts a number from the user. Include the `$arguments` global variable so that the command-line argument will be included in the CLI help output. Perform a mathematical operation on the command-line argument and output the result to the console. Execute the op script a few times, with a different number provided on the command-line each time to verify that the result changes according to the entered number.

```
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”;
```

```
var $arguments = {
  <argument> {
    <name> “number”;
    <description> “The number to multiply.”;
  }
}

param $number;

match / {
  <op-script-results> {
    <output> “Result: “ _ $number * 55;
  }
}
```

Chapter 3:
Try It Yourself: Conditionally Assigning Variable Values

Create a new script with a command-line argument which can be set to either + or - signifying the mathematical operation that you wish to perform. Create a variable that is assigned conditionally based on the value of the command-line argument. If the command-line argument is specified as a + then two values should be added together and assigned to the variable. If the command-line argument is specified as a - then subtraction should be performed between the two values and assigned to the variable. Output the result to the console.

```
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»;

var $arguments = {
  <argument> {
    <name> «operator»;
    <description> «Either + or -»;
  }
}

param $operator;

match / {
  <op-script-results> {
    var $first = 31;
    var $second = 14;
    var $conditional = {
      if( $operator == «+» ) {
        expr $first + $second;
      }
      else if( $operator == «-» ) {
        expr $first - $second;
      }
    }
    <output> $conditional;
  }
}
```

Chapter 3:
Try It Yourself: Working with Named Templates

Create a new script that contains a named template. The template should write a string to the result tree. Redirect this into a variable in the calling template and output the variable value to the console.

```
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 $redirected = { call output-string(); }

    <output> $redirected;
  }
}

template output-string() {
  expr “Here is the output string”;
}
```

Chapter 3:
Try It Yourself: Working with Functions

Create a new script with a variable assigned to the value “Juniper Networks”. Output the following to the console on separate lines:

1. The variable value
2. The variable value - right justified in a 20 space field
3. The string length of the variable
4. The substring before the space
5. The string converted entirely into upper-case.

```
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 $variable = “Juniper Networks”;
    <output> $variable;
    <output> jcs:printf( “%20s”, $variable );
    <output> string-length( $variable );
    <output> substring-before( $variable, “ “ );
    <output> translate( $variable, “abcdefghijklmnopqrstuvwxyz”, “ABCDEFGHIJKLMNOPQRST
UVWXYZ” );
  }
}
```

Chapter 4:
Try It Yourself: Invoking JUNOS Operational Commands

Following the example of the clear-statistics op script shown in this section, create an op script that reboots the system. (Hint: The XML API Element needed is `<request-reboot>`).

```
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 $result = jcs:invoke( «request-reboot» );

  }
}
```

Chapter 4:
Try It Yourself: Retrieving Information from JUNOS

Create a script similar to the show-admin-status.slax example script above, but instead of the Admin Status report the MTU of a physical interface to the screen. The interface to be displayed should be selected through a command-line argument.

```
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”;

var $arguments = {
  <argument> {
    <name> “interface”;
    <description> “Show MTU of interface”;
  }
}

param $interface;

match / {
  <op-script-results> {

    var $results = jcs:invoke( “get-interface-information” );

    var $mtu = $results/physical-interface[name==$interface]/mtu;

    /* Output the interface mtu to the console */
    <output> “The mtu of “ _ $interface _ “ is “ _ $mtu;
  }
}
```

Chapter 4:
Try It Yourself: Retrieving Information from JUNOS

Create a script that displays the logical interface MTU of all interfaces within your JUNOS device.

```
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 $results = jcs:invoke( “get-interface-information” );

    for-each( $results/physical-interface/logical-interface/address-family/mtu ) {

      if( . != “Unlimited” ) {
        <output> “The family “ _ ../address-family-name _ “ MTU for “ _ ../../name _ “ is
“ _ .;
      }
    }
  }
}
```

Chapter 4:
Try It Yourself: Interacting with the User

Modify your script that displays the MTU of a single physical interface. Add a check to see if the command-line argument for the interface has been entered. If it has not then request the information from the user through the `jcs:get-input()` function.

```
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”;
```

```
var $arguments = {
  <argument> {
    <name> "interface";
    <description> "Show MTU of interface";
  }
}

param $interface;

match / {
  <op-script-results> {

    var $interface-value = {
      if( string-length( $interface ) > 0 ) {
        expr $interface;
      }
      else {
        expr jcs:get-input( "Enter interface: " );
      }
    }

    var $results = jcs:invoke( "get-interface-information" );

    var $mtu = $results/physical-interface[name==$interface-value]/mtu;

    /* Output the interface mtu to the console */
    <output> "The mtu of " _ $interface-value _ " is " _ $mtu;
  }
}
```

Chapter 4:
Try It Yourself: Writing to the Syslog

Create an op script that logs the user name, script, product, and hostname to the syslog from the user facility with a severity level of info.

```
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 $syslog-message = "User: " _ $user _ " Script: " _ $script _ " Product: " _
```

```
           $product _ " Hostname: " _ $hostname;
    expr jcs:syslog( "user.info", $syslog-message );
  }
}
```

Chapter 4:
Try It Yourself: Reading the JUNOS Configuration

Create an op script that reads the configuration and outputs all the syslog file names to the console.

```
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 $configuration = jcs:invoke( "get-configuration" );

    for-each( $configuration/system/syslog/file ) {
      <output> "Syslog File: " _ name;
    }
  }
}
```