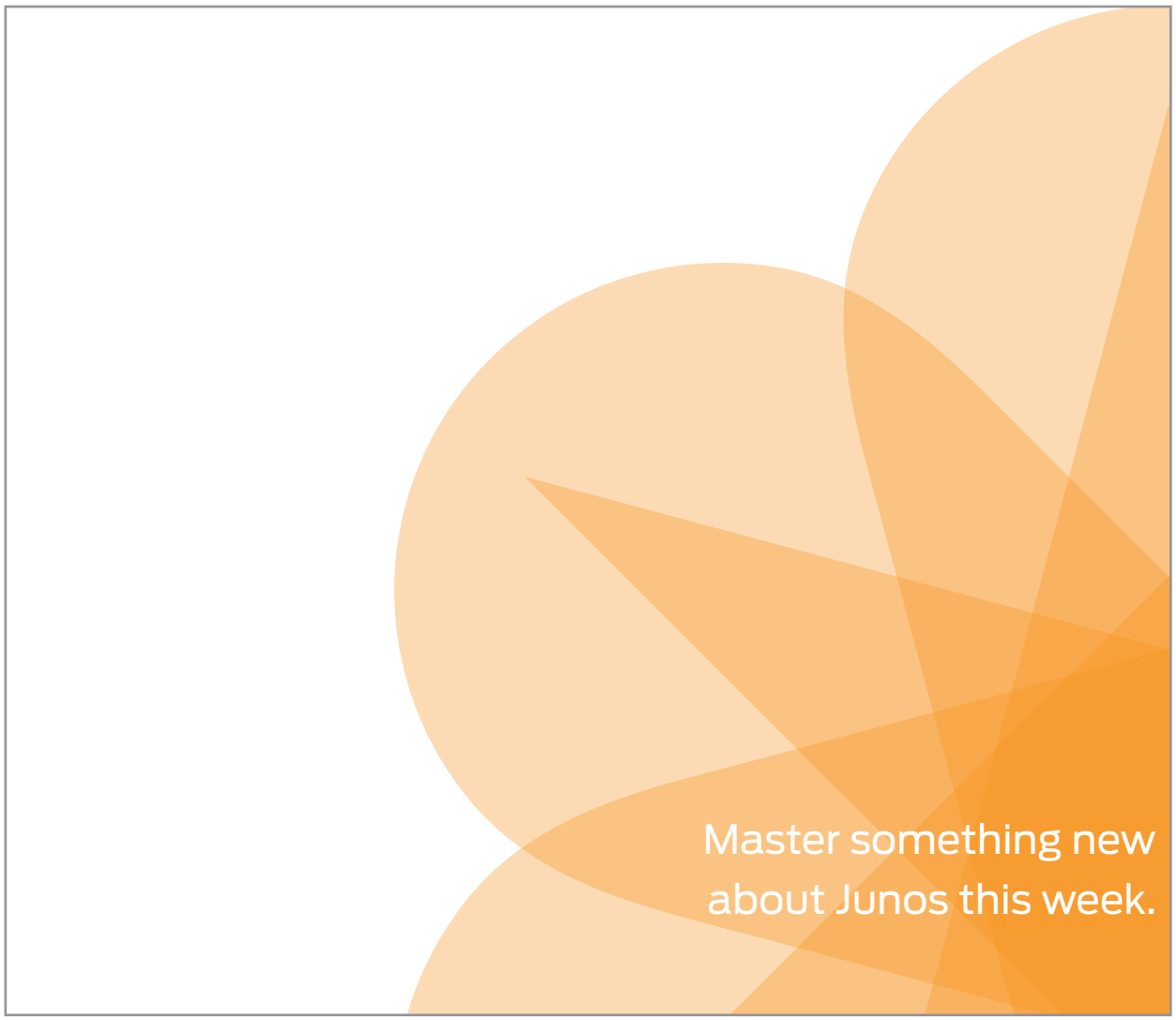
Junos® Automation Series

THIS WEEK: MASTERING JUNOS AUTOMATION PROGRAMMING



By Jeremy Schulman & Curtis Call

THIS WEEK: MASTERING JUNOS AUTOMATION PROGRAMMING

Junos automation scripting, known in the field simply as *Junos scripting*, is a key technology and a fundamental capability that enables you to automate your Junos devices for your own (and unique) operational requirements. You can deploy Junos scripts on any Juniper Networks device that runs the Junos operating system, such as the highly successful MX Mid-Range series (MX5/ 10/40/ 80 routers), the M- and T series of routers, the EX series of Ethernet switches, and the SRX Services Gateways series of network devices. That's a lot of powerful iron.

The most common knowledge shift for many new Junos automation developers is to acquire a good grasp of the XSLT programming paradigm. While many programmers may be familiar with procedural languages such as Perl and Java, the transformation nature and programming framework of XSLT could be new. So *This Week: Mastering Junos Automation Programming* is written from the perspective of a "classical" script programmer, teaching you about the specific tasks and functions of the Junos automation development environment. Spend a week with this book and you'll be able to write, deploy, and debug Junos automation scripts.

"*Mastering Junos Automation Programming* provides all the information you need to quickly harness the power of Junos scripting and automation. This book teaches the reader SLAX, a friendlier, more concise alternative to XSLT, through clear instruction, countless example scripts, and helpful comparisons to other common programming languages and concepts. All the while the authors provide best practices and valuable tips for overcoming common obstacles when scripting with SLAX."

Skyler Bingham, Security Development Engineer, Global Crossing

LEARN SOMETHING NEW ABOUT JUNOS THIS WEEK:

- Interact with Junos using the XML API.
- Use advanced print -formatting and regular-expression processing.
- Understand advanced file storage and "scratch memory" usage.
- Create complex XPath expression techniques for Junos automation.
- Review techniques for integrating Junos automation with your management systems.
- Research advanced event scripting topics.

Published by Juniper Networks Books
www.juniper.net/books

ISBN 978-1936779321

9 781936 779321 52000

07500214

Junos® Automation

This Week: Mastering Junos Automation Programming

By Jeremy Schulman and Curtis Call

JUNIPER NETWORKS

Published by Juniper Networks Books
Writers: Jeremy Schulman, Curtis Call
Editor in Chief: Patrick Ames
Copyediting and Proofing: Nancy Koerbel
Junos Product Management: Cathy Gadecki
J-Net Community Management: Julie Wider

ISBN: 978-1-936779-32-1 (print)
Printed in the USA by Vervante Corporation.
ISBN: 978-1-936779-33-8 (ebook)

Version History: v2 October 2011
3 4 5 6 7 8 9 10 #7500214

About the Authors

Jeremy Schulman is a Senior Systems Engineer at Juniper Networks who brings over 15 years of software engineering experience to the company. Jeremy immediately recognized the vast potential in using Junos automation technologies to help Juniper customers lower cost, reduce risk, and ultimately deliver improved services to their end-customers. Jeremy has created innovative demonstrations and solutions for a wide range of service provider and enterprise solutions, and continues to be a driving force in the Junos Automation community.

Curtis Call is a Senior Systems Engineer at Juniper Networks, has over a decade of experience working with Junos, and has authored multiple books on Junos on-box automation. He is a Juniper Networks Certified Internet Expert (JNCIE-M #43).

Authors' Acknowledgments

Jeremy Schulman would like to acknowledge Phil Shafer and Curtis Call for their inspiration, mentoring, and continuous dedication to Junos automation efforts. Phil, a Distinguished Engineer and the "father" of Junos automation, has in sharing his insights into human interface automation has unlocked incredible potential for using Junos automation well beyond common applications. Curtis is a pillar of the Junos automation community. His highly successful set of Junos automation books have been instrumental in enabling a new generation of Juniper customers to realize the potential of this key differentiating technology. I would also like to thank Patrick for his editorial mentoring and dedication. I greatly appreciate his tireless behind the scenes efforts to make this book a reality. I am extremely fortunate to be able to work with Phil, Curtis, and Patrick and to contribute to the Junos automation community through writing this book.

Curtis Call would like to acknowledge Jeremy for the great work he put into this book and his diligence in seeing the project through to the end, and would also like to acknowledge Patrick for the large contributions he made in shaping this book. At times, editors have such a big impact that they are almost an additional co-author, and this is one of those times.

This book is available in a variety of formats at: www.juniper.net/dayone.

Send your suggestions, comments, and critiques by email to dayone@juniper.net.

Welcome to *This Week*

This Week books are an outgrowth of the extremely popular *Day One* book series published by Juniper Networks Books. *Day One* books focus on providing just the right amount of information that you can apply, or absorb, in a day. On the other hand, *This Week* books explore networking technologies and practices that in a classroom setting might take several days to absorb. Both book series are available from Juniper Networks at: www.juniper.net/dayone.

This Week is a simple premise – you want to make the most of your Juniper equipment, utilizing its features and connectivity – but you don't have time to search and collate all the expert-level documents on a specific topic. *This Week* books collate that information for you, and in about a week's time, you'll learn something significantly new about Junos that you can put to immediate use.

This Week books are written by Juniper Networks subject matter experts and are professionally edited and published by Juniper Networks Books. They are available in multiple formats, from eBooks to bound paper copies, so you can choose how you want to read and explore Junos or other Juniper Networks technologies, be it on the train or in front of terminal access to your networking devices.

What You Need to Know Before Reading

Before reading this book, you should be familiar with the basic administrative functions of the Junos operating system. This includes the ability to work with operational commands and to read, understand, and change the Junos configuration. Juniper's *Day One* books (www.juniper.net/dayone), as well as the training materials available on the Fast Track portal, can help to provide this background (see the last page of this book for these and other references).

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

- Having access to a Junos device while reading this book is very useful. A number of practice examples that 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 book, a basic understanding of programming concepts is beneficial.
- You have read *Day One: Navigating The Junos Xml Hierarchy* and are familiar with introductory material on both XML and XPath as they relate to Junos.

Assumptions

This book makes a few assumptions about you, the reader, and your level of knowledge:

- You have a basic understanding of programming and scripting concepts – akin to experience with languages like Perl or Bash.
- You are familiar with XML document definitions, or can learn more about XML from http://www.w3schools.com.
- You are familiar with XPath expressions, or can learn more about XPath from http://www.w3schools.com.

If you meet these requirements then this book's contents will make sense to you and the concepts and constructs will be easy to learn and master.

After Reading This Book You'll Be Able To

This book can help you automate operation tasks on your devices and in your network. You will learn specific tasks and functions in this book, and when you're done with it, you'll be able to:

- Understand the Junos automation programming model.
- Understand the reasons for using SLAX as a programming language.
- Understand the Junos automation development environment to write, deploy, and debug scripts.

- Understand the fundamentals of the SLAX language from the perspective of a "classical" script programmer.
- Interact with Junos using the XML API.
- Learn to use interactive console I/O.
- Learn to use advanced print-formatting and regular-expression processing.
- Learn advanced file storage and "scratch memory" topics.
- Learn complex XPath expression techniques for Junos.
- Learn techniques for integrating Junos automation with your management systems.
- Learn advanced event scripting topics.

Additional Resources

The most common knowledge shift for many new Junos automation developers is to acquire a good grasp of the XSLT programming paradigm. While many programmers may be familiar with procedural languages such as Perl and Java, the transformative nature and programming framework of XSLT may be new to them. The following resources are excellent sources of information for getting up to speed or diving deep into these technologies.

- One of the best online places to learn about the standards-based languages and protocols is the W3school site:
  - XML Tutorial: http://www.w3schools.com/xml/default.asp
  - XPath Tutorial: http://www.w3schools.com/xpath/default.asp
  - XSLT Tutorial: http://www.w3schools.com/xsl/default.asp
- Junos also supports many of the Extensions to XSLT (EXSLT). Online documentation on EXSLT can be found at: http://www.exslt.org/ .
- There are also a number of very good books available from O'Reilly Media, including *XSLT 1.0 Pocket Reference*, *Learning XSLT*, and *The XSLT Cookbook*. For more, go to http://www.oreilly.com.
- Juniper Networks developed a four-hour interactive computer based training video. This is an excellent starting point for quickly learning many of the fundamentals of Junos scripting. This video can be found on the main Junos automation site:
  - http://www.juniper.net/us/en/community/junos/script-automation/ .
- The "Junoscriptorium" is a site hosted by Google Project that contains a vast collection of Junos scripts. To access these scripts go to the main site: http://code.google.com/p/junoscriptorium/ .
- Juniper Networks hosts an online message board forum dedicated to Junos Automation questions. The URL for this site is: http://forums.juniper.net/ and then select Junos Automation (Scripting).
- The Juniper Networks Technical Publications site contains the definitive reference manuals on Junos automation, both "on-box" scripting and "off-box" orchestration. As this material is periodically updated to document new

XML APIs, the documentation is found under a specific Junos release. The Techpubs main site is: http://www.juniper.net/techpubs/.

- The SLAX language was published to the open source community and is hosted on the Google Project site: http://code.google.com/p/libslax/.
- The complete SLAX reference language documentation can be found off this main page, and directly at: http://code.google.com/p/libslax/downloads/detail?name=slax-manual.html.
- The SLAX project site also includes an off-box utility that can be used to experiment and learn SLAX on a host machine (Unix/Cygwin). The 'slaxproc' utility provides a number of useful features, including converting between SLAX and XSLT.
- And of course there are other books about Junos automation, just like the one you are reading, from Juniper Networks Books . Check out the current book list, at www.juniper.net/dayone, and revisit frequently for new titles in the growing library.

Chapter 1

Getting Started with Junos Automation Scripting

Junos automation scripting, known in the field simply as *Junos scripting*, is a key technology and a fundamental capability that enables you to automate your Junos devices for your own (and unique) operational requirements. You can deploy Junos scripts on any Juniper Networks device that runs the Junos operating system, such as the highly successful MX Mid-Range series (MX5/ 10/40/ 80 routers), the M- and T series of routers, the EX series of Ethernet switches, and the SRX Services Gateways series of network devices. That's a lot of powerful iron.

This chapter will introduce key concepts about Junos automation scripting, such as what it is and how it works, and it will then show you how to get started – what you need and how to do it. Then at the end, it's practice time.

TIP If you haven't already, you should check the front matter of this book and the authors' list of assumptions about you and what you already know. After this introductory chapter, this book is off on a tear, and by the end you'll be writing some pretty powerful stuff. The more you match the basic assumptions of this book's instructional requirements, the more sense it will all make and the faster you'll be able to flip through the pages and get back to work.

NOTE The end of this chapter gets you started with the equipment you need to follow along, but for now, let's introduce Junos automation scripting at a 10,000-foot level.

Introducing Junos Automation

Junos scripts are interpreted programs, meaning they do not need to be compiled into machine code like C, or intermediate code like Java. Read the following script, a complete example of a "Hello, World" Junos script written in 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> {
      <output> "Hello World!";
   }
}
```

Let's examine this script to understand what you're looking at. The first line must always declare the language version, and it is always 1.0 since Junos only supports XSLT/XPath 1.0 at this time. The lines starting with "ns" are namespace declarations – and don't worry, the script does not "go out to the Web" to retrieve anything. The "import" statement includes additional code provided by Juniper Networks, that is, "helper" functions. Next, you can think of the "match /" block as the main starting point for the script, like main() in "C". This is an example of an operational script (op script), so the results of the script must be returned within a `<op-script-results>` element. Finally the `<output>` tag tells Junos to display a string to the console.

You might be thinking "Wow – what is all this?? … and how do I make heads or tails of writing SLAX?" This book is your guide for learning SLAX for Junos automation, so keep reading and it will all become perfectly clear!

There are three main types of Junos automation scripts you will learn about:

- operations automation (*op scripts*)
- configuration automation (*commit scripts*)
- event based automation (*event scripts*)

These automation scripts are referred to as *on-box automation* because the scripts are physically located on the devices and run within the construct of the Junos operating system.

An overview of these scripts is listed in Table 1.1.

Table 1.1 On-Box Automation

	Operations Automation	Configuration Automation	Event Automation
What does it do?	Instructs Junos as prompted by the command-line, NETCONF, or other scripts.	Instructs Junos during the configuration / commit process.	Instructs Junos of actions to take in response to events.
How it Works with Junos	Creates custom commands for specific solution/user needs. Combines a series of iterative steps to diagnose network problems. Performs controlled configuration changes.	Abstracts a complex configuration into a simple set of base commands. Options to provide warnings, post logs, prevent the configuration, self-correct the configuration.	Gathers relevant troubleshooting information and correlates events from the first leading indicators. Automates event responses with a set of actions.
Key Benefits	Reduces risk and improves productivity. Automates troubleshooting. Offers a controlled configuration.	Assures compliance with business rules and network/ security policies. Provides change management to avert and even correct errors. Simplifies and speeds setup of complex configuration.	Automates time-of-day configuration changes (e.g. "green" power-saving actions). Speeds time-to-resolve to reduce the downtime and cost of events. Automates response to leading indicators to minimize the impact of events.

There is another form of Junos automation, commonly referred to as *network orchestration* or *off-box* automation, used by a remote management system, such as, for example, your management system (OSS/BSS). This type of automation uses the standards-based protocol NETCONF (essentially XML over SSH) to remotely connect to the Junos devices and perform operational and configuration commands.

The network orchestration topic is not covered in this book, but it is worth mentioning that Junos is capable of off-box automation using the same XML-based API used by the on-box automation.

TIP Off-box automation is another book, another topic, but remember what you learn here is applicable over there. (Look for a book on Off-box Junos Automation in the near future in the Day One library at www.juniper.net/dayone.)

Why Use Junos Automation Scripting?

Obviously, if you've read this far, you're interested in Junos scripting, but you should also spend a few minutes covering the *why,* if only for the duration of this book.

Junos scripting is used in the largest of complex environments to enable scalability and reliably, no matter the network's purpose. It's the key to simplifying operational complexities and thus reducing the overall operational expense of managing and troubleshooting the network. No matter your network size, you can simplify complex operations, automate manual procedures, maximize uptime and availability, and optimize operational efficiency. It's true – it works – and the authors have seen it in action.

Here are a few of the more common key benefits of using Junos automation from an operators' perspective, in case you might need some bullet points for your next staff meeting:

- Best Practices: You can validate and enforce configuration changes to always conform to your business rules and best current practices.
- Leverage Expertise: You can enable everyone in your team to leverage the configuration and troubleshooting expertise of your best engineers.
- Customize: You can customize your day-to-day operations tasks with user-created CLI commands for displaying, configuring, and troubleshooting workflow.
- Simplify: You can embed configuration macros to abstract complex configurations into simple sets of user-defined base commands.
- Velocity and Agility: You can extend the Junos operating system with automation capabilities to deliver differentiating features to address your network needs.
- Reactive: You can embed your team's diagnostic expertise in Junos to correlate on-device events to detect, report, and correct issues before they impact your network.
- Proactive: You can apply some nifty time-based event automation to execute repetitive tasks per maintenance and operating procedure.
- Consistency: The Junos XML API provides for both on-device automation and network orchestration via northbound interface to OSS/BSS management using standards-based NETCONF and ensuring structured input / structured output.
- Cross Device: You can leverage native Junos remote procedure call capability to configure services and diagnose network issues effectively, spanning multiple devices.

Is Junos Scripting Difficult?

In two words: Absolutely not!

Those familiar with such scripting languages as Perl, Python, or even Bash, can quickly become skilled Junos automation script developers.

You don't need to be a programming "guru" to write Junos automation scripts, but it is helpful if you are familiar with the fundamentals of scripting languages like Perl. The focus of this book is to capture the perspective of a classical script programmer and relate known techniques to those used for Junos scripting. That's because the Junos scripting programming model is a little different than what you might be used to, so

the book takes what you already know and applies it to Junos scripting. Whatever is different, you'll simply learn along the way.

So Do I Need to Learn Yet Another Programming Language?

Well, the point of this book is to teach you SLAX for Junos automation scripting, so yes. But it's easy to learn and you can pick it up in just a few hours. With this book in front of you and your favorite text editor, you will be shown what to do, step-by-step. You'll soon learn in the rest of this chapter that SLAX is relatively lightweight in terms of syntax, grammar, and programming constructs. By book's end you'll be just like the rest of us who think SLAX actually makes Junos automation easy (and fun!) to learn and implement.

Most people write Junos scripts in the Stylesheet Language Alternative syntax (SLAX) because it's easier than the alternative. SLAX is really nothing more than *syntactic sugar* for the XSLT language.

XSLT, Extensible Stylesheet Language Transformations, is a declarative, XML-based language used for the transformation of XML documents. XSLT is a standards-based language used to transform XML input into any form of textual output.

NOTE Junos scripts can be written in SLAX, XSLT, or a combination of both. However, this book focuses on SLAX.

As Junos scripts have their fundamentals in XSLT, you will need to be familiar with XML and XPath concepts as they are instrumental in the XSLT (and therefore SLAX) programming model.

Generally speaking, this is the most common "mind-shift" for new Junos script developers - the XSLT programming model and used XPath expressions. While you may be very familiar with procedural languages, such as Perl, the transformational nature and programming framework of XSLT may be new. Stay close and this book shows you how.

MORE? If you're not familiar with the basics of XSLT, XML, or XPath, note that a number of online references are presented in the front matter of this book.

Why XSLT?

Junos management "under the hood" is based in XML. When you enter commands on the Junos CLI, commands are actually converted to XML-based Remote Procedure Calls (RPCs) and executed by the Junos management daemon. The Junos configuration file is also represented as an XML file. XSLT is the standards based language to manipulate XML … so XSLT is a natural fit for Junos.

The XSLT programming model is shown in Figure 1.1. XSLT is an interpreted language meaning that the XSLT "program" is executed by a binary executable that is compiled on the target device (i.e. the Junos device). This binary is called the *XSLT processor*. The XSLT processor takes two inputs as shown in Figure 1.1: the first input is the XSLT program, also referred to as the *stylesheet tree*; and the second optional input is the XML document, also referred to as the *source tree*. The XSLT processor produces XML output (by default), and this output is referred to as the *result tree*.

In short: XML input + XSLT program = new XML output.

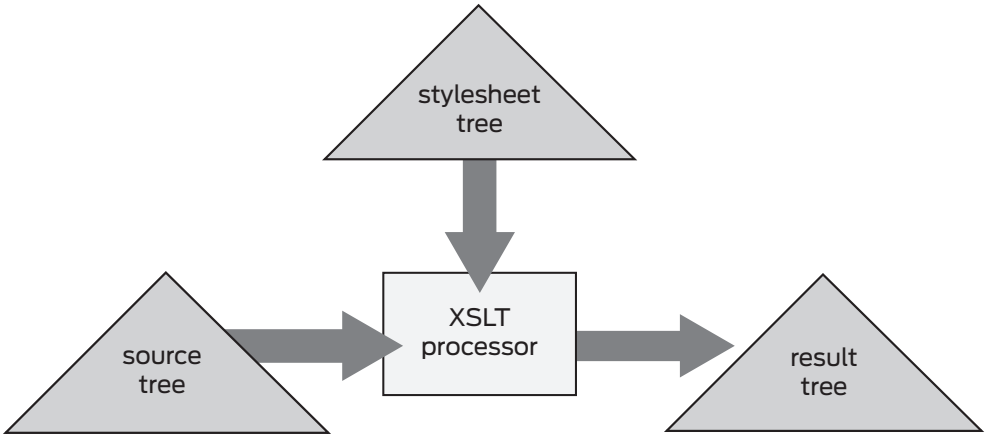


Figure 1.1 The XSLT Programming Model

Junos utilizes this same processing model for each of the automation script types: operational scripts, commit scripts, and event scripts. The XSLT processor is built into Junos, and the Junos script you write is the "stylesheet tree."

In short: XML input + Your Script = XML output that Junos knows how to process.

Why SLAX?

But why not just write scripts in native XSLT rather than learn a new syntax? XSLT is a powerful tool for handling XML, but the syntax is cumbersome, verbose, and "optimized" for machine-to-machine communication. XSLT can be inconvenient for humans to write especially complex programs. If you've never experienced writing in XSLT it might be worthwhile to examine some sample code on the Web. The occasional benefits of writing programs in native XSLT are often outweighed by the readability issues of dealing with the syntax. Simply put, writing programs in XSLT is tedious and error-prone, so the Junos development team conceived of the *Stylesheet Language Alternative syntaX*, or SLAX.

SLAX has a simple syntax that follows the style of C, or Perl. Programming constructs and XPath expressions are moved from complicated and verbose XML elements and attributes to "classical" language constructs. The cumbersome nature of writing XML hierarchies and dealing with escaped-quoting issues is replaced by simple use of parentheses, curly braces, and simple quotation marks - all familiar delimiters for programmers.

With SLAX, the underlying constructs are completely native to XSLT. Nothing is added to the XSLT engine. You could think of SLAX as a pre-processor for XSLT, turning SLAX constructs like `if/then/else` into the equivalent XSLT constructs like `<xsl:choose>` and `<xsl:if>` before the real XSLT transformation engine gets invoked.

A few highlights on SLAX include:

- You use `if/then/else` instead of `<xsl:choose>` and `<xsl:if>` elements.
- You put test expressions in parentheses.

- You use == to test equality (to avoid building dangerous habits).
- You use := for Result Tree Fragment (RTF) to node-set conversion.
- Curly braces are used to show containment instead of closing tags.
- You can perform string concatenation using the _ operator (lifted from perl6).
- You write text strings using simple quotes instead of the <xsl:text> element.
- Invoking named templates with a syntax resembling a function call.
- Defining named templates with a syntax resembling a function definition.
- Simplified namespace declarations.
- Reducing the clutter in scripts, by allowing the important parts to be more obvious to the reader.
- Writing more human-readable scripts.

MORE? The complete SLAX language reference is available online at: http://code.google.com/p/libslax/downloads/detail?name=slax-manual.html.

The Junos Script Programming Model

Let's turn our attention to the fundamental programming model for each of the three types of Junos scripts: op, commit, and event.

It's important for you to understand how Junos executes these scripts in the context of the operating system, but not via a detailed overview. Later sections in this book provide useful tutorials, examples, and step-by-step techniques to take advantage of each of these script types, and in some cases, combining two scripts to create very powerful automation solutions.

MORE? See all the other books in the *Junos Automation Series* at www.juniper.net/dayone, for in-depth coverage of all the script types.

Junos Op Scripts

Junos operational scripts, or *op scripts*, are typically the first type of script a new programmer begins developing. That "Hello, World" program at the start of the chapter is an op script. Op scripts are typically invoked by a user at the Junos CLI. They are written to perform a series of actions and can include parameters provided by the user or can prompt the user for input. An op script can automate the same operational commands the user could enter directly. An op script can also be used to automate configuration changes.

It is important to understand that op scripts are executed in the context of the user invoking the script. So this means that the op script can only perform the functions that the user could normally perform.

The Junos op-script programming model is illustrated in Figure 1.2.

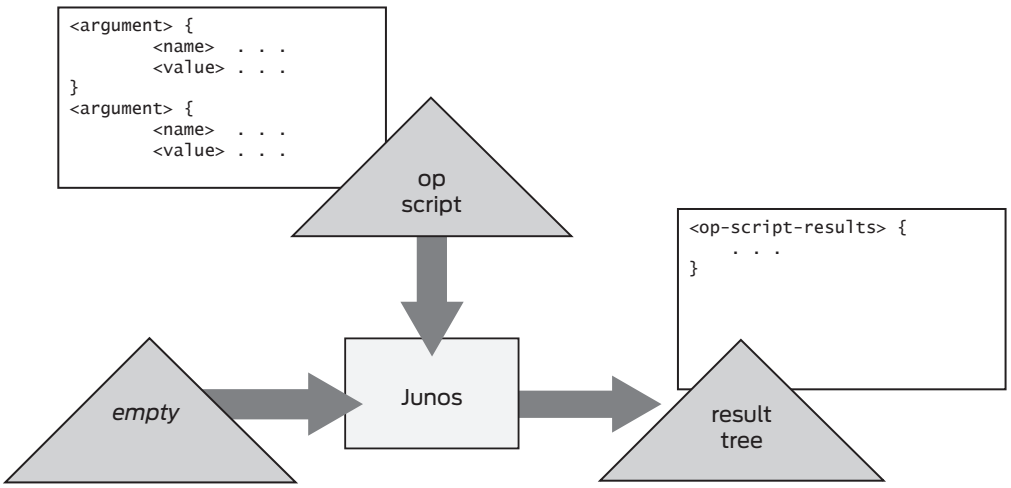


Figure 1.2 The Op-Script Programming Model

Here are some important aspects of the diagram in Figure 1.2:

- Conceptually, the *source tree* is empty, meaning that user-input is not provided as an XML document. If the op script has user-defined parameters, they are passed via `param` arguments defined in the script; this technique is similar to the use of $ARGV in Perl, and will be more fully covered later in this book.
- The op script is the *stylesheet tree*.
- The result tree is generally a series of output statements defined within an `<op-script-results>` XML block (such as the "Hello World" op script presented at the start of this chapter).

The Junos operating system provides a comprehensive set XML-based API remote procedure called (RPCs). The op script can invoke Junos commands by formulating the XML RPC equivalent of the CLI command and communicating it directly to the management daemon. *The op script does not issue native CLI commands.* For example, a user at the CLI prompt would enter the command `show chassis hardware` to display the hardware inventory. A Junos op script would use the equivalent XML RPC : `<get-chassis-inventory>`.

NOTE You can easily determine the XML RPC from the Junos CLI by using the "`display xml rpc`" pipe-option after the command. For example, `show chassis hardware | display xml rpc` would show you the XML RPC's `<get-chassis-inventory>`.

MORE? For more introductory information on Junos op script programming refer to the Juniper Networks booklet: *This Week: Applying Junos Automation*, available at www.juniper.net/dayone.

Junos Commit Scripts

Junos commit scripts are programs that are invoked by Junos any time a commit (or

commit check) operation occurs, as illustrated by Figure 1.3. A commit script can be used to modify the candidate configuration, to issues warnings, or to prevent the configuration from becoming active. The commit script can also perform operational commands to retrieve Junos information, for example, `<get-chassis-inventory>`.

Here is a short list of when commit operations could occur, and in all of these cases, the Junos commit script is executed in the root user context:

- At device boot-up time when the configuration is initially loaded.
- When a user at the CLI enters the `commit` or `commit check` command.
- When an op script performs a configuration change.
- When an event script performs a configuration change.
- Upon automatic configuration rollback triggered by a commit confirmed.
- When an OSS/BSS system is performing a configuration change; this is an example of network orchestration or off-box automation whereby an external management system is invoking the Junos XML API via the NETCONF protocol.

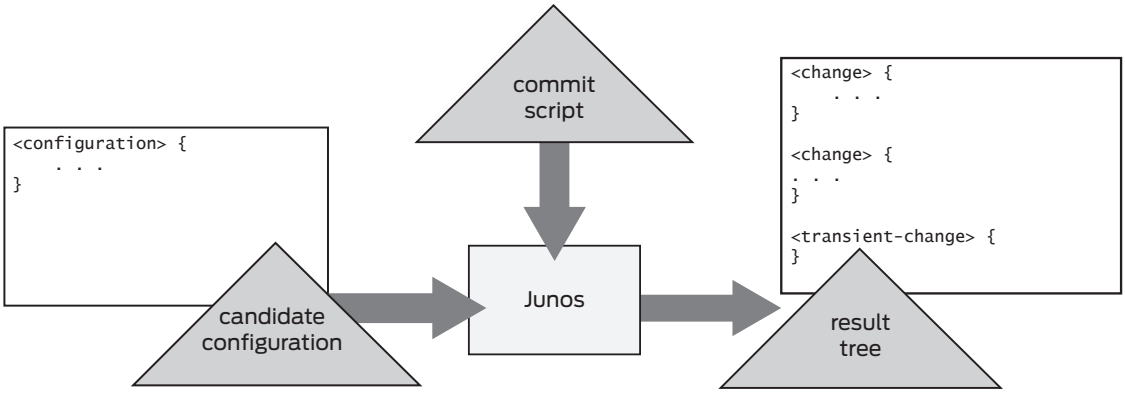


Figure 1.3 The Commit Script Programming Model

Here are some important aspects of the diagram in Figure 1.3:

- The candidate configuration file is the source tree input.
- The commit script is the stylesheet tree.
- The result tree is a series of specific XML blocks that can make changes to the candidate configuration, generate warnings, or even prevent the configuration from becoming active.

Any Junos device can be configured to run multiple commit scripts. Each script, however, receives the same copy of the candidate configuration. Although the Junos operating system executes the commit scripts in the order specified in the configuration file (in a serial fashion), the changes made by prior scripts in the list do not pass to later scripts in the list.

MORE? For more introductory information on Junos commit script programming refer to the Juniper Networks booklet: *This Week: Applying Junos Automation*, available at www.juniper.net/dayone.

Junos Event Scripts

Junos event scripts allow you to automate Junos when specific events occur. Examples of this include when an interface goes down or when it is Saturday at 5:00pm. The type of events, triggers, and sequences of actions, can be combined into some powerful automation controls, especially when you factor in that event scripts can perform operational commands, make changes to the configuration, or log information (e.g. to syslog).

By default event scripts are executed in the root user context, but they can be configured to run in as specific user content as well. The event script programming model is illustrated in Figure 1.4.

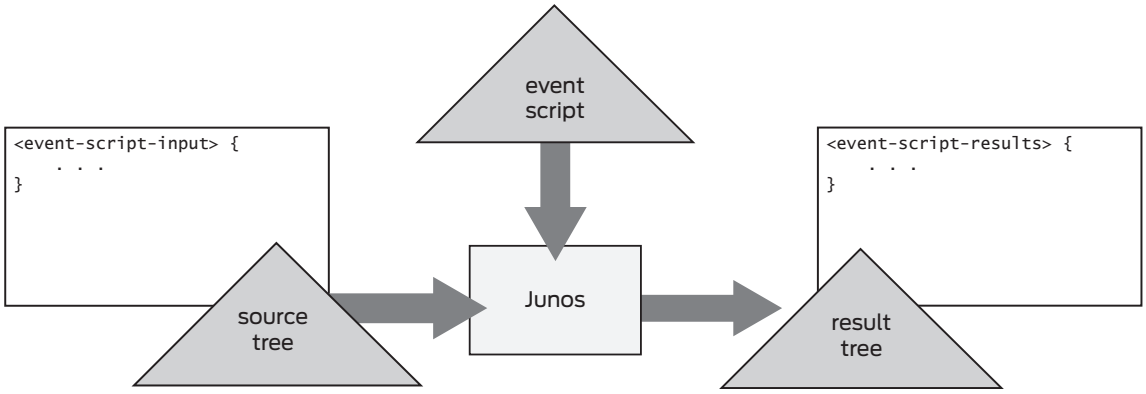


Figure 1.4 The Event Script Programming Model

Here are some important aspects of the diagram in Figure 1.4:

- The source tree is an XML **<event-script-input>** block that contains information about the event, including the <trigger-event> child block that provides the details on the event that triggered the event policy.
- The event script is the *stylesheet* tree.
- The result tree is contained within the **<event-script-results>** XML block.

MORE? For more introductory information on Junos commit script programming refer to the Juniper Networks booklet: *This Week: Applying Junos Automation*, available at www.juniper.net/dayone.

Junos Script Global Variables

Junos scripts have access to a number of global variables, namely: `$hostname`, `$product`, `$script`, `$user`, `$localtime`, and `$localtime-iso`.

Junos 11.1 adds a new global variable: `$junos-context`. This variable is a node-set that contains relevant information that scripts could be interested in. Facts such as the user's TTY, whether or not the script is running on the master RE, or the fact that the commit script is running as part of the boot-up commit, are things that have been requested in the past and are now being delivered through this global variable.

The use of a single variable to store all the desired information, rather than multiple

parameters, is a departure from the past but offers an efficient and scalable solution that can be added to in the future. The "legacy" global variables, e.g. `$hostname`, will still be available for backwards compatibility.

MORE? For more information on the `$junos-context` special variable, refer to "Special Variables" in Chapter 2 of this book.

Getting Started

There are three basic steps required to develop, test, and deploy Junos scripts. Remember that Junos scripts are physically stored on the Junos device.

The development cycle for Junos scripting is as follows:

- Step 1: Writing the Junos script code, typically done on a host (development) machine such as Windows, Mac, or Linux.
- Step 2: Copying the Junos script code to a Junos device, and into the correct directory.
- Step 3: Testing the Junos script code, and troubleshooting either (a) syntax errors or (b) logic errors. If the test fails, go back to Step 1.

NOTE There is also a one-time step of enabling the scripts so that Junos knows that these scripts are valid and will run in the proper context (op script, commit script, or event script).

Did you know that the Junos device file-system can be mounted to your host computer? This is a handy trick when you are developing scripts to eliminate the "copy files to target" step. Since Junos can use NFS mounts, mounting the device is fairly trivial if the host computer is Unix-based. If your host computer is Windows-based, then a number of applications can be used to support the same remote file system access. For Windows you could try WinSCP, which is free, and ExpanDrive, which is not. For Macs you might try a program called CyberDuck.

Another tip is to make small edits on the Junos device by dropping into the Junos OS shell and editing the script files directly with the VI Text Editor in Junos itself.

These are on-device development techniques that can cut down on the file and copy steps, but care must be taken that the final scripts are saved off the device so as to not lose important changes.

Step 1: Writing Scripts

You can use any text editor you wish to write Junos scripts.

There are also a number of commercial text editors that can be customized to support SLAX syntax, meaning that keywords can be highlighted, and the structure of the code can be properly indented. The Junos Automation user community has developed highlight modes to support SLAX for jEdit and Eclipse . These are not formally supported by Juniper Networks, but any question and suggestion regarding these tools can be directed to the J-Net Junos Automation board.

MORE? The jEdit program can be downloaded from: http://www.jedit.org/ and the SLAX highlight mode can be downloaded from the J-Net Automation forum at http://forums.juniper.net/t5/Junos-Automation-Scripting/What-tools-do-you-use-to-develop-Junos-automation-scripts/td-p/52352.

MORE? The Eclipse IDE can be downloaded from the site: http://www.eclipse.org/. It is important to note, however, that the SLAX plug-in for Eclipse is still under development, and should hopefully be completed by the time you are reading this book.

Step 2: Storing Scripts on the Junos Device

In order for Junos scripts to be correctly processed, they must be stored in the correct location within the Junos file system: */var/run/scripts*.

The `/var/run/scripts` directory should always point to the correct script directory whether they are stored on flash or disk. A listing of this directory shows:

```
user@junos> file list /var/run/scripts

/var/run/scripts/:
commit/
event/
import@ -> /usr/libdata/cscript/import
lib/
op /
```

The top-level scripts directory has sub-directories for each of the script types: op scripts are stored in the `/var/run/scripts/op`, commit scripts are stored in the `/var/run/scripts/commit` directory, and event scripts are stored in the `/var/run/scripts/event`.

ALERT! The `/var/run/scripts/import` directory is used exclusively by Junos, so don't try to put anything there.

NOTE Junos 11.1 added a new directory: `/var/run/scripts/lib`. This directory can be used to store any files you would like to share between scripts. You can think of this as an *include* directory.

Step 3: Enabling Scripts

In order for Junos SLAX scripts to be active, they must be properly enabled in the Junos configuration file.

For op scripts:

```
[edit system scripts op]
set file my-op-script.slax
```

For commit scripts:

```
[edit system scripts commit]
set file my-commit-script.slax
```

For event scripts:

```
[edit event-options event-script]
set file my-event-script.slax
```

Debugging Scripts

Finally let's review the debugging options available for Junos Automation scripting. (Further details on this topic are presented in subsequent chapters.)

Typically, *debugging* falls into two broad categories: (1) correcting syntax errors and (2) correcting logic errors.

People who are new to SLAX (or XSLT) typically debug syntax errors early on in the three-step process. The development cycle of writing a script, copying it to a device, and then finding that you have a syntax error can be quite time consuming. Juniper has developed an off-line tool called *slaxproc* that you can use to debug these types of syntax errors, to help ensure you have a syntax-error-free script before copying it over to the Junos device. Details on slaxproc are discussed in the Appendix.

Debugging logic errors can be a bit more challenging, and you have the following options available:

- Using "print" statements in your code.
- Using traceoptions file and flags to track program execution.
- Using "trace" statements in your code to log custom messages to a traceoptions file.
- Using "progress" statements in your code that are only invoked when you use the `detail` option; this is only used for op-scripts.

In Junos 10.4, an *on-target* debugger was added that can be executed for op scripts, using a hidden parameter: `invoke-debugger cli`, which is discussed in detail in the Appendix.

Hello, World Step-by-Step

Okay, this chapter has reviewed what Junos Automation is, how it works, and what you need to get started. Now let's show you an example by using step-by-step instructions for testing out the Hello, World op script that was at the start of this chapter. You should now readily understand the steps cited.

The first step is to create the file, named `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";

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

When invoked, this script will output the string "Hello, World" to the CLI console.

The next step is to copy this file to the target Junos device in the target directory `/var/run/scripts/op`. This step is commonly performed either using an FTP client or the `scp` utility. The following is an example using `scp`:

```
user@mylaptop$ scp hello-world.slax user@junos:/var/run/script/op
```

The next step is to enable this script on the Junos device:

```
user@junos> configure

[edit]
set system scripts op file hello-world.slax

[edit]
commit and-quit
```

Finally the op script can be executed at the Junos CLI:

```
user@junos> op hello-world
Hello, World!
```

Voila! Hello World.

Summary

The rest of this book builds upon the short introduction of this chapter. Next you should roll up your sleeves and get into the basic fundamentals of SLAX. After that, the book covers a variety of topics including XPath, OSS integration, and file and storage options.

There are three items in the Appendix that you can refer to at any time: jcs functions, the slaxproc utility, and the op script debugger.

All of the material in this book is geared to provide a guide between familiar procedural- based languages that you may know (such as Perl) and SLAX. Other books in the Day One Junos Automation Series are available at www.juniper.net/dayone.

Chapter 2

SLAX Fundamentals

To be proficient in SLAX programming you must be aware of a few fundamental programming concepts. These concepts come from the XSLT programming environment, so if you are familiar with XSLT, then these will not be new to you. Wherever you see SLAX in the following text, the same applies to XSLT, so you won't need to see "SLAX / XSLT" throughout this document. If Perl or Java is your programming background, then some of these concepts might be unfamiliar and perhaps challenging at first. The goal of this chapter is to provide a guide to help you navigate between familiar procedural based languages (such as Perl) and SLAX.

MORE? XPath expressions form the foundation for SLAX programming. If you are not familiar with XPath expressions you are strongly encouraged to read *Day One: Navigating the Junos XML Hierarchy* book found at www.juniper.net/dayone, or any other online materials that can be found in the Reference Section at either the beginning or end of this book.

Fundamentals

There are a few fundamental SLAX rules and concepts:

SLAX is based on XPath 1.0 and XSLT 1.0.
You can find a lot of reference material for XPath 2.0 and XSLT 2.0, but unfortunately you cannot use it.

Variables are immutable.
This means that you can set a variable only once and cannot change it later.

The "Node-Set" variable data-type is from XPath.
A node-set is a set of XML nodes that can be processed by XPath expressions. Recall that Junos RPC APIs are XML based, so their return values are node-sets. The Junos configuration file is also an XML document, so it is very easy to process using the node-set data-type. The following code snippet illustrates the definition of a node-set variable:

```
var $my-ns-var := {
   <interface> {
      <name> "ge-0/0/0";
   }
   <interface> {
      <name> "ge-0/0/1";
   }
}
```

The "Result Tree Fragment" (RTF) variable data-type is from XSLT.
An RTF can either store a simple string or block of XML data (not a node-set). A SLAX script can only perform the following actions on an RTF: emit it to the XSLT result tree, convert it to a string, or convert it to a node-set so you can then operate on the contents of the XML nodes. The following code snippet illustrates the definition of a RTF variable:

```
var $my-rtf-var = {
   <interface> {
      <name> "ge-0/0/0";
   }
   <interface> {
      <name> "ge-0/0/1";
   }
}
```

ALERT! You should pay close attention to the node-set and RTF code examples. They are almost identical *except* that the variable assignment for the node-set variable `$my-ns-var` is made using the colon-equals (:=) operator and the RTF variable `$my-rtf-var` is made using equals (=) operator. This very small difference is a very common programming error. Watch out for this one!

A Template only returns Result Tree Fragments.
A template is a block of code that can be used similarly to a *procedure* or *subroutine* in procedural languages. Technically it is different from a XSLT function, but this chapter discusses the specific differences in a later section. A template will only return an RTF. If you want to use any resulting XML data, you must first convert it to a node-set. Fortunately the SLAX language makes this a very easy process. You've actually just seen this in the above examples – just use the colon-equals (:=) operator to convert an RTF to a node-set.

Context Processing.
XPath and XSLT processing maintain a context for the current XML node being processed. In Perl there is the $_ variable that serves a similar purpose. In SLAX, the *current* context is identified using the decimal (.) expression value. It is important to understand when the context is set and how to use this mechanism as it is a regular technique for SLAX programming. Using the node-set code example shown above, the following is a code snippet that illustrates the use of the current context node (.) to iterate through the list of interfaces and output the `name` element value:

```
var $my-ns-var := {
   <interface> {
      <name> "ge-0/0/0";
   }
   <interface> {
      <name> "ge-0/0/1";
   }
}

for-each( $my-ns-var/interface ) {
   var $ifd = .;                 /* $ifd is being assigned the 'current' interface node */
   <output> $ifd/name;
}
```

SLAX Script File Structure

It is important for you to understand the structure of the SLAX script file, as there are specific programming directives that must be placed in a specific order. Fortunately all Junos script types follow the same basic file structure.

As you have already seen, SLAX supports code commenting using C-style open-comment (/*) and close-comment (*/) notation. Comments can be located anywhere in the SLAX file.

At the top of the file is the SLAX boilerplate that defines the commonly used namespaces, and imports the templates provided by Junos.

```
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";
```

You can include any additional namespace definitions following the `ns jcs` line, and examples of adding your own namespaces are shown in a later section of this chapter.

You can include any additional import files after the `junos.xsl` file. These could be written in SLAX, XSLT, or a combination of both. You can think of these as *include* files, as their contents are effectively included into your script file.

NOTE The `import` statement is a top-level directive, meaning that it must at the top of the file before any other variable or template definitions.

If you are writing an op script, you would then need to define any command line parameters using the `param` directive:

```
param $interface;
param $mtu-size = 2000;
```

The `param` variables are immutable just like any other SLAX variable. You can, however, define a parameter with a default value as illustrated by the `mtu-size` parameter. If the user does not specify the `mtu-size` value when invoking the op script, then the value (2000 in this case) would be used.

Op scripts also support the use of the special global variable called `$arguments`. The `$arguments` variable is used by Junos to provide CLI help information to the user. The `$arguments` variable is optional, but it should always be included to provide CLI help:

```
var $arguments = {
   <argument> {
      <name> 'interface';
      <description> 'The interface you want to control';
   }
   <argument> {
      <name> 'mtu-size';
      <description> 'The MTU size (default is 2000)';
   }
}
```

For event scripts you could embed the event definition into the script rather than placing it into the Junos configuration file. This capability was added in Junos 9.0, and is the recommended approach. The special global variable `$event-definition` is recognized by Junos for this purpose.

Both the `$arguments` and `$event-definition` special variables are discussed later in this chapter.

Next in the file structure is the *main template*, where the script code conceptually begins. The most common programming practice is to put the main template at the top of the file. The main template is the template that matches the first element in input XML document, or `empty` if there is no input XML document (source-tree). The template that matches anything (including an empty source-tree) is `match /`.

Each Junos script type has a different main template.

The main template for an op script is:

```
match / {
   <op-script-results> {
      /* your code goes here */
   }
}
```

As you can see, the main template is using the match anything notation (`/`), and the code execution begins here.

The main template for an event script is:

```
match / {
   <event-script-results> {
      /* your code goes here */
   }
}
```

And the main template for a commit script is:

```
match configuration {
     /* your code goes here */
}
```

Here you can see that the main template is more specifically scoped. It matches the XML node called `configuration`, which is part of the candidate configuration presented to the commit script at runtime.

MORE? The top-level match for a commit script is different from the op script and event script because of helper-code found in the import file `junos.xsl`. For details on why `configuration` is the main template, refer to *This Week: Applying Junos Automation*, available at www.juniper.net/dayone

Variables

Let's review the variables in SLAX, starting with declarations.

Declarations

Naming

When declaring variables, you must observe the following rules:

The keyword `var` is used to define a variable. Once a variable is defined, it cannot be changed. SLAX variables are *immutable*.

Variable names must start with the dollar-sign (`$`).

Variable names can contain any alpha characters (upper and/or lower case), numeric values, the underscore (`_`), and the dash (`-`).

```
var $myvar = 5;
```

Do not start any variable name with `junos` because it is commonly reserved by Juniper Networks.

Variable names can also contain the colon (`:`) character when designating a namespace. Namespaces allow you to attach a prefix to a name so that you can identify it as your own. This prevents you from clobbering other variables that might have the same base name. Using namespaces is a good practice if you are creating global variables or creating functional libraries of code. Junos provides a functional library for scripting, for example, the `jcs:parse-ip()` function:

```
var $myco:ip_info = jcs:parse-ip("192.168.1.29/24");
```

Do not declare any variables, templates, functions, parameters, elements, etc., using any of the following reserved namespaces: `junos`, `jcs`, `xnm`, `slax`.

Do not declare any variables with the same name as any of the special variable

names. You have already been introduced to the `$arguments` and `$event-definition` special variables. Refer to the *Special Variables* section in this chapters for a complete listing.

Scoping

Variables can be scoped in one of three levels: global (1), template/function (2), and within a programming control-block (3). Only global variables can be overridden by lower scoped definitions. Best practice is to avoid using the same variable name in multiple nested scopes.

Global variables and parameters are declared outside of any templates. Global variables and parameters are typically declared at the top of the script file, but this is not a requirement.

Templates and functions are ways to modularize your code (the specific differences between these two will be explained later in this chapter). These scoped variables are declared within the code definition of the template or function. The following example illustrates template variables (`$var1, $str`) and a template parameter (`$arg1`):

```
template foo( $arg1 ) {
   var $var1 = $arg1 + 10;
   var $str = "the value is: " _ $var1;     /* the "underscore" is the string-concat operator */
}
```

Assuming that the template `foo` was called with an argument of 1, the resulting value in `$str` would be `the value is: 11`.

NOTE SLAX simplifies the syntax of template definitions. The template parameter `$arg1` was declared within the definition of the template and does not require the keyword `param`.

Template variables are typically declared at the top of the template definition, but this is not required. Variables must be declared, however, before they can be used. For example, the following produces a SLAX syntax error:

```
template foo( $arg1 ) {
   var $str = "the value is: " _ $var1;     /* SYTAX ERROR !! */
   var $var1 = $arg1 + 10;
}
```

However, you can intersperse your variables throughput the template definition:

```
template foo( $arg1 ) {
   var $v1 = $arg1 + 10;
   <output> "v1 is: " _ $v1;

   var $v2 = $v1 * 1000;
   <output> "v2 is: " _ $v2;
}
```

SLAX supports two programming control blocks: `for-each` and the `if/then/else`. Variables can be declared within the code portions of these control blocks. The following illustrates a global variable (`$iflist`) being used by a template (`show-interface-list`) that is using a `for-each` block. The `$ifd` variable is locally scoped within the `for-each` block:

```
var $iflist := {
   <interface> {
      <name> "ge-0/0/0";
```

```
  }
  <interface>
    <name> "ge-0/0/1";
  }
}

template show-interface-list() {
  for-each( $iflist/interface ) {
    var $ifd = .;                 /* the "dot" is the context node */
    <output> $ifd/name;
  }
}
```

The use of a locally scoped variable is a common technique to overcome some of the challenges presented by immutable variables. In other procedural languages, the `$ifd` variable would be declared outside the `for-each` loop, and it would be assigned on each interaction. For example, the the following variable is NOT valid, but would be a common error:

```
template show-interface-list() {
  var $ifd;

  for-each( $iflist/interface ) {
    $ifd = .;                     /* INVALID - variables are immutable! */
    <output> $ifd/name;
  }
}
```

Variables can be declared within the `if/then/else` control structures as well. For example:

```
if( $this > $that ) {
  var $msg = "this is really bigger than that";
  <output> $msg;
}
else {
  var $msg = "that is not bigger than this";
  <output> $msg;
}
```

Control structures are further explained in an upcoming section of this chapter.

XPath Data Types

SLAX supports the four XPath data types: (string, number, boolean, and node-set,) and the XSLT result-tree-fragment (RTF). Since XPath and XSLT variable types are well documented in many reference documents, including the *Day One* books, this section serves to provide only a brief overview and highlight key concepts.

Booleans

Boolean variables resolve to either true or false. There are no Boolean literal values, but XPath provides `true()` and `false()` functions. XPath also provides a `boolean()` function that explicitly converts an expression to either true or false.

The numeric value of 0 converts to false and any non-zero number converts to true. An empty string (“” or ‘’) converts to false, and any non-empty string converts to true. A non-empty node-set converts to true. An empty node-set converts to false. An RTF always converts to true because it always contains a root node.

ALERT! Be careful when evaluating RTFs in a Boolean context since even an empty RTF coverts to true.

Examples:

```
var $bol = true();    /* true */
var $bol = 0;         /* false in boolean expression */
var $bol = 17;        /* true in boolean expression */
var $bol = "hello";   /* true in boolean expression */
var $bol = "";        /* false in boolean expression */
```

Here is an interesting example where a variable is assigned an empty node-set value. This example assumes that the XML input document (source-tree) or current context node does not have an element called `null`. The word null has no special meaning to SLAX:

```
var $bol = /null;     /* false in boolean expression */
```

Numbers

Numbers are stored as IEEE 754 floating point, including the special Not-A-Number (NaN) value. You can define numeric variables with or without decimal points, as math expressions, and by using the XPath `number()` function.

Examples:

```
var $num1 = 5;
var $num2 = 5.7;
var $num3 = $num1 + $num2;
var $num4 = $num1 + number("5.7");   /* results in 10.7 */
var $num5 = $num1 + true();          /* results is 6 because true() converts to the value 1 */
```

Strings

Strings can be defined by enclosing the words with either the double-quotation (") or the single-quotation mark ('); but the same mark must be used to start and end the string value. XPath also provides a `string()` function to convert a non-string variable into a string expression.

You can have a double-quoted string that contains a single quote:

```
var $str =  "How's it going?";
```

You can define a single-quoted string that contains a double quote:

```
var $str =  'She said: "How is it going?"';
```

But you cannot define a string value that includes both single and double quotes:

```
var $str = 'She said: "How's it going?';      /* INVALID!! */
```

Standard "C" character escaping is allowed as well:

```
var $str = "Hello, World!\n";
```

Result Tree Fragments

XSLT introduced the result tree fragment concept. As previously mentioned, there is not a lot you can do with these other than emit them to the script output (result-tree), treat them as a string, or convert them to a node-set.

ALERT! When declaring a result-tree-fragment variable with literal data, do not end the statement with a semicolon.

```
var $my-rtf-var = {
  <interface> {
```

```
    <name> "ge-0/0/0";
  }
  <interface>
    <name> "ge-0/0/1";
  }
}
```

Notice that there is no semicolon after the final close-brace (`}`).

NOTE The important concept to remember is that templates always return an RTF.

Consider the following example. Recall that op scripts can use the `<output>` tag to cause the output to the Junos CLI. You can put these tags within a template and the result would be the same if they were not in the template. The "Hello, World" script:

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

This script *emits* the `<op-script-results>` element and all child elements to the result-tree, as a result-tree-fragment! The `match /` code is a template, so anything it returns is an RTF.

Junos processes the result-tree and knows that when it processes the `<output>` tag, it transmits the text to the Junos CLI.

Since a template produces a result-tree-fragment, i.e. a *blob* that could to be processed in the context of the result-tree, the script could be rewritten to produce the same results:

```
match / {
  <op-script-results> {
    call hello-world();
  }
}

template hello-world() {
  <output> "Hello, World!";
}
```

Programmers often use a template to create complex XML data structures, and then use the results. Since templates produce RTFs, the results must be converted to a node-set using the `:=` operator. Consider the following template that *emits* an RTF of `<color>` elements:

```
template make-colors()
{
  <color> {
    <name> 'blue';

  <color> {
    <name> 'green';
  }
  <color> {
    <name> 'red';
  }
}
```

Then the code to call this template and convert the results into a usable node-set would be:

```
template list-colors() {

   var $colors := { call make-colors(); }

   for-each( $colors/color) {
      var $color = .;

      var $msg = "This color is " _ $color/name _ ".";
      <output> $msg;
   }
}
```

The result on the Junos CLI would be:

```
This color is blue.
This color is green.
This color is red.
```

Node-Sets

Node-sets allow you to create, and more importantly, to manipulate complex data structures in XML form. All of your interaction with Junos RPC API, for example operational commands, is done by using node-sets. All of your interaction with the Junos configuration is operating on the `configuration` node-set.

```
var $my-ns-var := {
   <interface> {
      <name> "ge-0/0/0";
   }
   <interface>
      <name> "ge-0/0/1";
   }
}                  /* no semicolon here */
```

NOTE The key difference between the node-set declaration and the RTF declaration is the use of the `:=` in the assignment of the literal value. Technically speaking, the code is actually declaring an RTF and then converting it to a node-set using the `:=` operator.

The key difference between using node-sets and RTFs is that XPath expressions only operate on node-sets. Since most of your programming uses Junos RPCs that return node-sets, and use XPath expressions, you quickly become very accustomed to using node-sets.

There are a couple of special notes regarding node-sets.

The first is the use of the union operator (`|`). The union operator can be used to create a combination of two node-sets. Consider the following node-sets:

```
var $primary-colors := {
   <color> { <name> 'red'; }
   <color> { <name> 'green'; }
   <color> { <name> 'blue'; }
}

var $fav-colors := {
   <color> { <name> 'blue'; }
   <color> { <name> 'yellow'; }
   <color> { <name> 'cranberry'; }
}
```

The union of these two node-sets:

```
for-each( ($primary-colors | $fav-colors)/color ) {
```

```
  var $color = .;
  <output> 'This color is ' _ $color/name _ ".";
}
```

Would produce the output:

```
This color is red.
This color is green.
This color is blue.
This color is blue.
This color is yellow.
This color is cranberry.
```

NOTE Notice that the union of the two node-sets does not remove duplicates. Actually, it doesn't remove duplicate string values, but it does remove duplicate nodes. The two <color> nodes with the same "blue" value are different nodes from different XML documents. A node-set is a "set" so it cannot contain duplicate nodes.

MORE? You can find a number of great set-notation related XSLT cookbook recipes in the *XSLT Cookbook* from O'Reilly Media (http://www.oreilly.com).

The second consideration with node-sets is with the comparison operators. The following comparison operators can be applied to node-sets: less than (<), less than or equal to (<=), greater than (>), great than or equal to (>=), is equal to (==), and not equal to (!=). Given the special nature of these operators, a full discussion is presented in Chapter 3.

MORE? EXSLT provides a set library of functions. Refer to http://www.exslt.org/set/index.html for more details.

Complex Data Structures

Complex data structures are *natively* supported in SLAX, again through the use of XML node-sets and XPath expressions. As you learned in the previous section, you can easily create hierarchical data structures using XML parent/child elements, as well as include XML attributes within the elements.

Mastering XML and XPath expressions enables you to master Junos script development.

MORE? If you are looking for additional training material on XML and XPath, please refer to the W3school website http://www.w3schools.com/ and the Day One book, *Navigating the Junos XML Hierarchy,* at http://www.juniper.net/dayone.

If you are looking for good examples of complex data structures, you can take a look at the Junos configuration. From any Junos device, use the following command:

```
user@junos> show configuration | display xml
```

You will be presented with the Junos configuration file in XML format.

Arrays

The XSLT language does not have a native array data-type, but there is a technique for using node-set XPath expressions to accomplish the same effect.

NOTE Array positions start at 1 and not at 0 as in Perl or C.

Since arrays are accessed via XPath expression, the array variable must be of node-set type. The following is an example array of numbers:

```
var $numbers := {
   <n> 30;
   <n> 10;
   <n> 12;
   <n> 27;
}
```

Notice that the numbers *must* have an element tag to identify each node-set element. In this case the tag <n> is being used to identify each element. The element name is completely up to you.

ALERT! You *must* use the colon-equals (:=) assignment operator when defining arrays since you need node-sets for XPath manipulation. Otherwise the variable assignment is an RTF and you will observe a syntax error when you attempt to use XPath expressions on an RTF.

The following code examples illustrate the use of arrays:

```
<output> $numbers/n[2];
```

Will produce:

```
10
```

Performing a FOR-EACH control loop:

```
for-each( $numbers/n ) {
   <output> .;
}
```

Will produce:

```
30
10
12
27
```

Hashes

The XSLT language does not have a native hash data-type, but there is a technique for using node-set XPath expressions to accomplish the same effect. Consider a Perl hash variable:

```
# perl

my %hash = ();

$hash{"ge-0/0/0"} = "up";
$hash{"ge-1/0/0"} = "down";

print "ge-0/0/0 is " . $hash{"ge-0/0/0"} , "\n"
```

There are a number of approaches in SLAX. One is to make the *key* an attribute of the data. For example:

```
var $hash := {
   <interface name="ge-0/0/0"> "up";
   <interface name="ge-1/0/0"> "down";
}

<output> "ge-0/0/0 is " _ $hash/interface[@name=='ge-0/0/0'];
```

NOTE The element name <interface> and the attribute name name are completely up to you as the programmer.

Since there is only one attribute in this example, you could *short hand* the XPath expression, as shown:

```
<output> "ge-0/0/0 is " _ $hash/interface[@*=='ge-0/0/0'];
```

A completely different approach is to make the key an element rather than an attribute. For example:

```
var $hash := {
   <interface> {
          <name> "ge-0/0/0";
          <status> "up";
   }
   <interface> {
          <name> "ge-1/0/0";
          <status> "down";
   }
}

<output> "ge-0/0/0 is " _ $hash/interface[name=='ge-0/0/0']/status;
```

As you can see in the above examples, introducing new element names and an element hierarchy changes the Path expression needed to extract the information. In the example where the key is an attribute, the data (status) was simply the value of the element <interface>. In the example where the key is an element (name), the value of status was another element and both name and status are child elements of interface.

Another approach is to use the XSLT <xsl:key> top-level element and key() function. Consider the following example used to parse the Junos routing table. The output of the show route command is:

```
user@junos> show route

inet.0: 4 destinations, 4 routes (4 active, 0 holddown, 0 hidden)
+ = Active Route, - = Last Active, * = Both

0.0.0.0/0          *[Static/5] 00:01:48
                   > to 192.168.2.1 via ge-0/0/2.0
10.10.10.1/32      *[Direct/0] 03:39:02
                   > via lo0.0
192.168.2.0/24     *[Direct/0] 03:38:28
                   > via ge-0/0/2.0
192.168.2.16/32    *[Local/0] 03:38:32
                    Local via ge-0/0/2.0
```

The following is the same output in XML format. Pay close attention to how the script uses the <protocol-name> and the <nh>/<via> values as the upcoming script will use these to produce hash-keys.:

```
user@junos> show route | display xml

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.2R3/junos">
    <route-information xmlns="http://xml.juniper.net/junos/10.2R3/junos-routing">
        <!-- keepalive -->
        <route-table>
            <table-name>inet.0</table-name>
            <destination-count>4</destination-count>
```

```
<total-route-count>4</total-route-count>
<active-route-count>4</active-route-count>
<holddown-route-count>0</holddown-route-count>
<hidden-route-count>0</hidden-route-count>
<rt junos:style="brief">
   <rt-destination>0.0.0.0/0</rt-destination>
   <rt-entry>
      <active-tag>*</active-tag>
      <current-active/>
      <last-active/>
      <protocol-name>Static</protocol-name>
      <preference>5</preference>
      <age junos:seconds="178">00:02:58</age>
      <nh>
         <selected-next-hop/>
         <to>192.168.2.1</to>
         <via>ge-0/0/2.0</via>
      </nh>
   </rt-entry>
</rt>
<rt junos:style="brief">
   <rt-destination>10.10.10.1/32</rt-destination>
   <rt-entry>
      <active-tag>*</active-tag>
      <current-active/>
      <last-active/>
      <protocol-name>Direct</protocol-name>
      <preference>0</preference>
      <age junos:seconds="13212">03:40:12</age>
      <nh>
         <selected-next-hop/>
         <via>lo0.0</via>
      </nh>
   </rt-entry>
</rt>
<rt junos:style="brief">
   <rt-destination>192.168.2.0/24</rt-destination>
   <rt-entry>
      <active-tag>*</active-tag>
      <current-active/>
      <last-active/>
      <protocol-name>Direct</protocol-name>
      <preference>0</preference>
      <age junos:seconds="13178">03:39:38</age>
      <nh>
         <selected-next-hop/>
         <via>ge-0/0/2.0</via>
      </nh>
   </rt-entry>
</rt>
<rt junos:style="brief">
   <rt-destination>192.168.2.16/32</rt-destination>
   <rt-entry>
      <active-tag>*</active-tag>
      <current-active/>
      <last-active/>
      <protocol-name>Local</protocol-name>
      <preference>0</preference>
      <age junos:seconds="13182">03:39:42</age>
      <nh-type>Local</nh-type>
      <nh>
         <nh-local-interface>ge-0/0/2.0</nh-local-interface>
      </nh>
```

```
            </rt-entry>
         </rt>
      </route-table>
   </route-information>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>
```

Now consider the following Junos script that is using the `<xsl:key>` top-level element and `key()` function. The `<xsl:key>` *element* is a top-level declaration that is used to create the cross-reference index used by the `key()` function. In the following code example, there are two keys: one based on the route protocol (`Static`) and the other is the next-hop value. Consider the following complete script:

```
version 1.0;

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

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

/* Define <xsl:key> elements */
<xsl:key name="protocol" match="route-table/rt" use="rt-entry/protocol-name">;
<xsl:key name="next-hop" match="route-table/rt" use="rt-entry/nh/via">;

var $match-protocol = "Static";       /* we want to match on "static" routes */
var $match-interface = "ge-0/0/0.0";   /* we want to match on routes via interface ge-0/0/0.0 */

match / {
   <op-script-results> {
      var $results = jcs:invoke("get-route-information");

      /* Change current node to the $results XML document */
      for-each( $results ) {

         /* Display all static routes */
         <output> $match-protocol _ " routes: ";
         for-each( key( "protocol", $match-protocol ) ) {
            <output> rt-destination;
         }

         /* Display all routes with next-hop of ge-0/0/0.0 */
         <output> "Next-hop " _ $match-interface _ ": ";
         for-each( key( "next-hop", $match-interface ) ) {
            <output> rt-destination;
         }
      }
   }
}
```

The `key()` function is used within the for-each control structures. The node-list that the `key()` function returns is defined by the key element. Both key element definitions return `route-table/rt` node-sets since this was the `match` criteria in the definition. Examining the code snipped:

```
for-each( key( "protocol", $match-protocol ) ) {
  <output> rt-destination;
}
```

The context node in the `for-each` loop returns any `route-table/rt` elements that have a `protocol-name` element with the value equal to "Static". In the above example there is only one route that matches this:

```
<route-table>
   <table-name>inet.0</table-name>
   <destination-count>4</destination-count>
   <total-route-count>4</total-route-count>
   <active-route-count>4</active-route-count>
   <holddown-route-count>0</holddown-route-count>
   <hidden-route-count>0</hidden-route-count>
   <rt junos:style="brief">
      <rt-destination>0.0.0.0/0</rt-destination>
      <rt-entry>
         <active-tag>*</active-tag>
         <current-active/>
         <last-active/>
         <protocol-name>Static</protocol-name>
         <preference>5</preference>
         <age junos:seconds="178">00:02:58</age>
         <nh>
            <selected-next-hop/>
            <to>192.168.2.1</to>
            <via>ge-0/0/2.0</via>
         </nh>
      </rt-entry>
   </rt>
```

And the resulting `rt-destination` value in the for-each loop would be 0.0.0.0/0.

The same for-loop could have been written:

```
for-each( .//rt-entry[protocol-name == 'Static' ) {
   <output> ../rt-destination;
}
```

The key element and functions can be used to reduce processing time if you need to traverse a data-set multiple times. There are also very powerful techniques to index and sort using keys.

As you can see from the many code examples, you have the choice and flexibility to determine the structure of your data and the techniques used to treat node-sets as hash structures. This is one of the key strengths of using SLAX as a programming language; the XML data processing capability is extremely flexible and powerful.

Special Variables

All Scripts

All scripts have the following parameters:

- `$product` - contains the name of the local Junos device model.
- `$user` - contains the name of the user executing the script.
- `$hostname` - contains the local hostname of the Junos device.
- `$script` - contains the name of the script that is currently running.

- `$localtime` - contains the local time when the script was executed using the following example format: Fri Jan 7 14:07:33 2011.
- `$locatime-iso` - contains an ISO format of the local time, for example: 2011-01-08 03:38:57 UTC.

Op scripts

The `$arguments` variable is a special global variable for op scripts that provides the command line argument help to the user.

Consider an op script that has two parameters: interface and admin-control. The code to provide user help would look like the following:

```
/* global parameters - script command-line options*/

param $interface;
param $admin-control = 'disable';

/* global variable to provide command-line option help */

var $arguments = {
   <argument> {
      <name> 'interface';
      <description> 'The interface to control';
   }
   <argument> {
      <name> 'admin-control';
      <description> '[enable | disable], default is disable';
   }
}
```

NOTE Take special notice that the parameter name and the name in the `$arguments` variable must match in name. For example, the first `param` is `$interface`, and the first `<argument>/<name>` is `'interface'`.

NOTE Also notice that the `$admin-control` is assigned a value of `'disable'`. This means that if the user does not provide the `admin-control` command-line option, then the script behaves as if the user did provide the `admin-control` option with the value of `disable`.

When a user selects help (`?`) on the command line, they would see the following:

```
user@junosdev> op change-interface ?

Possible completions:
  <[Enter]>          Execute this command
  <name>             Argument name
  admin-control       [enable | disable], default is disable
  detail             Display detailed output
  interface          The interface to control
  |                 Pipe through a command
```

Event scripts

Starting in Junos 9.0, event policies used to trigger event-based automation can be directly configured into the event script rather than being stored in the Junos configuration file. There are three main advantages to this approach:

- Reduced configuration size: Because the event policy is no longer a part of the configuration file, the event-options configuration hierarchy is smaller, especially when multiple event scripts are in use.
- Easier deployment: By integrating the event policy within the event script, installation becomes a matter of copying the script to the Junos device and enabling the script under event-options. The actual event policy is distributed within the event script and does not have to be configured on each device.
- Consistent event policies: Because the event policy is embedded within the event script, all Junos devices that enable the script share a consistent policy configuration.

The `$event-definition` variable is a special global variable used to embed the event policy.

The following is an example of an embedded event policy definition that would be declared in the event script called `check-var-utilization.slax`:

```
var $event-definition = {
  <event-options> {
    <generate-event> {
      <name> "every-hour";
      <time-interval> "3600";
    }
    <policy> {
      <name> "check-var-utilization";
      <events> "every-hour";
      <then> {
        <event-script> {
          <name> "check-var-utilization.slax";
        }
      }
    }
  }
}
```

MORE? For more details on event scripting and embedding event policies, refer to *This Week: Appling Junos Automation* book at www.juniper.net/dayone.

Junos Context

Starting in Junos 11.1, a new global variable, `$junos-context`, contains information that is common to all scripts as well as information that is specific to the script type. The following presents the XML example formats for each of the script types. Following these examples, each element is defined.

For op scripts:

```
<junos-context>
   <hostname>srx210</hostname>
   <product>srx210h</product>
   <localtime>Wed Aug 18 09:04:30 2010</localtime>
   <localtime-iso>2010-08-18 09:04:30 UTC</localtime-iso>
   <script-type>op</script-type>
   <pid>1643</pid>
   <tty>/dev/ttyp0</tty>
   <chassis>others</chassis>
   <routing-engine-name>re0</routing-engine-name>
   <re-master/>
   <user-context>
```

```
      <user>jnpr</user>
      <class-name>j-super-user</class-name>
      <uid>2001</uid>
      <login-name>jnpr</login-name>
   </user-context>
   <op-context>
      <via-url/>
   </op-context>
</junos-context>
```

For commit scripts:

```
<junos-context>
   <hostname>srx210</hostname>
   <product>srx210h</product>
   <localtime>Wed Aug 18 12:30:20 2010</localtime>
   <localtime-iso>2010-08-18 12:30:20 UTC</localtime-iso>
   <script-type>commit</script-type>
   <pid>3003</pid>
   <tty>/dev/ttyp1</tty>
   <chassis>others</chassis>
   <routing-engine-name>re0</routing-engine-name>
   <re-master/>
   <user-context>
      <user>jnpr</user>
      <class-name>j-super-user-local</class-name>
      <uid>2001</uid>
      <login-name>jnpr</login-name>
   </user-context>
   <commit-context>
      <commit-sync/>
      <commit-confirm/>
      <commit-check/>
      <commit-boot/>
      <commit-comment>Walking and chewing gum</commit-comment>
   </commit-context>
</junos-context>
```

For event scripts:

```
<junos-context>
   <hostname>srx210</hostname>
   <product>srx210h</product>
   <localtime>Wed Aug 18 14:25:00 2010</localtime>
   <localtime-iso>2010-08-18 14:25:00 UTC</localtime-iso>
   <script-type>event</script-type>
   <pid>1587</pid>
   <chassis>others</chassis>
   <routing-engine-name>re0</routing-engine-name>
   <re-master/>
   <user-context>
      <user>root</user>
      <class-name>super-user</class-name>
      <uid>0</uid>
      <login-name>root</login-name>
   </user-context>
</junos-context>
```

The top-level elements:

- `<hostname>` - The system hostname, same as `$hostname`.
- `<product>` - The product, same as `$product`.

- <localtime> - Time that the script was invoked, same as $localtime.
- <localtime-iso> - Time that the script was invoked - in ISO format, same as $localtime-iso.
- <script-type> - op, event, or commit.
- <pid> - process ID number for the cscript process.
- <tty> - TTY of the user's session (if invoked by a user session).
- <chassis> - either TX Matrix (lcc or scc), JCS (psd or rsd), or otherwise it is "others".
- <routing-engine-name> - name of the RE that the script is running on.
- <re-master> - only present if the script is running on the master routing-engine.

The <user-context> elements:

- <user> - local name of user that invoked the script.
- <class-name> - local class of user that invoked the script.
- <uid> - UID of user that invoked the script.
- <login-name> - login name of user that invoked the script. If AAA is in use, then this is the name on the remote authentication server.

The <op-context> element:

- <via-url> - only present if op script was executed via "op url".

The <commit-context> elements:

- <commit-sync> - only present if "commit synchronize" was performed.
- <commit-confirm> - only present if "commit confirmed" was performed.
- <commit-check> - only present if "commit check" was performed.
- <commit-boot> - only present on the initial boot-up commit.
- <commit-comment> - only present if a comment was included for the commit.

Here are some potential ideas that can now be explored, or completed, more efficiently:

- A login script that only allows certain accounts to login at certain times of day: Use $junos-context/tty to accurately logout the user through the <request-logout-user> RPC.
- A login script that offers a menu-based system rather than giving the user console access: Use $junos-context/tty to accurately logout the user through the <request-logout-user> RPC.
- An event script that should only attempt configuration changes on the master routing-engine: Use $junos-context/re-master to determine if the script is running on the master routing-engine or not.
- A commit script that should only run during the boot-up config: Use $junos-context/commit-context/commit-boot to determine if the system is booting or not.
- A commit script that should prevent certain changes based on the user: Use $junos-context/user-context/user to determine the local user name, or $junos-context/user-context/class-name to determine their local class.

- A script that should monitor its memory usage: Use `$junos-context/pid` to determine the correct process ID to pull from "show system processes extensive".
- A script that should kill a user's login session: Use `$junos-context/tty` to accurately logout the user through the `<request-logout-user>` RPC.

Control Statements

If / Then / Else

SLAX supports a "C"/Perl style of the IF/THEN/ELSE construct. The syntax takes the following forms:

```
if( <condition> ) {
   …
}
```

Or:

```
if( <condition> ) {
   …
}
else {
   …
}
```

Or:

```
if( <condition> ) {
}
else if (<condition> ) {
}
else {
}
```

This last example could have multiple `else if` clauses.

The `<condition>` is any valid Boolean expression. Keep in mind that a Boolean expression can be the result of any simple test expression, complex XPath expression, results of function calls (but not templates!), and any combination of the above.

MORE? For details on the specific Boolean test operations, please refer to http://www.w3schools.com/xpath/xpath_operators.asp and the SLAX reference documentation http://code.google.com/p/libslax .

One of the common questions asked is: "If I can only set a variable once, how do I set a variable based on a conditional statement?"

The classic standard code would look something like this (Perl style):

```
#Perl
my $number = 0;

if( $input < 10 ) {
   $number = 2;
}
else {
   $number = 17;
}
```

SLAX style takes advantage of the fact that conditional controls effectively produce expressions or RTFs. The above example would be coded in SLAX as follows:

```
var $number := {
   if( $input < 10 ) {
      expr 2;
   else {
      expr 17;
   }
}
```

NOTE Technically speaking, when making the assignment to `$number`, the equals (=) sign could have been used in place of the colon-equals (:=) sign. This is true only because the variable `$number` is being treated as a simple data type (number) and the conditional expression RTF produced a simple number; that is, not a node-set. If variable assignment is being treated as a node-set, then the colon-equals assignment operation is necessary. Here is such an example:

```
var $mynodes := {
   <color> "Green";
   if( $this < $that ) {
      <color> "Blue";
      <color> "Yellow";
   }
   else {
      <color> "Pink";
      <color> "Purple";
   }
}

for-each( $mynodes/color ) {
   <output> .;
}
```

When the IF clause is true, then the resulting output would be:

```
Green
Blue
Yellow
```

When the ELSE clause is true, then the resulting output would be:

```
Green
Pink
Purple
```

However, if you coded the `$mynodes` with the equals assignment (=) rather than the colon-equals (:=), then the output would produce a runtime error, as shown:

```
error: Invalid type
error: runtime error: file /var/db/scripts/op/test.slax element for-each
error: Failed to evaluate the 'select' expression.
```

The reason for this error is that the `for-each` statement requires a node-set.

For-Each

The `for-each` control statement is a native looping construct in XSLT that allows you to loop through a node-set. The syntax is:

```
for-each( <xpath-expresssion> ) {
   …
```

```
}
```

You've seen an example of using `for-each` in the *If/Then/Else* section:

```
for-each( $mynodes/color ) {
   <output> .;
}
```

The dot value (.) becomes the iterated variable pointing to each of the nodes in the node-set as they are processed by the `for-each` loop. The dot variable is used in the same way as the `$_` variable in Perl. The above could be re-written as:

```
for-each( $mynodes/color ) {
   var $color = .;
   <output> $color;
}
```

The same code as above would be written in Perl as:

```
# perl

foreach (@mynodes) {
   printf $_;
}
```

NOTE For large blocks of code with multiple layers for control, it is recommended to explicitly assign a variable to the dot special variable.

As shown with the `if/then/else`, the `for-each` control can be used to create a variable set of data. Here is an example that extracts a list of ge interfaces, just the name and the IP-address fields.

```
var $ge_interfaces := {
   for-each( $interfaces/interface[starts-with(name, "ge-")] ) {
      var $ifd = .;

      <interface> {
         <name> $ifd/name;
         <ip-addr> $ifd/unit[name==0]/family/inet/address/name;
      }
   }
}
```

NOTE The node-list elements within the `for-each` control are created completely at your discretion. The elements available to the `$ifd` are specific to the node-set presented to the for-each `<xpath expression>`. That is, the choice of using the element name `<interface>` with child elements of `<name>` and `<ip-addr>` are not tied to anything specific. But `$ifd/name` is tied to the fact that the for-each interface node-set has a `name` element and a `unit` element.

Effectively, you are able to dynamically create complex data structures.

```
for-each( $ge-interfaces/interface ) {
   <output> "Interface " _ name _ " has IP-addr of " _ ip-addr;
}
```

Would produce an example output of:

```
Interface ge-0/0/0 has IP-addr of 192.168.10.1/24
Interface ge-0/1/5 has IP-addr of 192.168.20.2/24
Interface ge-7/1/1 has IP-addr of 192.168.30.17/24
```

While

SLAX 1.0 does not have native support for a `while` control structure, but it is available in SLAX 1.1. Until SLAX 1.1 is available in Junos, you must use the

technique of using recursive templates to produce the desired coding results.

MORE? For further details on SLAX 1.1 and the `while` control structure, refer to http://code.google.com/p/libslax.

Consider the following Perl *while loop*:

```
# perl

my $counter = 0;

while( $counter < $SOME-BIG-VALUE ) {
   …
   $counter += $SOME-INC-VALUE;
}
```

The equivalent code in SLAX 1.0 would be:

```
match / {                /* main block of code */

   call do-while-stuff();   /* call the template 'do-while-stuff' */

}

template do-while-stuff( $counter = 0 )
{
   if( $counter < $SOME-BIG-VALUE ) {
      …
      call do-while-stuff( $counter = $counter + SOME-INC-VALUE );      /* recursive template call *
   }
}
```

The use of the recursive template may appear to be a bit clumsy since you cannot simply inline a *while loop* construct in your code.

ALERT! Junos script supports a maximum recursion depth of 3000. Exceeding this causes the script to abort with an error. Also keep in mind that memory allocation for any variables is not released until all of the recursive processing completes.

The recursive template *must* contain two parts: the first is the `if` control at the beginning to ensure that the condition is still met, and second, the recursive call to the template modifying the value of the condition variable.

Do-While

SLAX 1.0 does not have native support for a `do-while` control structure. You must use the recursive template technique to produce the desired coding results.

Compare the standard `do-while` code:

```
# perl

do {
   …
   $status = get-new-status();

} while ($status != "down")
```

With the SLAX equivalent:

```
match /  {

   var $status = get-first-status();
```

```
   call do-while-status-not-down( $status );      /* call the recursive template */
}

template do-while-status-not-down( $status )
{
   …
   var $new-status = get-new-status();

   if( $new-status != "down" ) {
      call do-while-status-not-down( $status = $new-status );
   }
}
```

The primary difference between the `while` control structure and the `do-while` control structure is that placement of the `if` within the recursive template.

ALERT! Keep in mind the recursion limits described in the prior sections.

For

SLAX 1.0 does not have native support for a `for` control structure, but it is available in SLAX 1.1. Until SLAX 1.1 is available in Junos, you must use the technique of using recursive templates to produce the desired coding results.

Compare a typical example for loop code:

```
# perl

my $counter;

for( $counter = 0; $counter < 10; $counter++ ) {
   …
}
```

With the SLAX equivalent:

```
match / {
   call do-ten-times();
}

template do-ten-times( $counter = 0 )
{
   if( $counter < 10 ) {
      …
      call do-ten-times( $counter = $counter + 1 );
   }
}
```

You can see that the for control structure and the while control structure are basically the same technique.

ALERT! Keep in mind the recursion limits described in the prior sections.

Loop Controls

In other programming languages, there exist loop control instructions. For example, Perl offers `next` within a loop to force execution back to next iteration/conditional evaluation.

SLAX does not offer these types of looping control instructions.

Code Modularity

There are a number of ways in which you can create modular code. One of the first ways is to create *import* files – effectively libraries of routines that you can use in multiple scripts. The second way is to separate sections of your code into templates or functions. Templates and functions are akin to subroutines in other languages, but each has specific differences that are covered in upcoming sections.

Importing Files

Import files are a way for you to create code modularity between script files. Typically this technique is used to create your own personal library of routines.

There are two items that you need keep in mind when *creating* import files.

The first is that import files must contain the same boilerplate as any other script file:

```
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";
```

Do take note that the import of the "`junos.xsl`" files is *not* included.

The second item is that you *should* define your own namespace to ensure that the names of your templates and functions do not collide with others. Creating a namespace is very easy using the `ns` keyword. For example, let's say you want to create a namespace for your company called mycorp:

```
ns mycorp = http://xml.mycorp.com/junos;
```

The test string does not need to be a reachable URL; it simply must be a unique character string. You then use the namespace prefix when you declare your templates or functions. For example:

```
template mycorp:get-status() { … }
```

Once you have created your import file, you must do the following in the main script file: first you must declare the same namespace that was used in the import file, and second, import the file using the import keyword. If your import file was called `mycorp-lib.slax`, then the main script would have the following:

```
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";

ns mycorp = "http://xml.mycorp.com/junos"

import "../import/junos.xsl";
import "mycorp-lib.slax";
```

ALERT! The import statement imports files relative to the location of the calling script. If the calling script was located in `/var/run/scripts/op` and the import statement did not include a directory prefix, then the imported file must also be located in `/var/run/scripts/op`.

ALERT! The Junos import file is stored in `/var/run/scripts/import`. This is a read-only directory, and you cannot put your library files here. You could create the directory `/var/run/scripts/lib` and store your import files there. If you do create a separate directory, you must do this as root user, and change the directory permissions so that other users can store their files. Starting in Junos 11.1, the `lib` directory is available by default, and you do not need to create it.

Templates

Named templates are a native capability within SLAX. They are akin to macros in the sense that they are called and expanded. This section highlights the SLAX syntax and usage.

MORE? For more information on template definition and usage, please refer to *This Week: Applying Junos Automation* at http://www.juniper.net/dayone.

The following is an example of a named template:

```
template my-template( $counter = 0, $name )
{
   …
}
```

A named template can take parameters, and each of these parameters can be declared with a default value. In the above example, if the caller does not include the `$counter` parameter, then the value is 0 by default. The `$name` parameter does not have a default value.

A template is expanded using the `call` statement. Parameters are passed by name, meaning that their position in the template call is not important. For example, the following examples are all equivalent:

```
call my-template( $counter = 0, $name = "Bob" );
call my-template( $name = "Bob" );
call my-template( $name = "Bob", $counter = 0 );
```

The other key point regarding templates is that they always produce a Result-Tree-Fragment. For example, the following produce identical results:

```
var $mycolors := {
   <color> "Green";
   <color> "Blue";
   <color> "Yellow";
}
```

Is equivalent to :

```
template make-colors()
{
   <color> "Green";
   <color> "Blue";
   <color> "Yellow";
}

var $mycolors := {
   call make-colors();
```

And …

```
var $list := {
   <item> "Green";
   <item> "Blue";
```

```
   <item> "Yellow";
}

template make-colors-from-list()
{
   for-each( $list/item ) {
      <color> .;
   }
}

var $mycolors := {
   call make-colors-from-list();
}
```

Functions

Functions are not native to SLAX but have been included as part of the Extended XSLT Library (EXSLT). Junos includes the EXSLT functions that are part of the `libxslt` distribution.

MORE? For complete library documentation on EXSLT support, refer to http://www.exslt.org/.

SLAX 1.0 does not have native function syntax support, but this will be included in SLAX 1.1. Until SLAX 1.1 is included in Junos, you will need to code SLAX functions in XSLT.

In order to use functions in SLAX, you must include the `func` namespace declaration at the top of the script file:

```
ns func extension = "http://exslt.org/functions";
```

ALERT! Notice the presence of the extension keyword. You *must* include this keyword when including the EXSLT Functions namespace. A complete listing of EXSLT namespaces and libraries can be found at http://www.exslt.org/.

You must then declare your own namespace, since custom functions are required to have a defined namespace. Using the *mycorp* example, the complete top-of-file would look like the following:

```
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";
ns func extension = "http://exslt.org/functions";

ns mycorp = "http://xml.mycorp.com/junos"

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

You must use XSLT to declare the function since the syntax is not supported in SLAX 1.0. The following function simply increments the provided value:

```
<func:function name="mycorp:incr">
{
   param $num;

   <func:result select="number($num+1)">;
}
```

ALERT! It is important to understand that the `<func:result>` element defines the value that is being returned. It does not, however, stop code execution. This behavior is different than the `return` statement in Perl or C. Also note that the data-type of the return value is explicitly defined. For functions returning node-set values, for example, use `exsl:node-set(…)`

The key benefit with functions is that they can return any type of value, not just an RTF, as is the case with templates. Another difference is that function calls can be included in Boolean expressions, and templates cannot. The following is a valid use:

```
if( mycorp:incr( $somevalue ) < 100 ) {
 …
}
```

MORE? Chapter 3 of this book has additional information on using functions in SLAX.

Using Junos Remote Procedure Calls (RPC)

All Junos script types can execute Junos RPCs. This means that commit scripts, for example, could obtain runtime values via an RPC and update the configuration file based on those results.

Creating an RPC Request

The first step in executing an RPC is creating the XML-based command. The easiest way to determine the RPC for virtually any Junos command is to use the `| display xml rpc` option after a command. For example, to obtain the XML RPC for the command `show chassis hardware detail`:

```
user@junos> show chassis hardware detail | display xml rpc

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.2R3/junos">
    <rpc>
        <get-chassis-inventory>
              <detail/>
        </get-chassis-inventory>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

The portion you need to construct the Junos script code is in the `<rpc>` element. The equivalent command would be coded in SLAX as:

```
var $get-inventory = <get-chassis-inventory> {
    <detail>;
 }
```

Notice how the `$get-inventory` variable is being assigned a Result-Tree-Fragment that is the SLAX representation of the XML RPC.

Atomic Execution of RPC

In order to execute an RPC command, the script needs to *open* a connection to the Junos management daemon, then execute the command, and finally close the management connection. If your script has one or just a few RPCs to execute, you can use the `jcs:invoke()` function to perform the open/execute/close steps atomically.

Using the RPC from the previous section, the code to atomically execute the RPC would be:

```
match / {
   <op-script-results> {
          var $get-inventory = <get-chassis-inventory> {
            <detail>;
          }

          var $result = jcs:invoke( $get-inventory );
     }
}
```

The results of the RPC command are stored in the `$results` variable. The `$results` variable is a node-set, which then allows you to use XPath expression to use the results.

If your RPC does not take any arguments, for example if you simply wanted to execute the `show chassis hardware` command, you do not need to construct the RPC in a variable, but rather just provide the RPC top-level element name to the `jcs:invoke()` routine. For example:

```
match / {
   <op-script-results> {
          var $result = jcs:invoke( 'get-chassis-inventory' );
     }
}
```

You should always check the return variable for an error. This is typically handled by code in the following form:

```
if( $result//self::xnm:error ) {
   /* do error handling */
}
```

MORE? Chapter 3 of this book hasfurther information on error handling techniques.

Executing a Large Number of RPCs

If your scripts need to run a large number of RPCs, then it would be more efficient to explicitly open a connection, execute the large number of RPCs, and then finally close the connection.

The `jcs:open()` routine is used to open a connection to the Junos management daemon. If you call this routine without any parameters, it opens a connection to the local Junos device; therefore, the one running the script.

```
var $connection = jcs:open();
```

You can, however, use this same routine to open a connection to a remote Junos device, and execute RPCs on that device:

```
var $remote-dev = jcs:open( "192.168.17.22", "admin", "admin123" );
```

As you can see from the above form, the first parameter is the hostname or IP-address of the remote device, the next is the user-name, and finally there is the password. Generally speaking you do not want to hardcode passwords into your scripts. You could prompt the user for a password using `jcs:get-secret()`, for example.

The `jcs:execute()` routine is used to execute the RPC once you have an open connection. For example:

```
var $inventory = jcs:execute( $connection, 'get-chassis-inventory' );
var $inventory-detail = jcs:execute( $connection, $get-inventory );
```

The `jcs:close()` routine is used to close the connection to the Junos management daemon. For example:

```
expr jcs:close( $connection );
```

The use of the `expr` keyword here is used to consume the return value, assuming you do not care about the result of the `jcs:close()` routine.

Using the RPC Results

Once you have executed an RPC you need to understand the XML structure of the results so you can access the information accordingly. Recall that the return variable to `jcs:invoke()` and `jcs:execute()` is a node-set data type.

Using the `show chassis hardware` as an example, you can examine the results in XML format using the `| display xml` on the command line. For example:

```
user@junos> show chassis hardware | display xml

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.2R3/junos">
    <chassis-inventory xmlns="http://xml.juniper.net/junos/10.2R3/junos-chassis">
        <chassis junos:style="inventory">
            <name>Chassis</name>
            <serial-number>JN1113BF5ADD</serial-number>
            <description>J2320</description>
            <chassis-module>
                <name>Midplane</name>
                <version>REV 06</version>
                <part-number>710-017558</part-number>
                <serial-number>VG6201</serial-number>
            </chassis-module>
            <chassis-module>
                <name>System IO</name>
                <version>REV 07</version>
                <part-number>710-017562</part-number>
                <serial-number>VG6640</serial-number>
                <description>J23X0 System IO</description>
            </chassis-module>
            <chassis-module>
                <name>Crypto Module</name>
                <description>Crypto Acceleration</description>
            </chassis-module>
            <chassis-module>
                <name>Routing Engine</name>
```

```
            <version>REV 12</version>
            <part-number>710-017560</part-number>
            <serial-number>VG6966</serial-number>
            <description>RE-J2320-2000</description>
         </chassis-module>
         <chassis-module>
            <name>FPC 0</name>
            <description>FPC</description>
            <chassis-sub-module>
               <name>PIC 0</name>
               <description>4x GE Base PIC</description>
            </chassis-sub-module>
         </chassis-module>
         <chassis-module>
            <name>FPC 3</name>
            <version>REV 13</version>
            <part-number>750-015153</part-number>
            <serial-number>VG9667</serial-number>
            <description>FPC</description>
            <chassis-sub-module>
               <name>PIC 0</name>
               <description>8x GE uPIM</description>
            </chassis-sub-module>
         </chassis-module>
         <chassis-module>
            <name>Power Supply 0</name>
         </chassis-module>
      </chassis>
   </chassis-inventory>
   <cli>
      <banner></banner>
   </cli>
</rpc-reply>
```

The result variable is *rooted* at the element immediately following the `<rpc-reply>` element; in this example, that would be `<chassis-inventory>`.

For example, if you wanted to loop through each of the `<chassis-module>` elements displaying only those with serial numbers, the code would be:

```
match / {
  <op-script-results> {

         var $inventory = jcs:invoke( 'get-chassis-inventory' );

         for-each( $inventory/chassis/chassis-module[serial-number] ) {
           <output> "Module " _ name _ ", serial number: " _ serial-number;
         }
  }
}
```

This script would produce the following output:

```
Module Midplane, serial number: VG6201
Module System IO, serial number: VG6640
Module Routing Engine, serial number: VG6966
Module FPC 3, serial number: VG9667
```

ALERT! Mastering XPath expressions is the key to mastering Junos automation scripting. Without a good understanding of XPath and XSLT processing, the `for-each` loop in the above example might be difficult to understand.

Console Input / Output

Using jcs:get-input() and jcs:get-secret()

Both `jcs:get-input()` and `jcs:get-secret()` can be used in op scripts to obtain console input from the user. The `jcs:get-input()` routine echoes the text that the user enters, while the `jcs:get-secret()` does not echo the text.

Consider the following example op script that is used to remotely login to another Junos device and retrieve the software version:

```
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 $remote-host = jcs:get-input( "Enter host-name: " );
      var $user-name = jcs:get-input( "Enter user-name: " );
      var $password = jcs:get-secret( "Enter password: " );

      /* Open the remote connection */
      var $connection = jcs:open( $remote-host, $user-name, $password );

      if( $connection ) {
         var $result = jcs:execute( $connection, "get-software-information" );

         /* Grab the version - This path should work on both single and multi-re systems */
         var $version =
            $result/..//software-information[1]/package-information[name == "junos"];
         expr jcs:output( "Junos version on remote host: ", $version/comment );
      }
      else {
         expr jcs:output( "Unable to open a connection." );
      }
   }
}
```

Understanding <output> Versus jcs:output() Versus jcs:printf()

Both `<output>` and `jcs:output()` can be used by op scripts to send strings to the console.

The `<output>` elements are consumed by Junos as part of the result-tree, that is, at the completion of the Junos script. When you use `jcs:output()`, the output is immediately sent to the console. Consider the following code:

```
match / {
  <op-script-results> {
    <output> "One";
    <output> "Two";
    expr jcs:output( "Three" );
    expr jcs:output( "Four" );
    <output> "Five";
  }
}
```

The actual console output will be:

```
Three
Four
One
Two
Five
```

ALERT! You cannot use <output> or jcs:output() from a commit script. You can use these with event scripts if you also configure an output file in the event policy definition because it always signals a commit failure to Junos, which halts the requested commit.

The jcs:printf() function does not actually send a string to the console. Rather it creates a formatted string, which can then be output to the console (or file, or other variable, etc.)

The following code example uses jcs:printf() to display the op script global parameters:

```
match / {
   <op-script-results> {

      /* Display the parameters */
      expr jcs:output( jcs:printf( "%15s %-10s", "Parameter", "Value" ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$user", $user ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$hostname", $hostname ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$script", $script ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$localtime", $localtime ) );

      var $string = jcs:get-input( "Enter string: " );
      var $width = jcs:get-input( "Enter width: " );
      var $precision = jcs:get-input( "Enter precision: " );
      expr jcs:output( jcs:printf( "|%*.*s|", $width, $precision, $string ) );
   }
}
```

MORE? For details on the jcs:printf() formatting controls, please refer to the Appendix.

Using <xsl:message>

The <xsl:message> element is used to produce an immediate error message. For example:

```
<xsl:message> "Unable to create connection"
```

would produce the following console output:

```
error: Unable to create connection
```

ALERT! The error message is sent to the standard error (STDERR) device. You should be careful when using this element in commit scripts

One feature of the <xsl:message> element is that it can be used to immediately terminate the script. This produces a similar result as *exit* or *die* routines in other languages. To do this, include the terminate="yes" attribute. For example:

```
<xsl:message terminate="yes"> "No connection, aborting script now"
```

Using jcs:progress()

The `jcs:progress()` routine is used for debugging purposes. When you run op scripts you can include a `detail` option. If you use the `detail` option, then the output from `jcs:progress()` is included in the detailed output. If you do not include the detail option, then the `jcs:progress()` output is not displayed. If you do have traceoptions enabled, however, the strings are logged to the traceoptions file regardless of if the `detail` option is used.

For example:

```
match / {
   <op-script-results> {
      <output> "One";
      <output> "Two";
      expr jcs:progress( "Starting jcs:output" );
      expr jcs:output( "Three" );
      expr jcs:output( "Four" );
      <output> "Five";
   }
}
```

When invoking the script with the detail command:

```
user@junos> op test detail
2011-03-02 04:10:56 UTC: running op script 'test.slax'
2011-03-02 04:10:56 UTC: opening op script '/var/db/scripts/op/test.slax'
2011-03-02 04:10:56 UTC: reading op script 'test.slax'
2011-03-02 04:10:56 UTC: Starting jcs:output
Three
Four
2011-03-02 04:10:56 UTC: inspecting op output 'test.slax'
One
Two
Five
2011-03-02 04:10:56 UTC: finished op script 'test.slax'
```

Using <xnm:warning> and <xnm:error> for Commit Scripts

There are special element tags for producing output from commit scripts.

The `<xnm:warning>` element can be used to output a warning message, for example:

```
match configuration {
          <xnm:warning> {
            <message> "This is a test";
          }
}
```

would produce the following output:

```
user@junos# commit
warning: This is a test
configuration check succeeds
```

The `<xnm:error>` element can be used to output an error message, and then signals Junos to prevent the candidate configuration from becoming active. For example:

```
match configuration {
   <xnm:error> {
      <message> "Don't let this commit succeed";
   }
}
```

would produce the following output:

```
user@junos# commit
error: Don't let this commit succeed
error: 1 error reported by commit scripts
error: commit script failure
```

NOTE The <xnm:warning> and <xnm:error> elements are part of the commit script result tree. This means that your code can check for the existence of these warning/errors. For example, if you have an op script that makes a configuration change, your script should look for the presence of <xnm:error> elements in the result to determine if the commit succeeded or failed.

MORE? For additional information on commit scripts, please refer to *This Week: Applying Junos Automation* at http://www.juniper.net/dayone.

Storage Input / Output

Reading from Files

Your Junos scripts can read and write data files from the local file system.

One of the simplest ways to read an XML file that is stored on the local file system is using the XSLT `document()` function. The following example illustrates the use of `document()`:

```
match / {
   <op-script-results> {

      /* Load document into script */
      var $chassis-info = document("/var/home/jnpr/chassis-inventory.xml");

      /*
       * Retrieve chassis name, use local-name() function to avoid
       * dealing with namespaces
       */

      var $chassis = $chassis-info//*[local-name() == "chassis"];
      var $chassis-name = $chassis/*[local-name() == "name"];

      <output> "Chassis name is " _ $chassis-name;
   }
}
```

If the file is not XML, you can use the Junos <file-get> RPC and process each line using a `jcs:break-lines()` loop. Assume there is a file with the following contents:

```
Jones, Bob, 35, Male
Lee, Sandra, 82, Female
Baker, Lee, 19, Male
Lewis, Alison, 38, Female
```

The following script would read this file and display each line of data.

```
match / {
  <op-script-results> {
         var $rpc = <file-get> {
```

```
          <filename> "/var/tmp/people.csv";
          <encoding> "ascii";
        }

        var $people = jcs:invoke( $rpc );
        if( $people//self::xnm:error ) {
          expr jcs:output( "Unable to open file: ", $people//self::xnm:error/message );
          <xsl:message terminate="yes">;
        }

        var $people_data = jcs:break-lines( $people/file-contents );

        for-each( $people/* ) {
          var $line = .;
          <output> $line;
    }
  }
}
```

If the file did not exist, for example, the following would be displayed:

```
Unable to open file:
Failed to open file (/var/tmp/people.csv): No such file or directory
```

Writing to Files

There are a number of options available when writing to files. If you are writing a non XML-based file, you could use the Junos `<file-put>` RPC, the `<exsl:document>` element, or the `<redirect:write>` element.

The primary benefit of using `<file-put>` is you can control the file permissions when writing the file. The primary downside of using `<file-put>` is that you cannot use it to write XML documents.

The following example retrieves the system section of the configuration file in text (hierarchal tree) and stores it to a local file.

```
match / {
  <op-script-results> {

        var $show-config-system = <get-configuration format="text"> {
          <configuration> {
            <system>;
          }
        }

        var $config = jcs:invoke( $show-config-system );

        var $writing = <file-put> {
          <filename>"/var/tmp/system-config.txt";
          <encoding> "ascii";
          <permission> "644";
          <delete-if-exist>;
          <file-contents> $config;
        }

        var $result-writing = jcs:invoke( $writing );
  }
}
```

For writing ASCII files or XML documents, you can use either of the two extension

elements.

The primary benefit of using <exsl:document> is the flexibility of options that it offers. The downside is that you cannot control the permissions and the file owner is set to *nobody*.

When you use this element, you must include the exsl namespace and extension definition as shown in the following complete script example:

```
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";
ns exsl extension = "http://exslt.org/common";

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

match / {
   <op-script-results> {
          var $rpc = <get-chassis-inventory> { <detail>; }

          var $inventory = jcs:invoke( $rpc );

          <exsl:document href="/var/tmp/inventory.xml"> {
            copy-of $inventory;
          }
   }
}
```

MORE? For details on the <exsl:document> element, please refer to http://www.exslt.org/exsl/elements/document/index.html.

The primary benefit of using <redirect:write> is that you can append data to an existing file as of Junos 11.1.

The downside is that you cannot control the permissions and the file owner is set to *nobody*.

When you use this element, you must include the redirect namespace and extension definition as shown in the following complete script example:

```
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";
ns redirect extension = "org.apache.xalan.xslt.extensions.Redirect";

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

match / {
   <op-script-results> {
          var $rpc = <get-chassis-inventory> { <detail>; }

          var $inventory = jcs:invoke( $rpc );

          <redirect:write file="/var/tmp/inventory.xml"> {
            copy-of $inventory;
          }
   }
}
```

MORE? For details on the <redirect:write> element, please refer to http://xml.apache.org/xalan-j/extensionslib.html.

MORE? For further information on using File I/O in Junos script, please refer Chapter 3.

Writing to the Syslog File

All Junos scripts can write to the syslog (/var/log/messages) file using the jcs:syslog() routine. The jcs:syslog() routine is a variable argument routine with the first argument being the facility/severity level indication and the remaining parameters being a comma separated list of strings to include. For example:

```
match / {
   <op-script-results> {

      expr jcs:syslog( "external.info", "Example syslog message: \"Insert message here\"" );
      expr jcs:syslog( "daemon.debug", "This message is from ", $script );
      expr jcs:syslog( 12, "This is a ", "user\\", "warning." );
   }
}
```

Writing to the Traceoptions File

You can configure Junos to enable traceoptions for each of the different script types. The jcs:trace() routine takes a comma-separated list of strings and writes to the traceoptions file.

For example, if your Junos configuration included the following:

```
user@junos# show system scripts op
traceoptions {
   file opscripts;
   flag input;
   flag rpc;
}
```

And your script included a call to jcs:trace(), for example:

```
expr jcs:trace("This is a traceoptions message from script: ", $script);
```

Then you could see the message logged to the file opscripts:

```
user@junosjeremy@j2320-1> show log opscripts
Mar 4 06:32:13 j2320-1 clear-log[3112]: logfile cleared
Mar  4 06:32:14 complete script processing begins
Mar  4 06:32:14 opening op script '/var/db/scripts/op/test.slax'
Mar  4 06:32:14 reading op script 'test.slax'
Mar  4 06:32:15 op script processing begins
Mar  4 06:32:15 running op script 'test.slax'
Mar  4 06:32:15 opening op script '/var/db/scripts/op/test.slax'
Mar  4 06:32:15 reading op script 'test.slax'
Mar  4 06:32:15 This is a traceoptions message from script: test.slax
Mar  4 06:32:15 op script output
Mar  4 06:32:15 begin dump
<?xml version="1.0"?>
<op-script-results xmlns:junos="http://xml.juniper.net/junos/*/junos" xmlns:xnm="http://xml.
juniper.net/xnm/1.1/xnm" xmlns:jcs="http://xml.juniper.net/junos/commit-scripts/1.0"/>
Mar  4 06:32:15 end dump
```

```
Mar  4 06:32:15 inspecting op output 'test.slax'
Mar  4 06:32:15 finished op script 'test.slax'
```

"Scratch Pad Memory"

Junos scripts can read and write information into the Junos Utility MIB. This is a handy technique for sharing data between scripts, as well as making data accessible for SNMP- based management systems.

The following types of data can be stored in the Utility MIB:

- String
- Integer
- Unsigned Integer
- Counter
- Counter64

When you store a value in a Utility MIB, there is also a timestamp record of when the value was last written.

The following Junos RPCs are used to utilize the Utility MIB:

- `<request-snmp-utility-mib-set>` is used to create/set a MIB variable and update the associate timestamp.
- `<get-snmp-object>` is used to retrieve the value of any SNMP MIB, inclusive of Utility MIB values.
- `<request-snmp-utility-mib-clear>` is used to delete a Utility MIB variable and its associated timestamp value.

ALERT! Values that are stored in the SNMP Utility MIB are local only to the routing-engine where the script is running. You may need to take this into consideration when writing scripts for systems equipped with redundant routing-engines.

MORE? Junos script examples for using the Utility MIB can be found Chapter 3.

Summary

This chapter has covered the core SLAX programming language concepts that are similar to other common scripting languages. You should experiment with the examples from this chapter and get more familiar with the SLAX programming paradigm. After just a bit of practice you will have grasped the fundamentals of writing Junos automation scripts. The next chapter takes it up a notch, covering many "tips and tricks" as well as advanced topics that can help you master your automation activities.

Chapter 3

Essential SLAX Topics to Know

This chapter covers several topics that you need to know to be productive and creative with Junos automation scripting. The need to know items are arranged into four buckets that will make more sense when you're done with the chapter than when you're just beginning:

- Fundamentals
- File and storage
- OSS integration
- Event Script

Write notes in the margins, use the on-screen highlighter in Acrobat, or try the eBook tools in your eBook reader — do whatever works best to make notes about the many essential SLAX features and knobs that can make Junos sing.

Fundamental Topics

First, let's cover some fundamental SLAX topics that you will use as a foundation for future scripting. They include searching for error elements, learning complex XPath expressions, using custom functions, learning all you can about the jcs: functions, and displaying output from Junos commands.

Searching for Error Elements (<xnm:error>)

The `<xnm:error>` element is returned by Junos to indicate that an error occurred. It is important to check for this element whenever the operation of a script could result in an error and the script is expected to react to it. For example, anytime an op or event script performs a commit, it should be ready to handle the various potential errors, such as a database locking error, a configuration load error, or any kind of commit error.

Proper care must be taken when building the location path to ensure that when an error occurs it is actually retrieved. The `<xnm:error>` element is in a different location relative to the node-set's context, depending on whether your variable is a converted result tree fragment or was returned as a node-set.

The difference exists because a result tree fragment is converted to a node-set by placing the entire XML structure under a single root node and creating a node-set that contains that root-node. For example, the following results in a tree fragment:

```
<parent> {
   <child> "text";
}
```

And would be converted into this XML document:

```
/ {
   <parent> {
      <child> "text";
   }
}
```

And the node-set variable would consist of a single node: the root node of the XML document.

Node-sets returned by functions are different, however. These returned node-sets have as their context node the child of `<rpc-reply>`, rather than the root node of the XML document.

For example, here are the XML results of the `show version` command:

```
user@junosjnpr@srx210> show version | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4D0/junos">
    <software-information>
        <host-name>srx210</host-name>
        <product-model>srx210h</product-model>
        <product-name>srx210h</product-name>
        <jsr/>
        <package-information>
            <name>junos</name>
            <comment>JUNOS Software Release [10.4-20100719_ib4_11_1.0]</comment>
        </package-information>
    </software-information>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

The node-set returned in response to the `<get-software-information>` API element and would have a single node of `<software-information>`. Why does this matter? Because your node-set might consist of a `/ node`, which is a parent to the `<xnm:error>` node:

```
/ {
    <xnm:error> {
        <message> "jcs:load-configuration called with invalid parameters";
    }
}
```

Or, the node might be the actual `<xnm:error>` node:

```
<xnm:error> {
    <token> "This-Is-The-Bad-API-Element-I-Used";
    <message> "syntax error";
}
```

Or, the node might be an element node, which is a parent or ancestor of the `<xnm:error>` node:

```
<commit-results> {
    <routing-engine junos:style='normal'> {
        <name> "re0";
        <xnm:error> {
            <message> "\nssh is not enabled\n";
        }
        <xnm:error> {
            <message> "\n1 error reported by commit scripts\n";
        }
        <xnm:error> {
            <message> "\ncommit script failure\n";
        }
    }
}
```

The first error message is the result of an invalid call to the `jcs:load-configuration()` template. The second error message was generated by sending an invalid API element to `jcs:invoke()`. And the third error message was generated by attempting to perform a commit via `jcs:execute()` that resulted in a commit failure.

Converted Result Tree Fragments

The `jcs:load-configuration()` template is a common source of result tree fragments that need to be parsed for `<xnm:error>` elements. If the `$results` variable captured the result of `jcs:load-configuration()` then the following location path can be used to locate a `<xnm:error>` within the `$results` variable:

```
$results//xnm:error
```

The `//` is a shortcut for `/descendant-or-self::node()/`, and the default axis is the child axis, so the full location path is actually:

```
$results/descendant-or-self::node()/child::xnm:error
```

This location path works great for converted result tree fragments because the node-set's context node is always the root node of the XML document, so any nodes of interest are always children of the node rather than the node itself.

Node-sets

Using `$results//xnm:error` does not always work with node-sets, however, because `<xnm:error>` is often the actual node-set's context node. The `$results//xnm:error` location path only works if `<xnm:error>` is the child of the context node, not if it is the context node itself. Converted result tree fragments are always set to the root node, but returned node-sets are set to an actual element node. Because of this, it's necessary to move back to the context node's parent before attempting to use the `//` location path operator:

```
$results/..//xnm:error
```

The `..` operator is short for `parent::node()`, so the full path is actually:

```
$results/parent::node()/descendant-or-self::node()/child::xnm:error
```

This works whether the `<xnm:error>` message is the context node or a descendant of the context node, so it is appropriate to use with a node-set variable; however, it doesn't work with a converted result tree fragment because there is no parent of the root node so the `parent::node()` in the path results in an empty result.

A Location Path for All Cases

A location path that works for all converted result tree fragments:

```
$results//xnm:error
```

And this works for returned node-sets:

```
$results/..//xnm:error
```

But isn't there a location path that could be used for both? Actually, there is. Recall that the problem with using the // operator is that the default axis is the child axis. This prevented us from using the // with a node-set because the <xnm:error> might be the context node rather than the child of the context node, but using the self axis rather than the child axis avoids this problem.

The following location path works for both converted result tree fragments as well as returned node-sets, whether the `<xnm:error>` node is the context node or not:

```
$results//self::xnm:error
```

And here it is expanded:

```
$results/descendant-or-self::node()/self::xnm:error
```

This location path always works because the descendant-or-self axis returns all nodes that are either the context node or its descendants, and the self axis returns the current context node. So whether <xnm:error> is the context node, or is a descendant, it will be selected, making this location path ideal to use when searching for a <xnm::error> element in your script results.

Complex XPath Expressions

For the first example, the goal is to retrieve the <interface> nodes of all interfaces with names that begin with *fe* and that do not have a logical unit 100.

Predicate Inside a Predicate

The name restriction is easy to accomplish through the use of a standard predicate:

```
$configuration/interfaces/interface[ starts-with( name, “fe” ) ]
```

However, the second restriction requires a more advanced design. The context node at this point is <interface>, so the objective is to devise a boolean expression that will return true if there are no child <unit> nodes that have a <name> of 100. This can be achieved by using the jcs:empty() function, but the argument for that function is a location path, and this location path itself requires a predicate, which provides the complete example of a location path that uses a predicate within a predicate:

```
$configuration/interfaces/interface[starts-with(name, “fe”)][jcs:empty(unit[name == “100”])];
```

This location path is processed by first looking for any <interfaces> child nodes, then looking for any <interface> child nodes. The <interface> nodes are subjected to two predicates. First, their <name> child element is compared to the fe string to determine if the text contents of <name> begin with fe. If that evaluates to *true* then the second predicate is checked, which provides a location path to the jcs:empty() function. This location path looks for any child <unit> nodes, but the returned <unit> nodes are themselves subjected to a predicate before being provided as input to jcs:empty(). This inner predicate checks for any <name> child nodes with a value of 100. If one is present then the unit node is provided to jcs:empty(), which returns *false* because the node-set argument is not empty. If no <name> nodes with values of 100 for the <unit> nodes are present, then an empty node-set is provided to jcs:empty(), which returns *true*, thereby causing the outer predicate to evaluate to *true* and the <interface> node to be included in the resulting node-set.

Preceding Sibling Nodes

The preceding-sibling axis is not used as much as the parent, child, and descendant axes, but it can prove valuable when it is necessary to match nodes at specific locations relative to the context node.

As an example, assume that a script needs to identify which op scripts have been annotated with a comment. Here is the relevant XML structure that must be parsed:

```
<configuration>
      <system>
         <scripts>
```

```
            <op>
               <junos:comment>/* Colossal Cave Adventure */</junos:comment>
               <file>
                  <name>adventure.slax</name>
               </file>
               <file>
                  <name>commit-script-builder.slax</name>
               </file>
               <junos:comment>/* Test script */</junos:comment>
               <file>
                  <name>test.slax</name>
               </file>
            </op>
         </scripts>
      </system>
</configuration>
```

As shown here, there is no hierarchical relationship between a comment and its annotated statement. It is instead *the order* that matters, because the <junos:comment> node annotates the element that follows.

Therefore, a location path that is designed to retrieve any commented op scripts must look for <file> nodes within the <op> hierarchy that have an immediately preceding <junos:comment> node. This can be achieved through the following location path:

```
$configuration/system/scripts/op/file[ preceding-sibling::*[1][self::comment] ]
```

When this location path processes the <file> nodes, they are subjected to a single predicate before being included in the resulting node-set. This predicate does a location path search along the preceding-sibling axis using a wildcard node test to ensure that all siblings are returned, but it subjects them to two inner predicates. First, it only selects the first sibling, matching only the node immediately prior to the <file> node being considered. Next, it checks to see if that first sibling is a <comment> node. (The self::comment check is equivalent to name() == "comment"). If this is the case, then the immediately preceding node is a <comment> node, which causes the predicate evaluates to *true*, and the <file> node under comparison is included as part of the returned node-set.

ALERT! Note that there is a discrepancy between the XML structure shown here and the location path, because the XML structure shown here had the node shown as <junos:comment>, whereas the location path referred to it as <comment>. The reason for this difference relates to the default namespace stripping that Junos performs on element nodes. Before the configuration is provided to the op script, the junos namespace has been stripped, transforming <junos:comment> into <comment>.

Checking Preceding Nodes

The next example of a complex XPath expression considers how to look through preceding nodes to see if the current node is the first of its kind. This could be useful if a script needs to perform a task on an IGMP group, as seen in the show igmp group command, but you only wish to perform the task once per group.

Here is the XML data structure in question:

```
<igmp-group-information>
   <mgm-interface-groups>
      <interface-name>fe-0/0/3.0</interface-name>
      <mgm-group-count>1</mgm-group-count>
      <mgm-group junos:style="igmp">
```

```
            <multicast-group-address>227.1.1.2</multicast-group-address>
            <multicast-source-address>0.0.0.0</multicast-source-address>
            <last-address>Local</last-address>
            <mgm-timeout>0</mgm-timeout>
            <mgm-type>Static</mgm-type>
        </mgm-group>
    </mgm-interface-groups>
    <mgm-interface-groups>
        <interface-name>ge-0/0/0.0</interface-name>
        <mgm-group-count>1</mgm-group-count>
        <mgm-group junos:style="igmp">
            <multicast-group-address>225.0.0.1</multicast-group-address>
            <multicast-source-address>0.0.0.0</multicast-source-address>
            <last-address>Local</last-address>
            <mgm-timeout>0</mgm-timeout>
            <mgm-type>Static</mgm-type>
        </mgm-group>
    </mgm-interface-groups>
    <mgm-interface-groups>
        <interface-name>fe-0/0/2.0</interface-name>
        <mgm-group-count>2</mgm-group-count>
        <mgm-group junos:style="igmp">
            <multicast-group-address>225.0.0.1</multicast-group-address>
            <multicast-source-address>0.0.0.0</multicast-source-address>
            <last-address>Local</last-address>
            <mgm-timeout>0</mgm-timeout>
            <mgm-type>Static</mgm-type>
        </mgm-group>
        <mgm-group junos:style="igmp">
            <multicast-group-address>226.1.1.1</multicast-group-address>
            <multicast-source-address>0.0.0.0</multicast-source-address>
            <last-address>Local</last-address>
            <mgm-timeout>0</mgm-timeout>
            <mgm-type>Static</mgm-type>
        </mgm-group>
    </mgm-interface-groups>
</igmp-group-information>
```

The challenge is to identify if a `<multicast-group-address>` node is the first occurrence of a particular group, and one way to do this works using two location paths. The first location-path is used to determine the nodes the for-each loop should process:

```
for-each( $results/mgm-interface-groups/mgm-group/multicast-group-address )
```

As shown here, this loop processes each `<multicast-group-address>` node, and the rest of the code is included within the for-each loop's code block.

The next step is to record the current address into a variable so that it can be referenced in the second location path:

```
var $address = .;
```

The second location path uses the preceding axis to look at all the prior `<multicast-group-address>` nodes within the XML document, and it checks if the value of the preceding node is equal to the value recorded in the `$address` variable. In this example, the resulting node-set is provided as an argument to `jcs:empty()`, so this could be performed within an if statement that should only be executed when this is the first occurrence of the particular multicast group:

```
if( jcs:empty( preceding::multicast-group-address[ . == $address ] ) )
```

The combined code example:

```
for-each($results/mgm-interface-groups/mgm-group/multicast-group-address) {
   var $address = .;
   if( jcs:empty( preceding::multicast-group-address[ . == $address ])) {
      /* actions that should only be performed once per group */
   }
}
```

Using Custom Functions

Templates are a familiar topic for most SLAX coders. It is through templates that large scripts are modularized and recursive tasks are performed. In addition to templates, functions greatly expand the capability and usefulness of Junos Automation with their range of uses, from counting the number of characters in a string with the `string-length()` function, to performing Junos API requests through the `jcs:invoke()` function.

Functions Versus Templates

A less familiar topic is the ability to replace custom templates with custom functions. This capability was released as part of Junos 9.4, when support for EXSLT functions and elements was added. Within EXSLT is an element called <*func:function*>. Using this element allows custom functions to be created and then used within SLAX scripts in the same manner as the standard functions available to SLAX scripts. Using custom functions instead of templates provides the following advantages:

- Functions can return all types of data values: boolean, string, number, node-set, or result tree fragment.
- Function results can be assigned directly to variables without requiring clunky syntax.
- Function calls can be performed within the argument list of other function calls, or within location path predicates.

These advantages allow for a more compact and elegant script. Instead of writing this:

```
var $true = "true";
var $false = "false";
var $upper-case = { call is-upper-case( $string = $input-string ); }
var $count = { call count-substring( $string = $input-string, $substring = "GE-" ); }
if( $upper-case == $true && $count > 5 ) {
...
```

...you can write this:

```
 if( st:is-upper-case($input-string) && st:count-substring($input-string, "GE-") > 5 ) {
...
```

There are, however, a few disadvantages to using custom SLAX functions that you should know about:

- Custom SLAX functions require Junos 9.4, whereas templates were supported when SLAX was first developed.
- The function definition syntax is not SLAX-ified yet in SLAX 1.0 (but will be in SLAX 1.1), meaning the function header will not say `function st:is-upper-case()`, it will say `<func:function name="st:is-upper-case">`.

- Most <func:function> error messages are not intuitive, and in one circumstance the error message is not even reported to the console.

The good news is that these disadvantages will disappear with time. Junos 9.4 was released in 1Q09, and as more time passes, more and more networks will have moved beyond this version, clearing the way for using custom functions. And Junos developers are considering enhancements to SLAX, both to create SLAX statements for function definition, as well as to improve error messages for <func:function> problems.

Namespaces

Before a custom function can be used, two namespaces must be declared within the script's header, in addition to the three default boilerplate namespaces that every script must contain. The first namespace is the EXSLT function namespace:

```
ns func extension = "http://exslt.org/functions";
```

Using func as the namespace prefix is not required, but it is highly recommended in order to match the example code and to follow the common convention. Also note the presence of the extension keyword – that's necessary so Junos treats <func:function> and <func:result> as extension elements.

The second namespace that must be created is a unique namespace that will be assigned to the custom functions.

ALERT! Unlike templates, it is not possible to create a custom function that lacks a namespace.

The assigned namespace can be given any unique value, but the appropriate method would be to base it on a URL possessed by you or your organization, so that no conflicts arise between namespaces used by different script writers.

Here is an example of a custom namespace. This namespace is used by the Colossal Cave Adventure SLAX script:

```
ns adv = "http://xml.juniper.net/adventure";
```

Function Definition

Custom functions are defined by enclosing the code contents within a <func:function> element. The name of the function is assigned to the name attribute of the <func:function> element:

```
<func:function name="test:example-function">
{
   expr jcs:output( "This is an example of a function" );
}
```

Once defined, this function can now be called in the same manner as a standard SLAX function:

```
match / {
   expr test:example-function();
}
```

Function Arguments

Function arguments are defined by including `param` statements at the top of the function code block. Multiple `param` statements can be included, and their order dictates the order of the arguments passed to the function. Default values can be provided for function arguments in the exact same way as with template parameters:

```
<func:function name="test:show-string">
{
   param $string;

   expr jcs:output( "Here is the string: ", $string );
}
```

It's legal to call a function without including all of its arguments, but when this occurs the default values are assigned to the non-included arguments (or an empty string if no default value is provided).It is an error, however, to call a function with *too many* arguments.

Function Results

One of the advantages of functions is their ability to return results of any data type. Templates can only return result tree fragments, but functions can also return booleans, numbers, strings, and node-sets. This is accomplished by assigning the desired return value to the `<func:result>` element by using its select attribute:

```
<func:function name="test:is-odd">
{
   param $number;

   if( math:abs( $number ) mod 2 == 1 ) {
      <func:result select="true()">;
   }
   else {
      <func:result select="false()">;
   }
}
```

ALERT! Note that `<func:result>` does not terminate the function, it only assigns the value to return when the function ends. It is an error to use `<func:result>` more than once in the same code path. For example, you cannot assign `<func:result>` to a default value and then later assign it to a more specific value.

ALERT! Do not write to the result tree within a function. It is invalid to do so, and it results in early script termination. Worse, prior to Junos 11.2, no error is shown to the script user.

Full Example

Here is an example of an op script that includes a custom function to convert a string 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";
ns func extension = "http://exslt.org/functions";
```

```
ns test = "http://xml.juniper.net/test";

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

match / {
    expr jcs:output( "abcd = ", test:to-upper-case( "abcd" ) );
}

<func:function name="test:to-upper-case">
{
    param $string;

    var $upper-case = translate( $string, 'abcdefghijklmnopqrstuvwxyz',
                                          'ABCDEFGHIJKLMNOPQRSTUVWXYZ' );

    <func:result select="$upper-case">;
}
```

When to Use jcs:invoke() and jcs:execute() Functions

Both the `jcs:invoke()` and `jcs:execute()` functions are used by Junos scripts to interact with the XML API. The `jcs:invoke()` function was introduced at the same time as commit scripts. By using it scripts were able to send the requested XML API element to Junos, which would execute the desired action and return the results.

Here is an example of a `jcs:invoke()` call, which returns the current software version information:

```
var $result = jcs:invoke( "get-software-information" );
```

This example illustrates how the name of the desired API element can be provided as a string to `jcs:invoke()`; however, if any child elements or attributes are necessary, then a result tree fragment variable can be created instead, allowing the API element to be provided in XML format rather than simply as a string name:

```
var $config-rpc = {
    <get-configuration database="committed"> {
        <configuration> {
            <interfaces>;
        }
    }
}

var $result = jcs:invoke( $config-rpc );
```

The `jcs:invoke()` function is atomic: it first creates a new Junoscript session and then invokes the desired API element within that session. Once the action has completed and the results are retrieved, the session is closed and the results are returned to the calling script as a node-set. In the preceding example, these results are assigned to the variable `$result`.

The script, however, proved insufficient when attempting to change the configuration from an op or event script, because safely changing the configuration involves first locking the configuration database, then loading the configuration changes, then committing the configuration, and finally releasing the configuration database lock. In other words, proper configuration changes require four API calls, but the `jcs:invoke()` function was only designed to process one request per Junoscript session. It isn't possible to lock the database in one `jcs:invoke()` call and then load

the changes within the locked session in the next, because each function call operates within an entirely separate Junoscript session, and the configuration lock is tied to the session, not to the script.

Because of this deficiency, a new approach was released in Junos 9.3: the `jcs:execute()` function. This function works identically to `jcs:invoke()` as far as its API input and output; however, it differs in that it does not create a Junoscript session. Instead, a session must first be opened through the `jcs:open()` function, stored in a variable, and then that variable must be provided to the `jcs:execute()` function so it knows in which session it should execute the desired command. Once all processing is done, `jcs:close()` should be called to close the created Junoscript session.

Returning to the initial examples shown for `jcs:invoke()`, here is how they could be completed using `jcs:execute()` instead:

```
var $connection = jcs:open();
var $result = jcs:execute( $connection, "get-software-information" );
expr jcs:close( $connection );
```

And the second example:

```
var $connection = jcs:open();
var $config-rpc = {
   <get-configuration database="committed"> {
      <configuration> {
         <interfaces>;
      }
   }
}

var $result = jcs:execute( $connection, $config-rpc );
expr jcs:close( $connection );
```

With the introduction of the `jcs:execute()` function, it is possible for op scripts and event scripts to safely modify the configuration without any concern that they will interfere with configuration changes that were already in progress (either by other users, or by other applications), because the database could be locked, the configuration loaded and committed, and the database unlocked, all within a single Junoscript session. Here is an example of how that can be done. In the code below, the `$configuration` variable has already been populated with the desired change:

```
var $connection = jcs:open();
var $lock-results = jcs:execute( $connection, "lock-database" );
var $load-results = jcs:execute( $connection, $configuration );
var $commit-results = jcs:execute($connection,"commit-configuration");
var $unlock-results = jcs:execute( $connection, "unlock-database" );
expr jcs:close( $connection );
```

ALERT! Remember best practice is to check for errors at all the steps listed above and to halt the commit process if any occurred.

This introduces the first type of action that can be performed by `jcs:execute()` but not by `jcs:invoke()`, namely changing the configuration, and this is the most common action that a script performs that would necessitate using `jcs:execute()`. However, there are two other circumstances where `jcs:execute()` must, or in the last case, should, be used.

The first case is when using remote procedure calls.

The ability for a script to interact with remote Junos devices through the Junos API was first introduced in Junos 9.3 in combination with ssh-agent, and Junos 9.6 added the ability to login with a provided username and password. In both cases, it's accomplished by providing a hostname argument to the `jcs:open()` function so that the function creates the desired Junoscript session on the remote device, rather than locally.

```
/* using ssh-agent to pass thru username/password credentials */

var $remote-connection-1 = jcs:open( $remote-host );

/* using explicing username/password credentials encoded in the script */

var $remote-connection-2 = jcs:open( $remote-host, $username, $password );
```

The first example demonstrates how `jcs:open()` can be used to open a session with a remote device in combination with ssh-agent, and the second example shows how a session could be opened by providing the username and password as strings to `jcs:open()`.

Once the session is opened, it's handled just like a local session, and any API element that can be used locally can be used remotely as well. Here is the original example of `show version` again, this time executed on a remote host:

```
var $connection = jcs:open( $remote-host, $username, $password );
var $result = jcs:execute( $connection, "get-software-information" );
expr jcs:close( $connection );
```

Configuration changes and remote procedure calls are two scenarios that mandate the use of `jcs:execute()` instead of `jcs:invoke()`, but the third scenario is not as stringent as these two.

This third scenario occurs when a script has to execute a large quantity of API calls. In this case, `jcs:invoke()` could be used rather than `jcs:execute()`, however, there could be a large performance difference in favor of `jcs:execute()` because it does not have to open and close a Junoscript session in order to execute its API element. While the opening and closing action is sub-second, repeating it dozens of times can become noticeable.

As an example, on a MX960 a simple script was written that called `show version` 100 times through two separate recursive templates, one that used `jcs:invoke()`, and one that used `jcs:execute()`. The `jcs:execute()` method took around a second to perform all 100 executions of `show version`. On the other hand, `jcs:invoke()` required almost 20 seconds. This is just a rough example, and the actual processing time depends on the routing engine's CPU and the load of the Junos device, as well as the complexity of the API request, but the difference in processing speed between the two functions when many API requests are made is always present.

Does this mean that `jcs:execute()` should always be used in place of `jcs:invoke()`? Not necessarily. The main appeal to `jcs:invoke()` is its simplicity, as there is no need to open and close a session separately from the function call. This approach is sufficient in a large amount of scripts that only use, at most, a handful of API calls and as such would not experience a noticeable performance difference. Once the number of API calls starts to number in the dozens, however, then it is a good idea to consider switching the `jcs:invoke()` calls with `jcs:execute()` calls instead.

Using the <command> RPC

There are times when you might simply want to execute a set of commands without invoking the underlying Junos RPC.

For example, the following two scripts are equivalent

```
match / {
  <op-script-results> {
         var $rpc = <command> "show chassis hardware";

         copy-of jcs:invoke( $rpc );

  }
}
```

… and …

```
match / {
  <op-script-results> {
         var $rpc = <get-chassis-inventory>;

         copy-of jcs:invoke( $rpc );

  }
}
```

There are very few cases when using the `<command>` RPC is necessary, generally only when an XML RPC is not available. (If you identify a command that does not provide an XML RPC, it is most likely a bug, and you should report this to Juniper Networks.)

A common mistake when using the `<command>` RPC is trying to use the pipe commands. For example, the following is not technically supported:

```
<command> "show interface terse | match up"
```

You may find that using the pipe command may work in some cases, on some devices, for some releases of Junos, but again, it's not technically supported, so you shouldn't use the pipe constructs.

Highlights of the jcs: Function Library

The `jcs:` namespace at the top of all Junos scripts provides you access to a set of useful functions. For example, you have already seen the use of `jcs:invoke()` and `jcs:execute()`. This section highlights a few others that you should familiarize yourself with, as they can prove very handy.

MORE? A complete listing of the jcs function library can be found at http://www.juniper.net/techpubs/en_US/junos11.1/information-products/topic-collections/config-guide-automation/index.html

jcs:break-lines()

Let's say that you've got data in a CSV file and you'd like to have an op script process the contents. You would first use the `<file-get>` RPC to load the contents into a variable. But now you need to break the file contents into distinct lines (rows/records) for processing. You can use the `jcs:break-lines()` function to do this:

```
match / {
   <op-script-results> {

      /* Read the file */
      var $rpc = {
         <file-get> {
            <filename> "/var/tmp/mydata.csv";
            /* Leading and trailing newlines aren't added when raw encoding is used */
            <encoding> "raw";
         }
      }
      var $file = jcs:invoke( $rpc );

      /* Split into lines */
      var $lines = jcs:break-lines( $file/file-contents );

      for-each( $lines ) {
        var $line = .;

        <output> "line " _ position() _ ": " _ $line;
      }
   }
}
```

If the `file mydata.csv` contained:

```
Jeremy,Schulman,Male,40
Bob,Jones,Male,23
Sara,Parker,Female,50
```

The resulting output of the script would be:

```
line 1: Jeremy,Schulman,Male,40
line 2: Bob,Jones,Male,23
line 3: Sara,Parker,Female,50
```

jcs:first-of()

This function is handy for finding the first non-empty value of a given list or essentially creating a series of "if item A is empty, then check item B, then check item C, etc."

For example, let's say you are creating routing policies for BGP. BGP supports hierarchical policy definitions: global definitions, group definitions, and peer specific definitions. You could use `jcs:first-of()` to find the import policy for a given peer, as illustrated in this example:

```
match / {
   <op-script-results> {

      /* Retrieve desired peer */
      var $peer = jcs:get-input( "Enter BGP peer address: ");

      /* Get the configuration */
      var $rpc = <get-configuration database="committed" inherit="inherit">;
      var $configuration = jcs:invoke( $rpc );

      /* Retrieve the peer config */
      var $peer-config = $configuration/protocols/bgp//neighbor[name == $peer];

      /* If peer is not found then give an error */
      if( jcs:empty( $peer-config ) ) {
```

```
            <output> "Error: BGP peer " _ $peer _ " was not found";
        }
        /* Otherwise, display the import policies */
        else {
            /* Find the correct import policy configuration for the peer */
            var $import = jcs:first-of( $peer-config/import, $peer-config/../import,
                                $peer-config/../../import, "*None*" );

            /* Handle node-sets differently than strings */
            var $policy-string = {
                /* If a node-set, then add all the values */
                if( exsl:object-type( $import ) == "node-set" ) {
                    for-each( $import ) {
                        expr . _ " ";
                    }
                }
                /* If a string then just add */
                else {
                    expr $import;
                }
            }

            <output> "Import policies: " _ $policy-string;
        }
    }
}
```

The `jcs:first-of()` in the example is looking at the configuration hierarchy first at the peer specific neighbor level (`$peer-config/import`), then up one level at the group level (`$peer-config/../import`), and finally at the global level (`$peer-config/../../import`).

jcs:hostname()

This is a handy function for performing a DNS lookup based on an IP address. For example:

```
var $host = jcs:hostname( "mydevice.juniper.net" );
```

It also looks at the on-box `system static-host-mapping` configuration for name resolution in addition to querying DNS.

jcs:parse-ip()

This function is useful for decomposing the various IPv4 and IPv6 addresses into specific elements.

- Host IP address (or NULL in the case of an error)
- Protocol family (inet for IPv4 or inet6 for IPv6)
- Prefix length
- Network address
- Network mask in dotted decimal notation for IPv4 addresses (left blank for IPv6 addresses)

For example:

```
match / {
    <op-script-results> {
```

```
      var $address-set := {
         <address> "10.0.0.1/255.0.0.0";
         <address> "192.168.254.13/23";
         <address> "::ffff:0:0:10.100.1.1/96";
      }

      /* Go through each address in the set and display its information */
      for-each( $address-set/address ) {
           <output> "----------------------------------------";
         <output> "Address: " _ .;
         var $result = jcs:parse-ip( . );
         <output> jcs:printf( "%8s %s", "Host:", $result[1]);
         <output> jcs:printf( "%8s %s", "Family:", $result[2]);
         <output> jcs:printf( "%8s %s", "Prefix:", $result[3]);
         <output> jcs:printf( "%8s %s", "Network:", $result[4]);
         if( $result[2] == "inet" ) {
            <output> jcs:printf( "%8s %s", "Mask:", $result[5]);
         }
      }
   }
}
```

Would result in the following output:

```
Address: 10.0.0.1/255.0.0.0
  Host: 10.0.0.1
 Family: inet
 Prefix: 8
Network: 10.0.0.0
  Mask: 255.0.0.0
Address: 192.168.254.13/23
  Host: 192.168.254.13
 Family: inet
 Prefix: 23
Network: 192.168.254.0
  Mask: 255.255.254.0
Address: ::ffff:0:0:10.100.1.1/96
  Host: ::ffff:0:0:a64:101
 Family: inet6
 Prefix: 96
Network: 0:0:0:ffff::
```

jcs:printf()

Unlike its counterpart in other languages, `jcs:printf()` is used to create a formatted string, but it does not output that string. For example:

```
match / {
   <op-script-results> {

      /* Display the parameters */
      expr jcs:output( jcs:printf( "%15s %-10s", "Parameter", "Value" ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$user", $user ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$hostname", $hostname ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$product", $product ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$script", $script ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$localtime", $localtime ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$localtime-iso", $localtime-iso ) );


   }
}
```

Results in:

```
      Parameter Value
          $user jnpr
      $hostname srx210
       $product srx210h
        $script test.slax
    $localtime Tue Jun 21 08:57:19 2011
$localtime-iso 2011-06-21 08:57:19 UTC
```

TIP There are many formatting features supported by `jcs:printf()`. These features are listed in the appendix of this book. Familiarize yourself with them as much as you can, as there are many unique things you can do in Junos and you will find this function very handy.

jcs:regex()

Junos supports the standard "C" library regular expression functionality. An entire appendix that comprehensively goes through the capabilities is dedicated to this subject, but here is a simple example:

```
match / {
   <op-script-results> {

      /* parse the $localtime-iso parameter */
      var $regex =
      "([[:digit:]]*)-0?([[:digit:]]*)-0?([[:digit:]]*) 0?([0-9]*):0?([0-9]*):0?([0-9]*).*";

      var $result = jcs:regex($regex, $localtime-iso );

      /* Display the complete match */

      <output> "Time: " _ $result[1];

      /* Display all the captured subexpressions */
      <output> "Year: " _ $result[2];
      <output> "Month: " _ $result[3];
      <output> "Day: " _ $result[4];
      <output> "Hour: " _ $result[5];
      <output> "Minute: " _ $result[6];
      <output> "Second: " _ $result[7];
   }
}
```

Results in:

```
Time: 2011-04-23 02:37:44 UTC
Year: 2011
Month: 4
Day: 23
Hour: 2
Minute: 37
Second: 44
```

jcs:split()

This function is another handy text-processing utility. In the case of the CSV file, you could use this function to split the comma-separated fields. The pattern for splitting can be a regular expression as well. Here is an example:

```
match / {
   <op-script-results> {

      call show-substrings( $pattern = "[A-Z]+", $string = "10.4R1.9" );

      call show-substrings( $pattern = "\\.", $string = "10.100.2.1" );

      call show-substrings( $pattern = "[[.space.]]+", $string =
         "ge-0/0/0.0            up    up   inet    10.0.0.10/24");
   }
}

template show-substrings( $pattern, $string ) {
   expr jcs:output( "_\r \n", $string );
   expr jcs:output( str:padding( string-length( $string ), "-" ) );
   var $result = jcs:split( $pattern, $string );
   for-each( $result ) {
      expr jcs:output( . );
   }
}
```

Results in:

```
10.4R1.9
--------
10.4
1.9

10.100.2.1
----------
10
100
2
1

ge-0/0/0.0            up    up   inet    10.0.0.10/24
------------------------------------------------------
ge-0/0/0.0
up
up
inet
10.0.0.10/24
```

jcs:sysctl()

This function provides read-only access to the SYSCTL variables, useful for trouble-shooting system level issues. For example:

```
match / {
   <op-script-results> {

      <output> "kern.osrelease=" _ jcs:sysctl("kern.osrelease");
      <output> "kern.hostname=" _ jcs:sysctl("kern.hostname", "s");
      <output> "kern.osrevision=" _ jcs:sysctl("kern.osrevision", "i");
      <output> "hw.re.mastership=" _ jcs:sysctl("hw.re.mastership", "i");
      <output> "hw.product.model=" _ jcs:sysctl("hw.product.model", "s");
   }
}
```

Results in:

```
kern.osrelease=10.4R1.9
kern.hostname=srx210
```

```
kern.osrevision=199506
hw.re.mastership=1
hw.product.model=srx210h
```

One handy sysctl is the `kern.ticks` value. You could use this value to *profile* sections of your code by placing calls to get this value around the code; then subtracting the start time from the end time.

Displaying Output from Junos Commands

One of the common uses for op scripts is to create new display commands, which show combined information that was previously not available through a single source in a format agreeable to the script writer. Information from other commands can be retrieved individually and then applied to the format of the new command, but what if a large section of an existing command's output is desired, verbatim, in the new op script's output?

It is, of course, possible to manually mimic the Junos command output through the op script's code, but there is an easier alternative that can be used: displaying the CLI output of normal Junos commands as part of your op script output.

To understand how this is possible, first consider the interaction between the management daemon MGD, and CLI processes in Junos. CLI commands are run through MGD, which typically return its results to CLI as a XML document. CLI then takes this XML document and displays it according to the programmed parameters for the command.

In other words, this is what MGD provides to the CLI when asked to display *show version*:

```
<software-information>
   <host-name>srx210</host-name>
   <product-model>srx210h</product-model>
   <product-name>srx210h</product-name>
   <jsr/>
   <package-information>
      <name>junos</name>
      <comment>JUNOS Software Release [10.2R1.8]</comment>
   </package-information>
</software-information>
```

But, CLI takes the XML input, renders the output in its specific format, and then shows the following to the user:

```
user@junosjnpr@srx210> show version
Hostname: srx210
Model: srx210h
JUNOS Software Release [10.2R1.8]
```

To illustrate the significance of this, consider the following op script:

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

match / {
   <op-script-results> {
      <software-information> {
         <host-name> "srx210";
```

```
            <product-model> "MX960";
            <product-name> "MX960";
            <package-information> {
               <name> "junos";
               <comment> "JUNOS Software Release [10.5R1]";
            }
         }
      }
}
```

Note that, instead of using the `<output>` result tree element, as might be expected, the script instead uses the `<software-information>` element, which is usually provided from MGD to CLI, and has crafted specific values. This XML data is then provided to the CLI, which displays it in the following way:

```
user@junosjnpr@srx210> op show-fake-version
Hostname: srx210
Model: MX960
JUNOS Software Release [10.5R1]
```

While displaying a modified `show version` is not of much practical use, the underlying concept, that the CLI is capable of interpreting MGD-delivered XML data, can be used in scripts in a couple of different ways:

- Displaying an aggregate of multiple CLI commands through a single op script
- Displaying a subset of a CLI commands output

The first bulleted example is likely the most relevant. As demonstrated by the preceding op script, it is possible to instruct the CLI to display normal Junos command output by providing the appropriate XML elements in the result tree. This can be done manually as above, but it is far more likely that a script writer instead delivers the XML results received from `jcs:invoke()` by using the `copy-of` statement to copy the returned variables XML contents into the result tree.

The main consideration when performing this action is that the command output *must* be copied into the top-level `<op-script-results>` result tree element, otherwise it isn't processed as desired by the CLI. Here is an example of a script that displays multiple `show chassis` outputs via a single op script:

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

match / {

   <op-script-results> {
      var $chassis-hardware = jcs:invoke( "get-chassis-inventory" );
      var $chassis-environment = jcs:invoke( "get-environment-information" );
      var $chassis-routing-engine = jcs:invoke( "get-alarm-information" );
      var $chassis-thresholds = jcs:invoke("get-temperature-threshold-information");

      <output> "Chassis hardware:";
      copy-of $chassis-hardware;

      <output> "__________\nChassis environment:";
      copy-of $chassis-environment;

      <output> "__________\nChassis alarms:";
```

```
      copy-of $chassis-routing-engine;

      <output> "__________\nChassis temperature thresholds:";
      copy-of $chassis-thresholds;
   }
}
```

And here is the output displayed on the CLI:

```
user@junosjnpr@srx210> op chassis-outputs
Chassis hardware:
Hardware inventory:
Item           Version  Part number  Serial number    Description
Chassis                              AD2309AA0380     SRX210h
Routing Engine   REV 28   750-021779   AAAG9521        RE-SRX210-HIGHMEM
FPC 0                                                 FPC
  PIC 0                                               2x GE, 6x FE, 1x 3G
Power Supply 0
__________
Chassis environment:
Class Item                        Status     Measurement
Temp  Routing Engine               OK        43 degrees C / 109 degrees F
     Routing Engine CPU            Absent
Fans  SRX210 Chassis fan           OK        Spinning at high speed
Power Power Supply 0               OK
__________
Chassis alarms:
No alarms currently active
__________
Chassis temperature thresholds:
                      Fan speed       Yellow alarm        Red alarm
Item                 Normal   High    Normal   Bad fan    Normal   Bad fan
Chassis default           48     54       65        55       75        65
Routing Engine            55     60       75        65       85        70
```

The second bulleted example where this can be useful is when an op script should display a subset of the normal output of a Junos command. This requires more manual coding than the prior bulleted example, as the parent elements must be manually added into the script underneath the `<op-script-results>` element, and then specific elements must be copied into the result tree based on what subset is desired to be shown. The result is that all the boilerplate for the command, such as the header, is displayed, but only the desired portion of the output is included.

As an example, consider the `show interface filters` command. It can provide specific interface names if only certain interfaces are desired, but it does not currently have an option to only display certain address families. The following script allows the user to select an address family, and only interfaces that have filters for that address family are displayed.

The first step to achieve it is to consider the command's XML output. This is done by appending `| display xml` to the command, which shows the following:

```
user@junosjnpr@srx210> show interfaces filters | display xml

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.2R1/junos">
   <interface-filter-information xmlns="http://xml.juniper.net/junos/10.2R1/junos-interface"
junos:style="filter">
      <physical-interface>
         <name>ge-0/0/0</name>
         <admin-status>up</admin-status>
         <oper-status>up</oper-status>
         <logical-interface>
            <name>ge-0/0/0.0</name>
            <admin-status>up</admin-status>
```

```
            <oper-status>up</oper-status>
            <filter-information>
            </filter-information>
            <filter-information>
               <filter-family>inet</filter-family>
            </filter-information>
            <filter-information>
               <filter-family>mpls</filter-family>
               <filter-input>mpls-example</filter-input>
            </filter-information>
         </logical-interface>
      </physical-interface>
...
```

The above gives us all the information needed to write the script. First, it indicates what the top-level XML element is for this output: `<interface-filter-information>`. Next, it shows what element needs to be selectively included in the result tree: `<logical-interface>`, which is a child of `<physical-interface>`. And finally, it shows the XML hierarchy that must be navigated in order to determine if a particular interface has a firewall filter of the indicated family or not.

With the above information in mind, here is the completed op script:

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

var $arguments = {
   <argument> {
      <name> "family";
      <description> "Address family to display: inet, inet6, mpls, etc";
   }
}

param $family;

match / {
   <op-script-results> {

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

      /* Embed top-level element into result tree */
      <interface-filter-information junos:style="filter"> {

         /* Only include the logical-interface nodes that have an assigned
          * filter from the desired family.
          */
         for-each( $results/physical-interface/logical-interface ) {

            /* Filter family must match, and an input or output filter must be present */
            if(filter-information[filter-family == $family][filter-input | filter-output]){
               <physical-interface> {
                  <logical-interface> {
                     /* Copy all child elements except other family filters */
                     copy-of *[name() != "filter-information" || filter-family==$family];
                  }
               }
            }
         }
      }
   }
}
```

And here is the script's output:

```
user@junosjnpr@srx210> op filters-by-family family inet
Interface      Admin Link Proto Input Filter       Output Filter
fe-0/0/2.100    up    down inet  inet-example
fe-0/0/2.200    up    down inet                  inet-example
jnpr@srx210> op filters-by-family family mpls
Interface      Admin Link Proto Input Filter       Output Filter
ge-0/0/0.0      up    up   mpls  mpls-example
fe-0/0/2.100    up    down mpls  mpls-example       mpls-output
fe-0/0/2.300    up    down mpls                  fe-0/0/2.300-o
```

This can be much more involved than simply copying the entire command results to the result tree, but it does provide the advantage of customizing the actual Junos-standard output that users are accustomed to.

NOTE Because scripts that follow this approach are taking advantage of the MGD-to-CLI XML communication patterns, there is some risk of change between incremental Junos versions that would not be present were one to merely stick with the standard `<output>` method.

File and Storage Topics

It's coffee break time. Let all those fundamental topics sink in and go back and refresh your memory about any that really caught your eye. If you're not yet excited by the potential of Junos automation scripting with SLAX, you must be a File and Storage kind of scripter. If so, put that coffee down and let's get at it.

File I/O: Reading

The Junos XML API contains two separate elements that can be used to retrieve files from the file system: `<file-get>` and `<file-show>`. While both have equivalent functionality, their usage and results vary.

Encoding Types

The same three encoding options exist for both `<file-show>` and `<file-get>`, and are selected based on the `<encoding>` child element:

- ascii: Retrieves the file in ASCII format, but extra newlines are added before and, if the file does not already end with a newline, after the file contents, so this encoding format is not appropriate if the exact contents are required.
- base64: This encoding returns the file contents in base64 format. It should always be used when working with binary files.
- raw: Added in Junos 10.1. This encoding returns the file contents in ASCII format but does not add any extra newlines. If the script must work with the exact file contents, then this would be the appropriate encoding type.

<file-show>

The `<file-show>` API element is the equivalent of the `file show` CLI command. Its only required child element is `<filename>`, which specifies what file should be retrieved. The `<encoding>` child element is optional, and should only be included if the default ASCII encoding is not desired:

```
<file-show> {
   <filename> "filename";     /* Mandatory */
   <encoding> "base64 or raw"; /* Optional - defaults to ascii */
}
```

XML Results

When a file is successfully read, the `<file-show>` API element returns the file contents within a `<file-content>` element. The following three examples show the `<file-content>` results of a text file that consists of the string `no newlines`. These are shown in SLAX format to make the presence of newlines more obvious.

ASCII encoding:

```
<file-content> "
no newlines
";
```

Raw encoding:

```
<file-content> "no newlines";
```

Base64 encoding:

```
<file-content> "
bm8gbmV3bGluZXMK
";
```

If an error occurs (for example, the file doesn't exist or the user does not have adequate permissions) then an `<xnm:error>` element is returned along with a `<message>` child element explaining why the file could not be retrieved:

```
<xnm:error> {
   <message> "
could not resolve file: /var/home/lab/test20
";
}
```

Location Paths

Location paths with `<file-show>` are simple. The file content is the returned node in the node-set, so the contents can be retrieved by referring to the variable itself:

```
var $results = jcs:invoke( $file-show-rpc );
if( contains( $results, "this is inside of the file?" ) )
...
```

Wildcard Filename Selection

Just like the `file show` CLI command, the `<file-show>` API element can include the * character in the filename as a wildcard. This could be useful if the exact filename is unknown; however, only one filename is allowed to match the wildcard expression. If more than one filename matches, then a `<xnm:error>` element is returned indicating that the wildcard expression *resolves to multiple files*.

The following example shows the use of `<file-show>` to retrieve the ASCII contents of a file. If the file contains the word *contents* then it is displayed on the console, otherwise an error string is displayed:

```
match / {
   <op-script-results> {
      var $rpc = {
         <file-show> {
            <filename> "/var/home/lab/test1";
         }
      }
```

```
      var $results = jcs:invoke( $rpc );
      if( contains( $results, "contents" ) ) {
         copy-of $results;
      }
      else {
         expr jcs:output("'contents' is not included within the file.");
      }
   }
}
```

ALERT! Prior to Junos 11.1, the <file-show> element had no default directory, so its <file-name> element must include the complete path to the file. In Junos 11.1 and beyond, the default directory depends on the script type: (user's home directory for op scripts, /var/tmp for commit and event scripts, and / for op scripts executed by event policies).

<file-get>

The <file-get> API element was added in Junos 9.0 along with the <file-put> element to facilitate Junos script inline file transfers. Similar to <file-show>, it requires that the <filename> child element indicate what file should be read, but unlike <file-show>, the <encoding> element is mandatory when using <file-get>, because it has no default encoding format.

The <file-get> element has no supported CLI equivalent, however, its functionality can be performed from the CLI by using the hidden `file-mgd get` command . No ASCII results are displayed, making the CLI command rarely beneficial, but the XML results can be seen by appending `| display xml` to the command.

```
<file-get> {
   <filename> "filename";          /* Mandatory */
   <encoding> "ascii base64 or raw"; /* Mandatory */
}
```

XML Results

When a file is successfully read, the <file-get> API element returns a <file-get-results> element that includes the file contents within a <file-contents> element as well as a <success> element. The next three examples show the <file-get-results> results of a text file that consists of the string `no newlines`. These are shown in SLAX format to make the presence of `newlines` obvious.

ASCII encoding:

```
<file-get-results> {
   <file-contents> "
no newlines
";
   <success>;
}
```

NOTE Prior to Junos 10.0R3 and 10.1R1, files that ended with a newline would have an additional newline added.

Raw encoding:

```
<file-get-results> {
   <file-contents> "no newlines";
   <success>;
}
```

Base64 encoding:

```
<file-get-results> {
   <file-contents> “
bm8gbmV3bGluZXMK
“;
   <success>;
}
```

If an error occurs (for example, the file doesn't exist or the user does not have adequate permissions) then an <xnm:error> element is returned along with a <message> child element explaining why the file could not be retrieved:

```
<xnm:error> {
   <message> “
Failed to open file (/var/home/lab/test): No such file or directory
“;
}
```

Default Directory

The <file-get> element defaults to the /var/tmp directory for relative filenames. Use an absolute filename if the file is located in a different directory.

Location Paths

Location paths with <file-get> differ from those used with <file-show>. The returned node in the node-set is <file-get-results>, and to retrieve the actual file contents, the <file-contents> child element must be referenced:

```
var $results = jcs:invoke( $file-get-rpc );
if( contains( $results/file-contents, “this is inside the file?” ) )
...
```

In addition, the <success> element can be checked with the following location path:

```
if( $results/success ) {
  ...
}
...
```

Example Usage

This example shows the use of <file-get> to retrieve the ASCII contents of a file. If the file contains the word *contents* then it is displayed on the console, otherwise an error string is displayed:

```
match / {
   <op-script-results> {
      var $rpc = {
         <file-get> {
            <filename> “/var/home/jnpr/test”;
            <encoding> “ascii”;
         }
      }
      var $results = jcs:invoke( $rpc );
      if( contains( $results/file-contents, “contents” ) ) {
         expr jcs:output( $results/file-contents );
      }
      else {
         expr jcs:output(“‘contents’ is not included within the file.”);
      }
   }
}
```

Table 3.1 Comparison of <file-show> and <file-get>

	<file-show>	<file-get>
Minimum Junos Version	5.6	9.0
<encoding> element	Optional – defaults to ascii if not present. Valid options are “base64” or “raw”.	Mandatory. Valid options are “ascii”, “base64”, or “raw”.
Returned node	<file-content>	<file-get-results>
File content node	<file-content>	<file-contents>, which is a child of <file-get-results>
Default directory	None, prior to Junos 11.1, but as of that version, it depends on the script type.	/var/tmp
Filename wildcards	Supported as long as it resolves to only one file.	Unsupported

File I/O: Writing

The <file-put> API element, added in Junos 9.0, is used to write information to disk. When invoked, it creates the indicated output file and then writes the desired content, provided as either an ASCII or base64 string, to the new file. An error is returned if the output file already exists, unless the <delete-if-exist> child element is specified, in which case the file is deleted prior to being recreated and written. It is not currently possible to append it to an existing file.

NOTE One workaround that can be used, if appending is necessary, is to read the current file contents into a variable, append the new content to the current content, and then overwrite the file with the combined contents.

The <file-put> API element contains both mandatory and optional child elements:

```
<file-put> {
   <filename> “filename”;              /* Mandatory */
   <encoding> “base64 or ascii”;       /* Mandatory */
   <permission> “file permission string”; /* Optional */
   <delete-if-exist>;                 /* Optional */
   <file-contents> “”;                /* Mandatory */
}
```

The mandatory <filename> element indicates the name, and optionally the complete path, of the output file. If an absolute file path is not provided then the file is placed into the default directory, which is the home directory of the script’s executing user. Remember, however, that commit scripts are executed by root, not the user that performed the commit, and event scripts are executed by root by default as well, so in both of those cases the default directory would be the home directory of the root user: /root. The created output file is owned by the user that executed the script, and its group will be set to *staff* (or *wheel* if the script is being run by root.)

The <encoding> element is mandatory and must be set to either base64 or ASCII . It indicates the format of the <file-contents> element’s content. If it is set to ASCII, then the content is written directly to disk, but if base64 is selected, then the content is converted from base64 into ASCII before being written.

By default, files are created with permissions of 0600 (u=rw), but if a different permission level is desired, then the optional <permission> element can be included

with the permissions specified in either numeric or symbolic format:

Permission examples:

```
<permission> “u=r”;       /* read-only for user */
<permission> “666”;       /* read/write access for all users */
<permission> “u=rw,go=r”;  /* read/write for user, read-only for group/others */
<permission> “644”;       /* read/write for user, read-only for group/others */
```

As previously mentioned, if the output file already exists then the write operation will fail and a <xnm:error> message is returned. Alternatively, the <delete-if-exist> element can be included in the API call if an existing file should be overwritten. When used, Junos deletes the existing file and then recreates it with its new content, rather than returning an error.

The <file-contents> mandatory element contains the content that should be written to disk in either ASCII or base64 format.

ALERT! This element has a unique restriction that it must be the final element within the <file-put> API call.

This is necessary so that the script engine is already aware of the destination file and its parameters before it processes the <file-contents> element, so that it can start writing the data to disk immediately as it reads the file content, which could be essential when working with large files.

Writing to Non-Local Routing-Engines

Beginning in Junos 10.4, scripts can use the re#, node#, or fpc# URLs to write to non-local routing-engines in dual-routing-engine systems, chassis clusters, and EX VCs respectively. When using this URL notation, Junos first writes the file contents to a temporary file on the local routing-engine and then copies the temporary file to its intended destination. This behavior causes a difference in the <delete-if-exist> and <permission> child elements of <file-put>. The file copy operation overwrites existing files by default, so <delete-if-exist> has no effect, and the <permission> element only affects newly created files, but existing files retain their current permissions, and as of the time of this writing, write permission can only be granted to the file owner when using this URL notation. Another difference that occurs when using URL notation is that the file’s group ownership is assigned based on the destination directory’s group ownership, rather than the group of the script’s executing user.

NOTE These differences apply whenever using URL notation with the <file-put> API element, even if the destination is to the local routing-engine.

XML Results

A successful write operation results in a <file-put-results> element being returned, with child elements of <success> and <filename>, which contain the full path of the written file.

```
<file-put-results>
   <success/>
   <filename>
      /var/home/jnpr/example-file
   </filename>
</file-put-results>
```

If an error occurs, then a <xnm:error> element is returned. The contents of its <message> element vary, depending on the error seen, but here are some examples:

When the destination file already exists and <delete-if-exist> is not used:

```
<xnm:error>
   <message>
      Destination file exists
   </message>
</xnm:error>
```

Permission is denied to create a new file:

```
<xnm:error>
   <message>
      Write to destination file (/root/example-file) failed: Permission denied
   </message>
</xnm:error>
```

Permission is denied to delete existing file through <delete-if-exist>:

```
<xnm:error>
   <message>
      User jnpr failed to delete file /root/example-file: Permission denied
   </message>
</xnm:error>
```

Example Usage

The following op script writes "Example file contents" to the output file "example-file" in the executing user's home directory. Permissions are set to 644, the file is overwritten if necessary, and any resulting error messages are displayed on the screen:

```
match / {
   <op-script-results> {
      var $rpc = {
         <file-put> {
            <filename> "example-file";
            <permission> "644";
            <encoding> "ascii";
            <delete-if-exist>;
            <file-contents> "Example file contents";
         }
      }
      var $results = jcs:invoke( $rpc );
      if( $results//self::xnm:error ) {
         for-each( $results//self::xnm:error ) {
            expr jcs:output( message );
         }
      }
   }
}
```

SNMP Utility MIB

The Utility MIB, first introduced in Junos 8.4, gives network operators the ability to make any data value available via SNMP. It consists of five separate tables, one per data type: 32-bit counter, 64-bit counter, integer, unsigned-integer, and octet string. Each data value is identified by an ASCII instance name and has an accompanying timestamp also recorded in the MIB, in DateAndTime Hexadecimal format, that stores the last time the value was modified.

CLI Access

The Utility MIB can be modified through two *hidden* CLI commands:

```
user@junosjnpr@srx210> request snmp utility-mib set ?
Possible completions:
  instance
  object-type
  object-value

user@junosjnpr@srx210> request snmp utility-mib clear ?
Possible completions:
  instance
  object-type
```

In both cases, the instance refers to the ASCII name of the data instance (80 character maximum), and the object-type refers to one of the five data types:

- counter – 32-bit counter
- counter64 – 64-bit counter
- integer – signed integer
- unsigned-integer – unsigned integer
- string – string

The MIB values can be polled remotely by a normal SNMP management system, or they can be viewed on the CLI using the `show snmp mib walk` command. For example, to see the entire contents of the Utility MIB, use the following command:

```
user@junosjnpr@srx210> show snmp mib walk jnxUtil
```

The data instances are displayed in OID notation by default, but their ASCII names are shown instead if the `ascii` option is appended to the `show snmp mib walk` command:

```
user@junosjnpr@srx210> show snmp mib walk jnxUtil ascii
```

Here is an example of adding, viewing, and then removing an instance from the Utility MIB:

First, use `request snmp utility-mib set` to add the data instance:

```
user@junosjnpr@srx210> request snmp utility-mib set instance example object-type integer object-
value 1000
Utility mib result: successfully populated utility mib database
```

The data instance can now be viewed via the `show snmp mib walk` command (the ASCII option was added in Junos 9.6):

```
user@junosjnpr@srx210> show snmp mib walk jnxUtil
jnxUtilIntegerValue.101.120.97.109.112.108.101 = 1000
jnxUtilIntegerTime.101.120.97.109.112.108.101 = 07 da 07 10  0a 1f 35 00  2b 00 00

user@junosjnpr@srx210> show snmp mib walk jnxUtil ascii
jnxUtilIntegerValue."example" = 1000
jnxUtilIntegerTime."example" = 07 da 07 10  0a 1f 35 00  2b 00 00
```

Last, remove the instance with the `request snmp utility-mib clear` command:

```
user@junosjnpr@srx210> request snmp utility-mib clear instance example object-type integer
Utility mib result: successfully de-populated utility mib database
```

Object Value OIDs

The OIDs of the Utility MIB instances are created by appending the ASCII numeric value of the instance name to the base OID for the value of the particular object type. Due to this conversion from ASCII, the numeric values are effectively between 32 and 126.

Table 3.2 Object Value OIDs

Object Type	Value MIB Object	Value Base OID
counter	jnxUtilCounter32Value	1.3.6.1.4.1.2636.3.47.1.1.1.1.2
counter64	jnxUtilCounter64Value	1.3.6.1.4.1.2636.3.47.1.1.2.1.2
Integer	jnxUtilIntegerValue	1.3.6.1.4.1.2636.3.47.1.1.3.1.2
unsigned-integer	jnxUtilUintValue	1.3.6.1.4.1.2636.3.47.1.1.4.1.2
string	jnxUtilStringValue	1.3.6.1.4.1.2636.3.47.1.1.5.1.2

To illustrate the data shown in Table 3.2, with an object-type of integer, and a name of "example", the Utility MIB instance can be accessed at the following OID (e=101,x=120, a=97, etc.):

```
user@junosjnpr@srx210> show snmp mib get 1.3.6.1.4.1.2636.3.47.1.1.3.1.2.101.120.97.109.112.108.101
jnxUtilIntegerValue.101.120.97.109.112.108.101 = 1000

user@junosjnpr@srx210> show snmp mib get 1.3.6.1.4.1.2636.3.47.1.1.3.1.2.101.120.97.109.112.108.101
ascii
jnxUtilIntegerValue."example" = 1000
```

Timestamp OIDs

To retrieve the DateAndTime timestamp of the latest change to the data instance, follow the same process as with the object value OIDs, except with a slightly different base OID:

Table 3.3 Timestamp OIDs

Object Type	Timestamp MIB Object	Timestamp Base OID
counter	jnxUtilCounter32Time	1.3.6.1.4.1.2636.3.47.1.1.1.1.3
counter64	jnxUtilCounter64Time	1.3.6.1.4.1.2636.3.47.1.1.2.1.3
Integer	jnxUtilIntegerTime	1.3.6.1.4.1.2636.3.47.1.1.3.1.3
unsigned-integer	jnxUtilUintTime	1.3.6.1.4.1.2636.3.47.1.1.4.1.3
string	jnxUtilStringTime	1.3.6.1.4.1.2636.3.47.1.1.5.1.3

```
user@junosjnpr@srx210> show snmp mib get 1.3.6.1.4.1.2636.3.47.1.1.3.1.3.101.120.97.109.112.108.101
jnxUtilIntegerTime.101.120.97.109.112.108.101 = 07 da 07 10  0a 37 34 00  2b 00 00
```

```
user@junosjnpr@srx210> show snmp mib get 1.3.6.1.4.1.2636.3.47.1.1.3.1.3.101.120.97.109.112.108.1
01 ascii
jnxUtilIntegerTime."example" = 07 da 07 10  0a 37 34 00  2b 00 00
```

MORE? For more information, see the Utility MIB, and other enterprise-specific MIBs, within the Juniper technical documentation suite, available online at www.juniper.net/techpubs/.

Adding MIB Instances Within a Script

The Utility MIB can be manipulated from within any type of script: op, commit, or event. Adding instances to the Utility MIB is done using the `<request-snmp-utility-mib-set>` API element, as shown in this example:

```
var $rpc = {
   <request-snmp-utility-mib-set> {
      <instance> "example";
      <object-type> "string";
      <object-value> "example string";
   }
}
```

Once you've created the RPC definition, you can execute it using either the `jcs:execute()` or `jcs:invoke()` function. For example:

```
var $result = jcs:invoke( $rpc );
```

When the RPC is successful, the following result is returned:

```
<snmp-utility-mib-results>
   <snmp-utility-mib-result>
      successfully populated utility mib database
   </snmp-utility-mib-result>
</snmp-utility-mib-results>
```

When an error occurs, it is returned in a `<xnm:error>` element.

Clearing MIB Instances Within a Script

To clear Utility MIB instances, use the `<request-snmp-utility-mib-clear>` API element, as shown below:

```
var $rpc = {
   <request-snmp-utility-mib-clear> {
      <instance> "example";
      <object-type> "string";
   }
}

var $result = jcs:invoke( $rpc );
```

If no errors are encountered, the following is returned:

```
<snmp-utility-mib-results>
   <snmp-utility-mib-result>
      successfully de-populated utility mib database
   </snmp-utility-mib-result>
</snmp-utility-mib-results>
```

(Note that the above result is returned even if the instance does not currently exist,

so long as there are no actual errors in the request).

Walking the Utility MIB Within a Script

The Utility MIB can be walked by a script in the same way as any other MIB:

```
var $rpc = {
    <walk-snmp-object> {
        <snmp-object-name> “jnxUtil”;
    }
}

var $result = jcs:invoke( $rpc );
```

This provides results similar to the following:

```
<snmp-object-information>
    <snmp-object>
        <name>
            jnxUtilStringValue.101.120.97.109.112.108.101
        </name>
        <index>
            <index-name>
                jnxUtilStringName
            </index-name>
            <index-value>
                example
            </index-value>
        </index>
        <object-value-type>
            ASCII string
        </object-value-type>
        <object-value>
            example string
        </object-value>
        <oid>
            1.3.6.1.4.1.2636.3.47.1.1.5.1.2.101.120.97.109.112.108.101
        </oid>
    </snmp-object>
    <snmp-object>
        <name>
            jnxUtilStringTime.101.120.97.109.112.108.101
        </name>
        <index>
            <index-name>
                jnxUtilStringName
            </index-name>
            <index-value>
                example
            </index-value>
        </index>
        <object-value-type>
            Hex string
        </object-value-type>
        <object-value>
            07 da 07 10  0e 2e 2b 00  2b 00 00
        </object-value>
        <oid>
            1.3.6.1.4.1.2636.3.47.1.1.5.1.3.101.120.97.109.112.108.101
        </oid>
    </snmp-object>
</snmp-object-information>
```

Alternatively, as of Junos 9.6, the ASCII instance name can be retrieved by including the <ascii> child element:

```
var $rpc = {
   <walk-snmp-object> {
      <snmp-object-name> “jnxUtil”;
      <ascii>;
   }
}

var $result = jcs:invoke( $rpc );
```

Which provides results similar to the following:

```
<snmp-object-information>
   <snmp-object>
      <name>
         jnxUtilStringValue.”example”
      </name>
      <index>
         <index-name>
            jnxUtilStringName
         </index-name>
         <index-value>
            example
         </index-value>
      </index>
      <object-value-type>
         ASCII string
      </object-value-type>
      <object-value>
         example string
      </object-value>
      <oid>
         1.3.6.1.4.1.2636.3.47.1.1.5.1.2.101.120.97.109.112.108.101
      </oid>
   </snmp-object>
   <snmp-object>
      <name>
         jnxUtilStringTime.”example”
      </name>
      <index>
         <index-name>
            jnxUtilStringName
         </index-name>
         <index-value>
            example
         </index-value>
      </index>
      <object-value-type>
         Hex string
      </object-value-type>
      <object-value>
         07 da 07 10  0e 2e 2b 00  2b 00 00
      </object-value>
      <oid>
         1.3.6.1.4.1.2636.3.47.1.1.5.1.3.101.120.97.109.112.108.101
      </oid>
   </snmp-object>
</snmp-object-information>
```

Retrieving Utility MIB Instance Values Within a Script

To retrieve a specific instance, use the <get-snmp-object> API element, as the example below demonstrates:

```
var $rpc = {
   <get-snmp-object> {
      <snmp-object-name> "1.3.6.1.4.1.2636.3.47.1.1.5.1.2.101.120.97.109.112.108.101";
   }
}

var $result = jcs:invoke( $rpc );
```

Which returns results similar to the following:

```
<snmp-object-information>
   <snmp-object>
      <name>
         jnxUtilStringValue.101.120.97.109.112.108.101
      </name>
      <index>
         <index-name>
            jnxUtilStringName
         </index-name>
         <index-value>
            example
         </index-value>
      </index>
      <object-value-type>
         ASCII string
      </object-value-type>
      <object-value>
         example string
      </object-value>
      <oid>
         1.3.6.1.4.1.2636.3.47.1.1.5.1.2.101.120.97.109.112.108.101
      </oid>
   </snmp-object>
</snmp-object-information>
```

The object name does not have to be provided in OID format. A string instance with a name of "example" could be referred to in the API call through any of these three options:

```
<get-snmp-object> {
   <snmp-object-name> "1.3.6.1.4.1.2636.3.47.1.1.5.1.2.101.120.97.109.112.108.101";
}

<get-snmp-object> {
   <snmp-object-name> "jnxUtilStringValue.101.120.97.109.112.108.101";
}

<get-snmp-object> {
   <snmp-object-name> "jnxUtilStringValue.e.x.a.m.p.l.e";
}
```

In addition, just as with <walk-snmp-object>, Junos 9.6 introduced the <ascii> child element option, which can be included in the API call to request that the name be returned in ASCII rather than OID format:

```
var $rpc = {
   <get-snmp-object> {
      <snmp-object-name> "jnxUtilStringValue.e.x.a.m.p.l.e";
```

```
        <ascii>;
    }
}

var $result = jcs:invoke( $rpc );
```

Which returns the following:

```
<snmp-object-information>
    <snmp-object>
        <name>
            jnxUtilStringValue."example"
        </name>
        <index>
            <index-name>
                jnxUtilStringName
            </index-name>
            <index-value>
                example
            </index-value>
        </index>
        <object-value-type>
            ASCII string
        </object-value-type>
        <object-value>
            example string
        </object-value>
        <oid>
            1.3.6.1.4.1.2636.3.47.1.1.5.1.2.101.120.97.109.112.108.101
        </oid>
    </snmp-object>
</snmp-object-information>
```

Using the Utility MIB as a "Scratch Pad"

One of the challenges of the SLAX language is its inability to change variable values because they are immutable, meaning that once set they must always retain the same value. This can be a strange concept for many script writers to understand, and though it is possible to achieve most objectives through other means, such as recursion, working around this limitation does require a different way of thinking about the program flow than some programmers might be used to.

One possible workaround to this language quirk is to use the Utility MIB to store transient data. In effect, the MIB can be used as a scratch pad, where values are recorded and later retrieved as needed. Returning to the subject of changeable variables, this means that when a script writer wants to set a variable's value, they would set an instance in the Utility MIB, and when they want to use that value, they would retrieve the instance from the Utility MIB. In this way, the same functionality available to changeable variables can be used within SLAX scripts.

There are a few things to consider before using this approach, however:

- Data stored in the Utility MIB is not script specific, meaning that any other program, user, or even external management system can read the values being used by the script. As a result, it would be inappropriate to put sensitive data such as passwords into the MIB.
- The Utility MIB is writable by other programs, scripts, or users. This means that the scripts MIB instances could be altered by others, thereby interfering with the script operation.

- Due to the high rate of Junos API calls that this approach can require, it is often preferable to use the jcs:execute() function rather than jcs:invoke() when using the MIB as a scratch pad. For details on the reason for this, consult the previous section of this chapter that discusses the jcs:execute() and jcs:invoke() functions.
- To keep the Utility MIB from becoming cluttered, scripts should remove any transient instances from the MIB before terminating their operation.

As an example, consider the case where a script is trying to determine the longest interface description. This could be accomplished through a recursive template script design, but in this case, the Utility MIB is used to store the string that is currently considered the longest, while the script parses through the rest of the interface configuration, looking for any that might be longer.

This design would include a *for-each* loop to cycle through all the interface descriptions, and an idea of how this can be done is shown in this next example, where the `$configuration` variable has been previously filled through a call to `<get-configuration>`, and the `$connection` variable has already been initialized by the `jcs:open()` function:

```
for-each( $configuration/interfaces/interface/description ) {
   /* Retrieve current longest description */
   var $get-rpc = {
      <get-snmp-object> {
         <snmp-object-name> "jnxUtilStringValue.1.o.n.g.e.s.t";
      }
   }
   var $longest-string = jcs:execute( $connection, $get-rpc );

   /* Replace if new description is longer */
   if( string-length( . ) > string-length( $longest-string/snmp-object/object-value ) ) {
      var $put-rpc = {
         <request-snmp-utility-mib-set> {
            <instance> "longest";
            <object-type> "string";
            <object-value> .;
         }
      }
      var $result = jcs:execute( $connection, $put-rpc );
   }
}
```

The code example works by first retrieving the current value of the Utility MIB during each pass through the for-each loop, and storing it in a locally scoped variable. This variable is then compared to the description of the current node, and if the current description is longer than the value stored in the Utility MIB, as a string instance named "`longest`", then it is written into the Utility MIB as the new longest description value.

After the for-each loop has processed all the interface descriptions, the script can retrieve the longest one by checking the final instance value in the Utility MIB:

```
/* Display the longest description */
var $get-rpc = {
   <get-snmp-object> {
      <snmp-object-name> "jnxUtilStringValue.1.o.n.g.e.s.t";
   }
}

var $longest-string = jcs:execute( $connection, $get-rpc );
```

```
<output> "Longest interface description: " _ $longest-string/snmp-object/object-value;
```

Finally, before terminating, the script should remove its instance from the Utility MIB to prevent any unnecessary clutter from accumulating in the MIB:

```
/* Remove from Utility MIB */
var $clear-rpc = {
   <request-snmp-utility-mib-clear> {
      <instance> "longest";
      <object-type> "string";
   }
}

var $result = jcs:execute( $connection, $clear-rpc );
```

OSS Integration Topics

Generating Junos SNMP Traps ("Spoofing")

Junos scripts can generate any of the supported SNMP traps by using the <request-snmp-spoof-trap> API Element. This RPC contains two child elements: <trap> - which indicates the name of the trap that should be generated, and <variable-bindings> - which is a string containing the values of any *varbinds* that should be set. Any required varbinds that are not given values will have default ones assigned.

Here is an example of spoofing the "coldStart" trap:

```
var $rpc = {
   <request-snmp-spoof-trap> {
      <trap> "coldStart";
   }
}

var $results = jcs:invoke( $rpc );
```

Using this code results in the following trap being generated as seen in the SNMP traceoptions file:

```
Aug 13 04:00:14 snmpd[0]  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Aug 13 04:00:14 snmpd[0]  <<< V2 Trap
Aug 13 04:00:14 snmpd[0]  <<<  Source:     10.0.0.10
Aug 13 04:00:14 snmpd[0]  <<<  Destination: 10.0.0.60
Aug 13 04:00:14 snmpd[0]  <<<  Version:    SNMPv2
Aug 13 04:00:14 snmpd[0]  <<<  Community:  Target
Aug 13 04:00:14 snmpd[0]  <<<
Aug 13 04:00:14 snmpd[0]  <<<   OID  : sysUpTime.0
Aug 13 04:00:14 snmpd[0]  <<<   type : TimeTicks
Aug 13 04:00:14 snmpd[0]  <<<   value: (1313378)  3:38:53.78
Aug 13 04:00:14 snmpd[0]  <<<
Aug 13 04:00:14 snmpd[0]  <<<   OID  : snmpTrapOID.0
Aug 13 04:00:14 snmpd[0]  <<<   type : Object
Aug 13 04:00:14 snmpd[0]  <<<   value: coldStart
Aug 13 04:00:14 snmpd[0]  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

The <variable-bindings> element is a string containing a comma-separated list of varbinds, each of which consists of the object's name, its instance number within brackets, and the assigned value.

Here is an example of providing varbind values for a spoofed linkDown trap:

```
var $rpc = {
   <request-snmp-spoof-trap> {
      <trap> "linkDown";
      <variable-bindings> "ifIndex[501]=501, ifAdminStatus[501]=1, ifOperStatus[501]=2,
ifName[501]=ge-0/0/0";
   }
}

var $results = jcs:invoke( $rpc );
```

And this code generates the following trap:

```
Aug 13 04:31:33 snmpd[0]  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Aug 13 04:31:33 snmpd[0]  <<< V2 Trap
Aug 13 04:31:33 snmpd[0]  <<<  Source:     10.0.0.10
Aug 13 04:31:33 snmpd[0]  <<<  Destination: 10.0.0.60
Aug 13 04:31:33 snmpd[0]  <<<  Version:    SNMPv2
Aug 13 04:31:33 snmpd[0]  <<<  Community:   Target
Aug 13 04:31:33 snmpd[0]  <<<
Aug 13 04:31:33 snmpd[0]  <<<   OID  : sysUpTime.0
Aug 13 04:31:33 snmpd[0]  <<<   type : TimeTicks
Aug 13 04:31:33 snmpd[0]  <<<   value: (1501186)  4:10:11.86
Aug 13 04:31:33 snmpd[0]  <<<
Aug 13 04:31:33 snmpd[0]  <<<   OID  : snmpTrapOID.0
Aug 13 04:31:33 snmpd[0]  <<<   type : Object
Aug 13 04:31:33 snmpd[0]  <<<   value: linkDown
Aug 13 04:31:33 snmpd[0]  <<<
Aug 13 04:31:33 snmpd[0]  <<<   OID  : ifIndex.501
Aug 13 04:31:33 snmpd[0]  <<<   type : Number
Aug 13 04:31:33 snmpd[0]  <<<   value: 501
Aug 13 04:31:33 snmpd[0]  <<<
Aug 13 04:31:33 snmpd[0]  <<<   OID  : ifAdminStatus.501
Aug 13 04:31:33 snmpd[0]  <<<   type : Number
Aug 13 04:31:33 snmpd[0]  <<<   value: 1
Aug 13 04:31:33 snmpd[0]  <<<
Aug 13 04:31:33 snmpd[0]  <<<   OID  : ifOperStatus.501
Aug 13 04:31:33 snmpd[0]  <<<   type : Number
Aug 13 04:31:33 snmpd[0]  <<<   value: 2
Aug 13 04:31:33 snmpd[0]  <<<
Aug 13 04:31:33 snmpd[0]  <<<   OID  : ifName.501
Aug 13 04:31:33 snmpd[0]  <<<   type : OctetString
Aug 13 04:31:33 snmpd[0]  <<<   value: "ge-0/0/0"
Aug 13 04:31:33 snmpd[0]  <<<   HEX  : 67 65 2d 30  2f 30 2f 30
Aug 13 04:31:33 snmpd[0]  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

When specifying ASCII strings, the following characters must be escaped with a backslash (\ ):

- Opening and closing brackets: `[ ]`
- Comma: ,
- Equal sign: =
- Space
- Backslash: \

The escape character must be present within the actual XML string provided to the API element, not just in the SLAX script itself, which means that each backslash must be doubled. To illustrate, the string "`\[\]\,`" is invalid because the single \ escape

characters are handled by SLAX itself, and are removed when the script is translated into XSLT and processed. The correct string would be "\\[\\]\\," because the presence of the double-slash within the SLAX code results in a single backslash in the string when the API element is provided to Junos.

Generating Custom SNMP Traps

Besides spoofing standard traps, Junos scripts can also generate customized *jnxEventTraps*. This trap type was designed specifically for use in Junos scripts and is defined in the Juniper enterprise specific Event MIB. When generating a jnxEventTrap a script should, at a minimum, include a string value for the jnxEventTrapDescr varbind to describe the purpose of the trap. In addition, multiple attribute/value pairs can be included by using distinct instances of jnxEventAvAttribute and jnxEventAvValue.

The following code demonstrates how to generate a jnxEventTrap and include two attribute/value pairs along with the trap description:

```
var $rpc = {
   <request-snmp-spoof-trap> {
      <trap> "jnxEventTrap";
      <variable-bindings> "jnxEventTrapDescr[0]=Example\\ custom\\ trap, jnxEventAvAttribute[1]=a
ttribute1, jnxEventAvValue[1]=value1, jnxEventAvAttribute[2]=attribute2,
jnxEventAvAttribute[2]=value2";
   }
}

var $results = jcs:invoke( $rpc );
```

This results in the following trap:

```
Aug 13 04:51:25 snmpd[0]  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
Aug 13 04:51:25 snmpd[0]  <<< V2 Trap
Aug 13 04:51:25 snmpd[0]  <<<  Source:     10.0.0.10
Aug 13 04:51:25 snmpd[0]  <<<  Destination: 10.0.0.60
Aug 13 04:51:25 snmpd[0]  <<<  Version:    SNMPv2
Aug 13 04:51:25 snmpd[0]  <<<  Community:  Target
Aug 13 04:51:25 snmpd[0]  <<<
Aug 13 04:51:25 snmpd[0]  <<<   OID  : sysUpTime.0
Aug 13 04:51:25 snmpd[0]  <<<   type : TimeTicks
Aug 13 04:51:25 snmpd[0]  <<<   value: (1620428)  4:30:04.28
Aug 13 04:51:25 snmpd[0]  <<<
Aug 13 04:51:25 snmpd[0]  <<<   OID  : snmpTrapOID.0
Aug 13 04:51:25 snmpd[0]  <<<   type : Object
Aug 13 04:51:25 snmpd[0]  <<<   value: jnxEventTrap
Aug 13 04:51:25 snmpd[0]  <<<
Aug 13 04:51:25 snmpd[0]  <<<   OID  : jnxEventTrapDescr.0
Aug 13 04:51:25 snmpd[0]  <<<   type : OctetString
Aug 13 04:51:25 snmpd[0]  <<<   value: "Example custom trap"
Aug 13 04:51:25 snmpd[0]  <<<   HEX  : 45 78 61 6d  70 6c 65 20
Aug 13 04:51:25 snmpd[0]  <<<         63 75 73 74  6f 6d 20 74
Aug 13 04:51:25 snmpd[0]  <<<         72 61 70
Aug 13 04:51:25 snmpd[0]  <<<
Aug 13 04:51:25 snmpd[0]  <<<   OID  : jnxEventAvAttribute.1
Aug 13 04:51:25 snmpd[0]  <<<   type : OctetString
Aug 13 04:51:25 snmpd[0]  <<<   value: "attribute1"
Aug 13 04:51:25 snmpd[0]  <<<   HEX  : 61 74 74 72  69 62 75 74
Aug 13 04:51:25 snmpd[0]  <<<         65 31
Aug 13 04:51:25 snmpd[0]  <<<
Aug 13 04:51:25 snmpd[0]  <<<   OID  : jnxEventAvValue.1
```

```
Aug 13 04:51:25 snmpd[0]  <<<  type : OctetString
Aug 13 04:51:25 snmpd[0]  <<<  value: “value1”
Aug 13 04:51:25 snmpd[0]  <<<  HEX  : 76 61 6c 75  65 31
Aug 13 04:51:25 snmpd[0]  <<<
Aug 13 04:51:25 snmpd[0]  <<<  OID  : jnxEventAvAttribute.2
Aug 13 04:51:25 snmpd[0]  <<<  type : OctetString
Aug 13 04:51:25 snmpd[0]  <<<  value: “attribute2”
Aug 13 04:51:25 snmpd[0]  <<<  HEX  : 61 74 74 72  69 62 75 74
Aug 13 04:51:25 snmpd[0]  <<<        65 32
Aug 13 04:51:25 snmpd[0]  <<<
Aug 13 04:51:25 snmpd[0]  <<<  OID  : jnxEventAvAttribute.2
Aug 13 04:51:25 snmpd[0]  <<<  type : OctetString
Aug 13 04:51:25 snmpd[0]  <<<  value: “value2”
Aug 13 04:51:25 snmpd[0]  <<<  HEX  : 76 61 6c 75  65 32
Aug 13 04:51:25 snmpd[0]  <<<
Aug 13 04:51:25 snmpd[0]  <<<  OID  : snmpTrapEnterprise.0
Aug 13 04:51:25 snmpd[0]  <<<  type : Object
Aug 13 04:51:25 snmpd[0]  <<<  value: jnxProductNameSRX210
Aug 13 04:51:25 snmpd[0]  <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
```

Creating “Custom RPCs” with Op Scripts

Typically administrators execute op scripts from the Junos CLI. The underlying mechanism to execute an op script is via the <op-script> RPC. Therefore you can do the following:

- Invoke op scripts from other scripts
- Invoke op scripts via NETCONF

Let’s looks at a few effective ways for you to use Junos scripting to integrate into larger OSS applications.

Understanding the Op Script RPC

The op script RPC is defined in the Junos documentation as follows:

```
<op-script>
  <script> script-name </script>          /* required */
  <detail/>                              /* optional */
</op-script>
```

This is fine if your script does not have any parameters ... but what if it does? If you use the `| display xml rpc` command you can see the RPC as it is invoked. For example:

```
user@junos> op say-hello firstname Jeremy  lastname Schulman | display xml rpc

<op-script>
  <script> say-hello </script>
  <argument>
    <name> firstname </name>
    <value> Jeremy </value>
  </argument>
  <argument>
    <name> lastname </name>
    <value> Schulman </value>
  </argument>
</op-script>
```

This would lead you to believe that you could include op script parameters using the <argument> block ... but this is actually not the case. The correct way to include

parameters is to treat the parameter name as an element and the parameter value as the element-node value, as shown:

```
<op-script>
  <script> say-hello </script>
  <firstname> Jeremy </firstname>
  <lastname> Schulman </lastname>
</op-script>
```

NOTE It is important to understand that the parameter elements can only accept string values. Simple scalar types can be converted to strings (i.e., numbers, booleans), but not complex node-sets (i.e., XML data). There is a technique for passing XML node-sets to op scripts, which will be explained in a later section of this book.

Invoking Op Script from Another Script

Let's examine the first approach of executing an op script from another script. Here is a simple script called `introduction.slax` that invokes the `say-hello.slax` op-script:

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

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

param $firstname;
param $lastname;

var $arguments = {
  <argument> {
    <name> 'firstname';
    <description> 'The first name'interface to control';
  }
  <argument> {
    <name> 'lastname';
    <description> 'The last name'[enable | disable], default is disable';
  }
}

match / {
  <op-script-results> {

        var $rpc = <op-script> {
          <script> 'say-hello';
          <firstname> $firstname;
          <lastname> $lastname;
        }

        var $result = jcs:invoke( $rpc );

        copy-of $result;
  }
}
```

The output from invoking the introduction command:

```
user@junos> op introduction firstname Jeremy lastname Schulman
Hello Jeremy Schulman
```

Reviewing the XML output of the command:

```
user@junos> op introduction firstname Jeremy lastname Schulman | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
    <output>
        Hello Jeremy Schulman
    </output>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

Invoking Op Script from NETCONF

Up until now, this book has been focused on *on box* automation using Junos scripts. NETCONF is the protocol used to manage Junos boxes remotely, also known as *network orchestration*. You can develop OSS applications to manage Junos devices using the same Junos XML API as you've used for on-box scripting.

MORE? NETCONF is essentially XML over SSH. NETCONF is defined in RFC 4742. For more information on the NETCONF standard, please refer to: http://tools.ietf.org/html/rfc4742. Juniper Networks NETCONF support is documented here: http://www.juniper.net/techpubs/en_US/junos11.1/information-products/topic-collections/netconf-guide/index.html.

A typical OSS application would make calls directly to the Junos XML API. While this is a common approach, there are a few disadvantages.

The primary disadvantage is that the OSS application is now tightly coupled to the Junos XML API. If a change is required to the OSS operation, then the corresponding Junos specific APIs must be updated as well.

Let's say that your OSS needs to provision a new service. This service has a few parameters such as customer-name, vlan-id, and device-interface. The OSS system would need to create the Junos specific configuration XML definition, and then use the Junos XML RPCs: `lock`, `load-configuration`, `commit`, and `unlock`. A different approach could be to call a Junos op script with the service parameters, have the op script perform the Junos specific functions, and report back any status results.

Now let's say that you also want to create some troubleshooting features in your OSS. The OSS could make the necessary Junos RPCs to perform the specific commands, parse the results, and perform the necessary logic and so forth, but that sounds a lot like a good function for an op script. The op script could perform all the necessary Junos commands and then present the information in a consumable form for the OSS system.

Creating Op Scripts for NETCONF

Writing an op script for a NETCONF application is essentially no different than any other op script. The key difference is the creation of the result tree. Rather than creating a result tree with a top-level element of <op-script-results>, you need to create your own top-level element that is specific to your OSS. The reason is that your op script is no longer *communicating* with the Junos CLI user.

Let's modify the `say-hello.slax` script to be *NETCONF friendly*:

```
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”;

param $firstname;
param $lastname;

var $arguments = {
   <argument> {
      <name> ‘firstname’;
      <description> ‘Your first name’;
   }
   <argument> {
      <name> ‘lastname’;
      <description> ‘Your last name’;
   }
}

match / {
   <myoss-response> {
      <myoss-say-hello-response> {
            <hello-name> {
                <first-name> $firstname;
                <last-name> $lastname;
            }
      }
   }
}
```

Here you can see in the main template (`match /`) that the top-level element is called `<myoss-response>` rather than `<op-script-results>`.

When this op script is invoked on the CLI, nothing really happens:

```
user@junos> op say-hello firstname Jeremy lastname Schulman
```

However, if you display the XML output, you will see the result-tree:

```
user@junos> op say-hello firstname Jeremy lastname Schulman | display xml

<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/10.4R1/junos”>
    <myoss-say-hello-response>
        <hello-name>
            <first-name>
                Jeremy
            </first-name>
            <last-name>
                Schulman
            </last-name>
        </hello-name>
    </myoss-say-hello-response>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

You can see that the top-level element `<myoss-response>` is consumed by Junos, and the remaining result-tree elements are returned.

If the `say-hello` op script was executed by a NETCONF client (OSS application), the entire RPC reply would look like the following:

```
<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/10.4R1/junos”>
```

```
   <myoss-say-hello-response>
      <hello-name>
         <first-name>
            Jeremy
         </first-name>
         <last-name>
            Schulman
         </last-name>
      </hello-name>
   </myoss-say-hello-response>
</rpc-reply>
```

What is interesting here is that the resulting elements <first-name> and <last-name> values have leading and trailing space. XML output should normally look like this:

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/10.4R1/junos">
   <myoss-say-hello-response>
      <hello-name>
         <first-name>Jeremy</first-name>
         <last-name>Schulman</last-name>
      </hello-name>
   </myoss-say-hello-response>
</rpc-reply>
```

Due to this extra *padding*, the OSS application would need to *trim* the element values before using them. Here is an example of Java code for invoking the *say-hello op script* and displaying the last name.

```
XML rpc = new XMLBuilder().createNewRPC( "op-script" );
rpc.append( "script", "say-hello" );
rpc.append( "firstname", "Jeremy" );
rpc.append( "lastname", "Schulman" );

XML reply = jdev.execute( rpc );

XPath xpath = XPathFactory.newInstance().newXpath();
String lname = (String) xpath.evaluate( "//last-name", reply.getOwnerDocument(),
                                        XPathConstants.STRING );

System.out.println( "Last-name is [" + lname.trim() + "]" );
```

Previously, this chapter presented a technique for an op script to return application-specific XML data. What about passing application specific XML data as the input parameter? The technique for passing parameters as name/value pairs has been covered, but what happens when the OSS would prefer to pass XML?

There are two basic approaches to solve this problem. The first is that the OSS could simply store the XML parameters into a file, transfer the file to the Junos device, and then have the op-script load that file using the native XSLT `document()` function. While this approach is feasible, it does add a series of extra steps to the OSS application. The second, and preferred, approach would be to keep the XML data part of the op script RPC execution.

The Junos `<op-script>` RPC can only accept string arguments. In order to pass XML as a string argument, it must be converted into a string (and the approach illustrated here uses base64 encoding).

The steps are:

1. The OSS application must convert the XML data into an encoded base64 string.

2. The OSS application then calls the op script using the `<op-script>` RPC and passes the encoded base64 string in a name/value pair.

3. The op script must then decode the base64 string. This is done by writing the data to local file using the <file-put> RPC and set the <encoding> to "base64".

4. The op script must then load the temporary file into the script using the native XSLT document() function.

Let's take a look at a new op script that is used to provision VLAN services. The OSS application would like to pass the parameters to this script as XML, for example:

```
<vlans>
   <vlan name="Blue" vlan-id="100" inet-address="10.29.1.12/24">
      <interface name="ge-0/0/0" mode="access"/>
      <interface name="ge-0/0/1"/>                    /* default mode is 'access' */
      <interface name="xe-1/0/1" mode="trunk">
   </vlan>
</vlans>
```

This XML structure represents the OSS application requirements, and is decoupled from the Junos specific configuration hierarchy.

The op script could take a single parameter; let's say oss-xmlargs, as illustrated:

```
var $arguments = {
   <argument> {
      <name> 'oss-xmlargs;
      <description> 'OSS params in base64 encoding';
   }
}
```

The op script would then do something like the following to store the encoded parameters to a file and then load them using the XSLT document() function.

```
/* store the XML data to a temporary local file */

var $rpcFilePut = <file-put> {
   <filename> '/var/tmp/vlans.xml';
   <encoding> 'base64';
   <permission> "ug=rw,o=r";
   <delete-if-exist>;
   <file-contents> $oss-xmlargs;
}

var $rpcRC = jcs:invoke( $rpcFilePut );

if( $rpcRC//self::xnm:error ) {
   for-each( $rpcRC//self::xnm:error ) {
      <output> message;
   }
}
else {

   /* read the XML data back from the file */

   var $vlans = document( '/var/tmp/vlans.xml' );
   for-each( $vlans//vlan ) {
      <output> "vlan " _ ./@name _ " is VLAN-ID " _ ./@vlan-id;
   }

}
```

The approach of encoding the XML into base64 does remove the extra steps from the OSS application, and simply *transfers* them to the Junos platform. While this approach simplifies the OSS application, it does place an extra step on the op script,

namely to write the XML data to a file.

The example also uses a *hardcoded* filename, but this may not always work. If your OSS application is making concurrent calls to the same op script, then the XML parameters could "clobber" each other. If your OSS application has these types of concurrency "features," then you would need to take steps to ensure that the file names are unqiue. Here are two approaches.

The first approach is to have the OSS application generate a random filename (token), and pass that value as a second op script parameter. One suggestion is to make the token the MD5 hash value, so the op script gets both a unique filename and can then use that value to validate the contents of the file using the `<get-checksum-information>` RPC.

The second approach would have the op script generating a random filename. One suggestion is to use the `math:random()` function. This function returns a floating-point number between 0 and 1, which can then be used to generate a random filename. For example:

```
var $filename = "/var/tmp/vlanargs" _ math:random() * 10000;
```

MORE? For more information on the EXSLT math library, please refer to http://www.exslt.org/math/index.html

Installer Scripts

Imagine that you have a collection of scripts that you deploy on your devices, and you are looking for a simple and easy way to load and enable these without too much hassle. To simplify this process, you can create an op script that *packages* your collection of scripts. When this *installer* script is executed, it will write the packaged scripts to the Junos file system and then enable the scripts in the Junos configuration.

When you want to deploy your scripts, you can invoke the installer op script using the `op url` command. This command enables you to invoke an op script without first having it enabled in the Junos configuration. For example, if you have an installer script called *ipsec-tools.slax* stored on the local device in the `/tmp` directory you could invoke the script:

```
user@junos> op url /tmp/ipsec-tools.slax
```

Another approach is that if you had the installer script stored on an FTP server, you could use the FTP URL, as shown.

```
user@junos> op url ftp://myname:mypassword@myftpserver/ipsec-tools.slax
```

There are two methods to embed a script within another script. The first is to simply store it as script text within a single giant string. This has the advantage of allowing the embedded script to be directly edited by simply editing the installer script, but the disadvantage is that it places restrictions on the quotations used within the embedded script, as it is essential that the string content is not broken until the very end of the script.

An alternative method is to embed the script as a base64 string. This prevents the embedded script from being directly edited, but it ensures that no punctuation expressed within the embedded script negatively affects the installer script, and it provides a clear demarcation between the embedded script and the installer script

code. This later approach is the one typically used.

In both cases, the <file-put> API element is used to write the embedded scripts to disk, with the only difference being which encoding method is used. If the script is stored as a large ASCII string, then ASCII encoding should be used. Likewise, a base64 string should be written using base64 encoding.

As an example, consider this simple op script that displays *Hello World!* on the console. The goal is to embed this script into an installer script.

```
version 1.0;
ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
import "../import/junos.xsl";
match / {
    <op-script-results> {
        <output> "Hello World!";
    }
}
```

This could be embedded within an installer script as an ASCII string by replacing all " characters with ' characters, and store that into a local variable (`$embedded-script`) as demonstrated here:

```
var $embedded-script =
"version 1.0;
ns junos = 'http://xml.juniper.net/junos/*/junos';
ns xnm = 'http://xml.juniper.net/xnm/1.1/xnm';
ns jcs = 'http://xml.juniper.net/junos/commit-scripts/1.0';
import '../import/junos.xsl';
match / {
    <op-script-results> {
        <output> 'Hello World!';
    }
}";
```

Next, the <file-put> API element could write the file to disk using ASCII encoding:

```
var $file-put-rpc = {
    <file-put> {
        <filename> "/var/db/scripts/op/hello-world.slax";
        <encoding> "ascii";
        <permission> "644";
        <file-contents> $embedded-script;
    }
}
```

The preferred approach, storing the script as a base64 string, is shown:

```
var $embedded-script="dmVyc2lvbiAxLjA7DQoNCm5zIGp1bm9zIDOgImh0dHA6Ly94bWwuanVuaXBlci5uZXQvanVub-
3Mv9qdW5vcyI7DQpucyB4bm0gPSAiaHR0cDovL3htbC5qdW5pcGVyLm5ldC94bm0vMS4xL3hubSI7DQpucyBqY3MgPSAiaH-
R0cDovL3htbC5qdW5pcGVyLm5ldC9qdW5vcy9jb21taXQtc2NyaXB0cy8xLjAiOw0KDQppbXBvcnQgIi4uL2ltcG9ydC9qd-
W5vcy54c2wiOw0KDQptYXRjaCAvIHsNCiAgICAvIHs8b3Atc2NyaXB0LXJlc3VsdHM+IHs8b3V0cHV0PiJIZWxsbyBXb3Js-
bGxvIFdvcmxkISI7DQogICAgfQ0KfQ0KDQo=";
```

And then <file-put> could be used with base64 encoding to write it to disk:

```
var $file-put-rpc = {
    <file-put> {
        <filename> "/var/db/scripts/op/hello-world.slax";
        <encoding> "base64";
        <permission> "644";
        <file-contents> $embedded-script;
    }
}
```

It is also recommended to verify the checksum once the embedded scripts are written to disk. This is accomplished by first including the checksum value into the installer as a separate variable, and then using the `<get-checksum-information>` RPC on the stored file and comparing the two values. The default checksum calculation is based on MD5, but you can also use SHA-256 and SHA1.

Once the embedded script has been stored to file, and the checksum verified, you could then make the appropriate configuration change to enable the script in the `[system scripts]` hierarchy.

Creating Custom "Databases" in Junos

Junos automation can be used to create a wide range of custom tools, to handle events, and to ensure that configurations meet your specific business practices. Junos automation techniques can also be combined to effectively create complete *application solutions*. When one thinks of an application there are generally three basic building blocks: (1) the user interface, (2) the database, and (3) the application logic that ties the first two together. Let's examine how Junos automation can be used to address each of these building blocks.

The user interface can be implemented though op scripts. Op scripts are often used by the network operator on the Junos CLI, but as shown in the previous sections of this chapter, op scripts can also be executed from an OSS/NETCONF. The former technique is great for *box jockies* and the later technique is great for OSS systems that want to *front end* the user experience with a GUI. In both cases, the op scripts can be used for provisioning activities as well as for service monitoring and troubleshooting.

The application logic can be implemented through a combination of op scripts and commit scripts. Commit scripts can examine the candidate configuration changes and perform additional functions. An op script, for example, could make a basic configuration change, and the commit script can detect that change and apply further changes. What is needed to effectively tie the two together is a database that enables the solution to tie together application specific information so that the op scripts and commit scripts can effectively work together. Let's take a look at a simple example.

Let's say that you are trying to build a solution that provisions VLAN based services. Each customer *record* would have a name, a VLAN-ID, an interface, and SLA characteristics. You could say that you have a customer-service database, with records for each customer. You can represent this in a Junos configuration file using *configuration groups* representing the database and apply-macro blocks within the group, each block representing a database record.

For example, say that you have a database called *vlan-services* and two customer records. They could be stored in a Junos configuration file:

```
user@junosjeremy@srx210# show groups
vlan-services {
   apply-macro customer-BOB {
      name BobCorp;
      description "The Bob Corporation, NY";
      vlan-id 112;
      interface ge-0/0/11;
      id 10719;
   }
   apply-macro customer-NAMCO {
      name Namco;
      id 772645;
```

```
        description "Namco Plastics, CA";
        vlan-id 718;
        interface ge-1/0/19;
    }
}
```

The information stored in the apply-macro blocks is entirely up to you as the application designer. This information can be used by Junos, though perhaps some of the information is not used by Junos, but is a relative marker for the OSS application (e.g., the id field).

Conceptually, you would then create a commit script that would look for apply-macro blocks (records) in the vlan-services group (database), and then perform the actual Junos specific configuration steps necessary to implement the given service.

This simple example illustrates a few key points. The database is a representation of the application and is decoupled from Junos specifics. Effectively this creates a database schema for the application. When changes are needed by the application, such as adding new fields, for example, the OSS application does not become highly coupled with the implementation specifics of Junos. Let's say that the OSS application needs to add a bandwidth field. This could easily be represented in the apply-macro block, and the application would not need to know how, exactly, the Junos device is provisioned to implement that feature.

Now, when you need to create a new customer record, you could do it in one of two ways. The first way could be to edit the Junos configuration file directly and create a new apply-macro block, a method that is prone to user errors, and again creates a coupling between the application goals and the Junos specifics – or having to create something called an apply-macro block.

A better approach would be to create an op script that either accepts service parameters on the command line and/or prompts the user for the service information. The op script is then responsible for creating the Junos specific configuration – for example, the apply-macro block. When the op script commits the changes, the underlying commit script would then use the data in the apply-macro block to expand the actual Junos configuration to implement the service.

By utilizing this approach, you can create a seamless service experience across many Juniper products without complicating your OSS. For example, let's say that you are trying to create a service that touches EXs, MXs, and SRX devices. The service on the EX may be a simple VLAN, but the MX might be a VPLS service. In both cases, the service definitions are the same, but the underlying configurations on each device would be different.

Once you have this type of database infrastructrure in place, it is relatively easy to construct monitoring and troubleshooting scripts. The input to these scripts could be the customer name and/or ID. Since the relevent information is present in the database record, the op scripts can use this information to perform their tasks. For example, by just providing the user name, you could check the VLAN status and the interface status.

Event Script Topics

Changing the configuration based on the success or failure of a RPM (Real-Time Performance Monitoring) test is a common automation goal as it allows the reachability of a remote device to be considered when determining the appropriate configuration for the device.

Reachability Based Configuration Changes

While RPM tests are capable of generating a number of events that an event script could be triggered by, when acting on reachability information the two events that are typically monitored are PING_TEST_FAILED and PING_TEST_COMPLETED, which indicate the failure or success of a specific RPM test. Both events have identical attributes: test-owner, the RPM owner of the test; and, test-name, the RPM test name. Both of these attributes are essential in differentiating between the various RPM tests that could be running on the Junos device.

Here is a basic attempt at creating RPM reachability triggered event policies. Assume that a RPM test is defined to a remote destination with an owner name of *server-check* and a test name of *alpha*. The following configuration would cause a separate event script to be executed any time the RPM test failed or succeeded:

```
policy test-failed {
    events PING_TEST_FAILED;
    attributes-match {
        ping_test_failed.test-owner matches server-check;
        ping_test_failed.test-name matches alpha;
    }
    then {
        event-script failed-config.slax;
    }
}

policy test-completed {
    events PING_TEST_COMPLETED;
    attributes-match {
        ping_test_completed.test-owner matches server-check;
        ping_test_completed.test-name matches alpha;
    }
    then {
        event-script successful-config.slax;
    }
}
```

In this example, the failed-config.slax script would make the necessary configuration change following a RPM test failure, and the successful-config.slax script would make the change needed following a successful RPM test. (These scripts should consult the current configuration before changing it, in order to ensure that the change isn't already in place and thereby preventing an unnecessary commit from occurring.)

But the above approach is still not optimal for a couple of different reasons. First, it requires the use of two separate scripts. A better approach is to use a single script that determines the necessary configuration change based on the trigger event that caused it to be executed. The trigger event can be learned by processing the <event-script-input> information provided to the event script when it is invoked. The trigger event name can be found through this location path: event-script-input/trigger-event/id. The script could check that value and behave differently if the trigger event is PING_TEST_FAILED than if it is PING_TEST_COMPLETED.

MORE? For more details on <event-script-input>, see *Day One: Applying Junos Event Automation* in the *Day One* library at www.juniper.net/dayone.

The second, and larger, problem with the above configuration is that it results in an event script being called after every PING_TEST_COMPLETED or PING_TEST_FAILED event.

This means that with a 60 second test interval, the event script is triggered every minute even if the RPM test hasn't lost a single probe in months, which results in a lot of unnecessary processing. The reason for this unneeded processing is that the event policy is reacting to the occurrence of a single event, but a PING_TEST_FAILED event by itself does not indicate that the configuration must be changed; rather, a PING_TEST_FAILED event that occurred following a PING_TEST_COMPLETED event does. So the solution is to change the event policies to only execute their script if the opposite event has occurred within a certain interval of time. In our example the test interval is 60 seconds, so to provide some buffer, the opposite event is required within the past 90 seconds:

```
policy test-failed {
   events PING_TEST_FAILED;
   within 90 events PING_TEST_COMPLETED;
   attributes-match {
      ping_test_failed.test-owner matches server-check;
      ping_test_failed.test-name matches alpha;
      ping_test_completed.test-owner matches server-check;
      ping_test_completed.test-name matches alpha;
   }
   then {
      event-script check-configuration.slax;
   }
}

policy test-completed {
   events PING_TEST_COMPLETED;
   within 90 events PING_TEST_FAILED;
   attributes-match {
      ping_test_completed.test-owner matches server-check;
      ping_test_completed.test-name matches alpha;
      ping_test_failed.test-owner matches server-check;
      ping_test_failed.test-name matches alpha;
   }
   then {
      event-script check-configuration.slax;
   }
}
```

In the above configuration, the check-configuration.slax script is only invoked if the opposite event has occurred within the last 90 seconds. (Note, this means that one extra script execution could occur at times, depending on the timing of the RPM test). The same script is used for both success and failure, so it must verify which event triggered it by consulting the <event-script-input> element in its source tree.

The script example is much improved, but faulty in practice because it can leave the Junos device with an incorrect configuration. To understand how this could happen, imagine that a remote destination, which was previously reachable, goes offline when the Junos device is rebooting. When the device comes back online and begins sending RPM probes to the offline destination, these will not be returned, resulting in a failure of the RPM test. But the PING_TEST_FAILED event will not trigger the event policy, because of the lack of any PING_TEST_COMPLETED events within the specified window. The result is that the configuration remains as if the device is online, while the device itself could be offline perpetually.

The solution to this scenario is to also watch for situations where no event history would be available, so the script should be run based solely on the occurrence of the event itself. The two events that could cause a lack of event history are system boot-up and restart of the event-processing daemon. The example below demon-

strates how to add these corner cases into the policy to ensure that the policy is triggered when the configuration might have to change:

```
policy test-failed {
   events PING_TEST_FAILED;
   within 240 events [ PING_TEST_COMPLETED KERNEL SYSTEM ];
   attributes-match {
      ping_test_failed.test-owner matches server-check;
      ping_test_failed.test-name matches alpha;
      ping_test_completed.test-owner matches server-check;
      ping_test_completed.test-name matches alpha;
      SYSTEM.message matches “Starting of initial processes complete”;
      KERNEL.message matches “event-processing \(PID.*\)started”;
   }
   then {
      event-script check-configuration.slax;
   }
}


policy test-completed {
   events PING_TEST_COMPLETED;
   within 240 events [ PING_TEST_FAILED KERNEL SYSTEM ];
   attributes-match {
      ping_test_completed.test-owner matches server-check;
      ping_test_completed.test-name matches alpha;
      ping_test_failed.test-owner matches server-check;
      ping_test_failed.test-name matches alpha;
      SYSTEM.message matches “Starting of initial processes complete”;
      KERNEL.message matches “event-processing \(PID .*\) started”;
   }
   then {
      event-script check-configuration.slax;
   }
}
```

Neither the reboot nor the event-processing restart are normal events. Instead they use the generic `SYSTEM` and `KERNEL` non-standard event IDs with their message being matched to identify the specific event of interest. The within timer was increased from 90 to 240 to allow more time for the initial RPM results following system boot-up. The result is that for four minutes after any of the three events – the opposite RPM result, system bootup, or event-processing restart – the script will execute, but once that period has passed, a consistent RPM response will not result in any unnecessary script processing.

MORE? For an example of an event script that triggers based on RPM reachability, consult the Next Hop Watcher script in the Appendix of *Day One: Applying Junos Event Automation* at www.juniper.net/dayone.

Time-based Configuration Changes

Sometimes it is necessary to automatically alter the configuration at certain times of day. This might be necessary to alternate between various network uplinks or to apply different firewall filter rules at different hours of the day. To do so, a time-of-day generate-event must be configured, and then that event must be applied to an event policy that executes an event script when triggered.

Here is an example of how this time-of-day generate-event can be applied. The goal of the example is to have two separate user accounts – day-shift and night-shift –

which are only activated at the appropriate times. Day-shift will be active from 8:00 am until 8:00 pm, at which time night-shift becomes active until 8:00 in the morning.

First, the two time-of-day generate events must be created:

```
generate-event {
   morning time-of-day “08:00:00 +0000”;
   evening time-of-day “16:00:00 +0000”;
}
```

Next, the event policy must reference these two events as its triggers and include the event script as the policy action:

```
policy user-accounts {
   events [ morning evening ];
   then {
      event-script check-user-accounts.slax;
   }
}
```

The event script loads the configuration, compares it against the current time, and if a change to either of the user accounts is warranted then it activates/deactivates the appropriate accounts and commits:

```
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”;
ns date = “http://exslt.org/dates-and-times”;
import “../import/junos.xsl”;
match / {
   /* Get current hour of day */
   var $hour = date:hour-in-day();

   var $connection = jcs:open();

   /* Retrieve the configuration */
   var $login-config = jcs:execute( $connection, “get-configuration” )/system/login;

   /* Record if active or not */
   var $day-user-active = not( jcs:empty( $login-config/user[name == “day-shift”][ jcs:empty( @
inactive ) ] ) );
   var $night-user-active = not( jcs:empty( $login-config/user[name == “night-shift”][ jcs:empty(
@inactive ) ] ) );

   /* Build the necessary configuration change */
   var $configuration := {
      <configuration> {
         /* Daytime? */
         if( $hour >= 8 && $hour < 16 ) {
            if( not( $day-user-active ) ) {
               <system> {
                  <login> {
                     <user active=”active”> {
                        <name> “day-shift”;
                     }
                  }
               }
            }
            if( $night-user-active ) {
               <system> {
                  <login> {
                     <user inactive=”inactive”> {
```

```
                        <name> “night-shift”;
                    }
                }
            }
        }
        /* Nighttime */
        else {
            if( $day-user-active ) {
                <system> {
                    <login> {
                        <user inactive=”inactive”> {
                            <name> “day-shift”;
                        }
                    }
                }
            }
            if( not( $night-user-active ) ) {
                <system> {
                    <login> {
                        <user active=”active”> {
                            <name> “night-shift”;
                        }
                    }
                }
            }
        }
    }

    /* Is there a change present? - then load it */
    if( $configuration/configuration/* ) {
        var $results := { call jcs:load-configuration( $connection, $configuration ); }
        /* Report any errors */
        if( $results//self::xnm:error ) {
            for-each( $results//self::xnm:error ) {
                expr jcs:syslog( “daemon.error”, “Event script error: “, message );
            }
        }
    }

    expr jcs:close( $connection );
}
```

The above event script and event policy is a decent example of how to change the configuration based on the time of day; it contains two flaws, however. First, it makes no attempt to retry the commit if an error prevented the change, and second, it requires that the current configuration be correct when the event script is applied and does not account for reboots. Solutions for both of these two deficiencies will be explored in the following sections.

Retrying a Configuration Change Within an Event Script

Many event scripts that change the configuration are not programmed to handle configuration errors, which is unfortunate because it is very likely that an event script that automatically changes the configuration will run into occasional interference, typically caused by the presence of some other user or script already having locked the configuration and thereby preventing the event script from doing so.

At a minimum, an event script that is unable to make its programmed changes should log an error message to the syslog to alert the operators that manual intervention is

necessary due to the failure. However, rather than requiring a user to login and make the needed change, the script can be written to handle temporary configuration lock failures by waiting a certain amount of time and then retrying its commit. In other words, the script can sleep for some number of minutes and then retry to gain a configuration lock. If it fails again then it could retry once again (up to a certain number of times), or just log an error and exit, requiring manual intervention to resolve the problem.

When retrying a configuration change, it is important to have the event script recheck the configuration following its pause, because the needed change might have already been made while the script was sleeping, or the underlying conditions that necessitated the change might have altered, meaning that the changes should no longer be performed.

The `check-user-accounts.slax` script has been modified to allow it to retry three times before finally giving up:

```
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";
ns date = "http://exslt.org/dates-and-times";
import "../import/junos.xsl";

/* Make connection a global variable */
var $connection = jcs:open();

match / {
   /* Call the configuration template */
   call make-change-if-needed();

   expr jcs:close( $connection );
}

/* $try defaults to 1, but is allowed to be as high as 3 */

template make-change-if-needed( $try = 1 ) {

   /* Get current hour of day */
   var $hour = date:hour-in-day();

   /* Retrieve the configuration */
   var $login-config = jcs:execute( $connection, "get-configuration" )/system/login;

   /* Record if active or not */
   var $day-user-active = not( jcs:empty( $login-config/user[name == "day-shift"][ jcs:empty( @
inactive ) ] ) );
   var $night-user-active = not( jcs:empty( $login-config/user[name == "night-shift"][ jcs:empty(
@inactive ) ] ) );

   /* Build the necessary configuration change */
   var $configuration := {
      <configuration> {
         /* Daytime? */
         if( $hour >= 8 && $hour < 16 ) {
            if( not( $day-user-active ) ) {
               <system> {
                  <login> {
                     <user active="active"> {
                        <name> "day-shift";
                     }
```

```
                }
            }
        }
        if( $night-user-active ) {
            <system> {
                <login> {
                    <user inactive="inactive"> {
                        <name> "night-shift";
                    }
                }
            }
        }
        /* Nighttime */
        else {
            if( $day-user-active ) {
                <system> {
                    <login> {
                        <user inactive="inactive"> {
                            <name> "day-shift";
                        }
                    }
                }
            }
            if( not( $night-user-active ) ) {
                <system> {
                    <login> {
                        <user active="active"> {
                            <name> "night-shift";
                        }
                    }
                }
            }
        }
    }
}

/* Is there a change present? - then try to load it */
if( $configuration/configuration/* ) {
    var $results := { call jcs:load-configuration( $connection, $configuration ); }

    /* Report any errors */
    if( $results//self::xnm:error ) {

        for-each( $results//self::xnm:error ) {
            expr jcs:syslog( "daemon.error", "Event script error: ", message,
                                            ": Try ", $try, " of 3" );
        }
        if( $try < 3 ) {
            /* Sleep for 2 minutes */
            expr jcs:sleep( 120 );
            call make-change-if-needed( $try = $try + 1 );
        }
        else {
            expr jcs:syslog( "daemon.error", "Event script is exiting." );
        }
    }
}
}
```

A new recursive template was added to the above script: `make-change-if-needed`. After determining that a configuration change is required, an attempt is made, and if an error occurs then, if the maximum number of retries haven't already been attempt-

ed, the make-change-if-needed template calls itself, incrementing the value of the `$try` parameter as it does so.

Pairing Commit Scripts with Event Scripts

Another common problem with event scripts that change the configuration is that they do not consider the initial state of the configuration when the script is first configured or when the box boots. Take a look at the current `check-user-accounts.slax` event script. Imagine that it is first enabled at noon. Is there any guarantee that the day-shift account is active and the night-shift account is inactive? No, instead the configuration could remain in an incorrect state until the next trigger event occurs at 8:00pm. A similar problem happens at boot-time, because the Junos device is only configured to check the accounts at the two set times of day, so it might leave its configuration in a faulty state for hours. And finally, the proper use of the script relies on an assumption that the day-shift and night-shift accounts are already in the configuration, already have their class assigned, and already have their authentication string configured. But what if they are not present? In that case, the configuration change will not be successful because the accounts lack a login class, or the user will not be able to login because they lack a password.

The ideal solution that covers all of the above scenarios is to include a *commit script pair* for the event script. The role of the event script is to react to certain events and change the configuration accordingly. For example, in the case studied in the past two sections, the role of the event script is to alter the day-shift and night-shift accounts as necessary at two distinct times of the day. Now, pair that with a commit script, which has the role of ensuring that the user accounts are configured correctly at the time of commit. The result is that when first enabling the event script, or when the box reboots, the commit script correctly activates/deactivates the accounts, or adds any missing user account configuration. This catches the above scenarios, ensuring that the configuration will be set as desired based on the time of day.

The event script and commit script pair must be separate files because they are stored in different directories on the Junos device, but in many cases it is possible to share the configuration change code between the two scripts by having the change built within a designated template, and then importing one of the scripts into the other. That way both scripts can rely on the exact same template for their configuration changes, and just add the distinct top-level element required for their type of configuration change (<change> for commit script, <configuration> for event script).

In this script, however, the configuration change made by the commit script differs from that of the event script because it also checks that there is a class and password configured for the user accounts.

The code required for the commit script:

```
version 1.0;
ns junos = “http://xml.juniper.net/junos/*/junos”;
ns xnm = “http://xml.juniper.net/xnm/1.1/xnm”;
ns jcs = “http://xml.juniper.net/junos/commit-scripts/1.0”;
ns date = “http://exslt.org/dates-and-times”;
import “../import/junos.xsl”;

match configuration {
    /* Get current hour of day */
    var $hour = date:hour-in-day();
```

```
   /* Retrieve the candidate configuration - so we see the inactive statements */
   var $login-config = jcs:invoke( “get-configuration” )/system/login;

   /* Record if active or not */
   var $day-user-active = not( jcs:empty( $login-config/user[name == “day-shift”][ jcs:empty( @
inactive ) ] ) );

   var $night-user-active = not( jcs:empty( $login-config/user[name == “night-shift”][ jcs:empty( @
inactive ) ] ) );

   var $day-user-inactive = not( jcs:empty( $login-config/user[name == “day-shift”][ @inactive ] )
);

   var $night-user-inactive = not( jcs:empty( $login-config/user[name == “night-shift”][  @inactive
] ) );

   /* Verify that login class is set */
   if( jcs:empty( $login-config/user[ name == “day-shift” ][ class == “super-user” ] ) ) {
      <change> {
         <system> {
            <login> {
               <user> {
                  <name> “day-shift”;
                  <class> “super-user”;
               }
            }
         }
      }
   }
   if( jcs:empty( $login-config/user[ name == “night-shift” ][ class == “super-user” ] ) ) {
      <change> {
         <system> {
            <login> {
               <user> {
                  <name> “night-shift”;
                  <class> “super-user”;
               }
            }
         }
      }
   }

   /* Verify that a password is set */
   if(jcs:empty($login-config/user[name == “day-shift”][authentication/encrypted-password])){
      <change> {
         <system> {
            <login> {
               <user> {
                  <name> “day-shift”;
                  <authentication> {
                     <encrypted-password> “$1$P1k.8T27$291PKVguOvXpbIn2/vV8V.”;
                  }
               }
            }
         }
      }
   }
   if(jcs:empty($login-config/user[name == “night-shift”][authentication/encrypted-password])){
      <change> {
         <system> {
            <login> {
               <user> {
```

```
                    <name> “night-shift”;
                    <authentication> {
                        <encrypted-password> “$1$P1k.8T27$291PKVguOvXpbIn2/vV8V.”;
                    }
                }
            }
        }
    }

    /* Build the necessary configuration change - based on the time of day*/
    /* Daytime? */
    if( $hour >= 8 && $hour < 16 ) {
        if( not( $day-user-active ) ) {
            <change> {
                <system> {
                    <login> {
                        <user active=”active”> {
                            <name> “day-shift”;
                        }
                    }
                }
            }
        }
        if( not( $night-user-inactive ) ) {
            <change> {
                <system> {
                    <login> {
                        <user inactive=”inactive”> {
                            <name> “night-shift”;
                        }
                    }
                }
            }
        }
    }
    /* Nighttime */
    else {
        if( not( $day-user-inactive ) ) {
            <change> {
                <system> {
                    <login> {
                        <user inactive=”inactive”> {
                            <name> “day-shift”;
                        }
                    }
                }
            }
        }
        if( not( $night-user-active ) ) {
            <change> {
                <system> {
                    <login> {
                        <user active=”active”> {
                            <name> “night-shift”;
                        }
                    }
                }
            }
        }
    }
}
```

Beyond verifying that the correct account is active or inactive, the commit script also checks that the login class is set correctly and a password is present. If neither is the case, then the script makes the appropriate change. It then checks that the correct user is marked active and that the correct user is marked inactive. To do this, the script requests the configuration via `<get-configuration>`. This is done because the configuration provided to commit scripts is post-inheritance, so inactive statements are not present, but by requesting the configuration, the pre-inheritance configuration can be considered and the appropriate statements added if they are lacking.

NOTE The script assumes that all user configurations are within the main system hierarchy instead of within a configuration group.

Summary

These are the essential topics to help you refine your scripting capabilities. There are some great items in the appendix, and there are more books for you in the *Day One / This Week* library on Junos Automation at www.juniper.net/dayone.

Hopefully, the many short tutorials and examples in this book have helped you get the knack of Junos automation scripting with SLAX.

TIP Look for a Copy and Paste Edition of this book and the other Junos Automation books at each book's detailed page at www.juniper.net/dayone. Its rich text format can allow you to easily copy and paste the script

Appendix

Using the Op Script on Target Debugger

An op script debugger, providing greater debugging options for both XSLT and SLAX op scripts, was introduced in Junos 10.4. This debugger is capable of processing the script line-by-line, displaying the script's variable values at a given point in the script operation, and showing the template back trace at a particular location. The command to invoke the debugger is currently hidden, so the debugger is not officially supported and is not appropriate for production use, but it is still a valuable tool to use during script development on test devices.

To invoke the debugger, add `invoke-debugger` to the command-line while executing an op script, followed by the selection of either CLI or remote debugging:

```
user@junos> op test invoke-debugger ?
Possible completions:
  cli              Invoke debugger in cli
  remote            Invoke debugger for remote client to attach
```

NOTE Only CLI based debugging is covered in this appendix.

NOTE Because the command is hidden, `invoke-debugger` must be typed in completely without relying on command auto-completion.

ALERT! The debugger cannot be used with op scripts that are executed via `op url`.

Once started, the debugger identifies and displays the first script line to be processed. Execution is halted at that point and the CLI user is given a debug prompt, waiting for instructions on what to do next. The first line of the script that is processed is either the first global variable to be initialized, or it is the match / template if no global variables are present.

NOTE The global variables are not always initialized in document order. In particular, if one global variable's value depends on the value of another global variable, then the latter variable is initialized first, even if it appears on a later line in the script file.

As of Junos 10.4R2, a global variable has been added into the *junos.xsl* import file named *$junos-context*, and the initialization of this variable is often, but not always, the first line executed by the debugger anytime the junos.xsl file is being imported. For example:

```
user@junos> op test invoke-debugger cli
Welcome to Script Debugger
Type 'help' for help

junos.xsl:31      event-script-input/junos-context"/>
sdbg>
```

As shown above, the first line that the debugger executes comes from the junos.xsl import file. This can be determined because the script file's name is included in the description of the current code line (above the debugger prompt), along with its line number within that script file (`31` in this example) and the actual code line itself.

In the preceding example, this initial line might appear strange because it is expressed in XSLT and only shows a portion of the code line, but the reason for this can be seen by examining the junos.xsl import file:

```
  <xsl:variable name="junos-context"
     select="op-script-input/junos-context |
     commit-script-input/junos-context |
     event-script-input/junos-context"/>
```

As can be seen in the above code snippet from junos.xsl, the initial line that is displayed within the debugger is actually the third line of a multi-line variable declaration for `$junos-context`, and it is expressed in XSLT because the junos.xsl import file is written in XSLT.

If junos.xsl is not imported, and there are no global variables, then the initial starting point is match / within the op script itself:

```
user@junos> op test invoke-debugger cli
Welcome to Script Debugger
Type 'help' for help

test.slax:8 match / {
sdbg>
```

Once the debugger has started, it provides a prompt: "sdbg>", which stands for script debugger. Entering the command help displays all of the available commands that can be entered at the prompt:

```
junos.xsl:31      event-script-input/junos-context"/>
sdbg> help
Supported commands ...
break [line] [file:line] [template] -> Put a breakpoint at certain point
continue -> Continue running the script
delete [num] -> Delete breakpoints
help -> Print this help message
next -> Execute the next line
print <var-name> -> Print the current value of a variable
where -> Print the backtrace of template calls
quit -> Quit debugger
```

Break Command

The break command is used to establish a breakpoint at a certain line or template. A script writer can set a breakpoint where they wish to investigate the code operation and then allow the script to run up until that point by entering the continue command. Breakpoints can be set in four ways:

1. Specifying the current line:

```
test.slax:15       var $reply = jcs:get-input("Interface-name: ");
sdbg> break
Breakpoint 1 at file /var/db/scripts/op/test.slax, line 15
```

2. Specifying a line in the current script:

```
hello-world.slax:9 match / {
sdbg> break 10
Breakpoint 1 at file /var/db/scripts/op/hello-world.slax, line 10
```

3. Specifying a line within a specific script file, which must be either the invoked op script or one of its import files:

```
junos.xsl:31      event-script-input/junos-context"/>
sdbg> break hello-world.slax:10
Breakpoint 1 at file /var/db/scripts/op/hello-world.slax, line 10
```

4. Specifying a template's name, to cause a breakpoint to be set for whenever the template is invoked. If the template has a preceding namespace prefix then the prefix must not be included. For example, rather than saying `jcs:load-configuration` say `load-configuration`:

```
junos.xsl:31      event-script-input/junos-context"/>
sdbg> break to-upper
Breakpoint 1 at file /var/db/scripts/op/test.slax, line 17
```

Breakpoints can only be added to lines that have script instructions, meaning that they cannot be set to blank lines or lines that contain just comments or strings.

NOTE When the debugger first starts, it is often executing the $junos-context global variable within the junos.xsl import file, so any breakpoints within the op script file itself must be set using the file:line method until the script execution has moved from the junos.xsl script file into the op script's file.

Continue Command

The continue command executes the script until the next breakpoint is reached or until the script terminates. It is typically used in conjunction with the break command as breakpoints are set first, and then the continue command causes the script to execute until reaching them:

```
junos.xsl:31      event-script-input/junos-context"/>
sdbg> break test.slax:11
Breakpoint 2 at file /var/db/scripts/op/test.slax, line 11

junos.xsl:31      event-script-input/junos-context"/>
sdbg> continue
Reached breakpoint 2, at /var/db/scripts/op/test.slax:11

test.slax:11        var $config = jcs:invoke( "get-configuration" );
sdbg>
```

Delete Command

If entered by itself, the delete command deletes all of the current breakpoints; otherwise, a specific breakpoint number to delete can be entered:

```
junos.xsl:31      event-script-input/junos-context"/>
sdbg> break test.slax:11
Breakpoint 1 at file /var/db/scripts/op/test.slax, line 11

junos.xsl:31      event-script-input/junos-context"/>
sdbg> break test.slax:15
Breakpoint 2 at file /var/db/scripts/op/test.slax, line 15

junos.xsl:31      event-script-input/junos-context"/>
sdbg> delete 2
Deleted breakpoint '2'

junos.xsl:31      event-script-input/junos-context"/>
sdbg> delete
Delete all breakpoints? (yes/no) yes
Deleted all breakpoints
```

Next Command

The next command executes the following script statement, which is generally on the next line, but at times multiple next commands must be entered to move past a single line. This command can be used to step through a script's processing line-by-line:

```
jnpr@srx210> op test invoke-debugger cli
Welcome to Script Debugger
Type 'help' for help

junos.xsl:31      event-script-input/junos-context"/>
sdbg> next

test.slax:9 match / {
sdbg> next

test.slax:9 match / {
sdbg> next

test.slax:10    <op-script-results> {
sdbg> next

test.slax:11       var $config = jcs:invoke( "get-configuration" );
sdbg>
```

Print Command

The value of a variable or parameter can be displayed through the print command, which displays the data type, whether it is global or local, and the variable's content.

```
test.slax:19       <output> $results;
sdbg> print $rpc
(Local) rpc => (RTF)
<get-configuration>
  <configuration>
    <interfaces/>
  </configuration>
</get-configuration>

test.slax:19       <output> $results;
sdbg> print $interface-name
(Local) interface-name => ge-0/0/0 (String)

test.slax:19       <output> $results;
sdbg> print $user
(Global) user => jnpr (String)

test.slax:19       <output> $results;
sdbg> print $junos-context
(Global) junos-context => (Nodeset)
<junos-context>
<hostname>srx210</hostname>
<product>srx210h</product>
<localtime>Fri Apr 22 10:11:25 2011</localtime>
<localtime-iso>2011-04-22 10:11:25 UTC</localtime-iso>
<script-type>op</script-type>
<pid>2664</pid>
<tty>/dev/ttyp1</tty>
<chassis>others</chassis>
<routing-engine-name>re0</routing-engine-name>
<re-master/>
```

```
<user-context>
<user>jnpr</user>
<class-name>j-super-user</class-name>
<uid>2001</uid>
<login-name>jnpr</login-name>
</user-context>
<op-context>
</op-context>
</junos-context>
```

If a global variable or parameter is overridden by a local one, then only the local is displayed.

Where Command

A backtrace of the current template can be displayed through the where command. Both match templates and named templates are included, but custom functions, created using the <func:function> extension element, are not reported within the backtrace.

```
junos.xsl:255   <xsl:template name=”jcs:load-configuration”>
sdbg> where
#0 load-configuration() from test.slax:21
#1 change-configuration() from test.slax:11
#2 /
```

The backtrace shows the current template first, followed by the calling templates in reverse order. The script file and line number where the template was invoked is included on each line as well.

Quit Command

The quit command halts the debugger. It does not terminate the script but rather causes script operation to continue uninterrupted from the current code line.

Shortcuts

Pressing the enter key, without any input, causes the prior command to be executed again:

```
user@junos> op test invoke-debugger cli
Welcome to Script Debugger
Type ‘help’ for help

junos.xsl:31      event-script-input/junos-context”/>
sdbg> next

test.slax:9 match / {
sdbg>

test.slax:9 match / {
sdbg>

test.slax:10    <op-script-results> {
sdbg>

test.slax:11        call change-configuration();
sdbg>

test.slax:15 template change-configuration() {
```

```
sdbg>

test.slax:15 template change-configuration() {
sdbg>

test.slax:16    var $config = {
sdbg>

test.slax:17        <configuration> {
sdbg>
```

The full commands shown above do not have to be entered – instead, just the initial character is required:

- b – break
- c – continue
- d – delete
- h – help
- n – next
- p – print
- w – where
- q - quit

```
junos.xsl:31      event-script-input/junos-context"/>
sdbg> b test.slax:11
Breakpoint 1 at file /var/db/scripts/op/test.slax, line 11

junos.xsl:31      event-script-input/junos-context"/>
sdbg> c
Reached breakpoint 1, at /var/db/scripts/op/test.slax:11

test.slax:11        var $ns := { <node>; }
sdbg> n

test.slax:11        var $ns := { <node>; }
sdbg> n

test.slax:11        var $ns := { <node>; }
sdbg> n

test.slax:12        apply-templates $ns/node;
sdbg> p $ns
(Local) ns => (Nodeset)
<node/>
```

Debugging Example

The script change-host-name.slax will here be used to demonstrate a debugging session. Line numbers are included in the code example to highlight the breakpoints that are added:

```
 1: version 1.0;
 2:
 3: ns junos = "http://xml.juniper.net/junos/*/junos";
 4: ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
 5: ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
```

```
 6:
 7: import "../import/junos.xsl";
 8:
 9: match / {
10:
11:    <op-script-results> {
12:
13:       var $host-name = jcs:get-input( "Enter new host-name: " );
14:
15:       var $configuration = {
16:          <configuration> {
17:             <system> {
18:                <host-name> $host-name;
19:             }
20:          }
21:       }
22:
23:       var $connection = jcs:open();
24:       var $results := { call jcs:load-configuration($connection, $configuration); }
25:       copy-of $results;
26:       expr jcs:close($connection);
27:    }
28: }
```

The debugger is first started, and breakpoints are added, for line 24 of change-host-name.slax as well as for the jcs:load-configuration template.

```
user@junos> op change-host-name invoke-debugger cli
Welcome to Script Debugger
Type 'help' for help

junos.xsl:31      event-script-input/junos-context"/>
sdbg> break change-host-name.slax:24
Breakpoint 1 at file /var/db/scripts/op/change-host-name.slax, line 24

junos.xsl:31      event-script-input/junos-context"/>
sdbg> break load-configuration
Breakpoint 2 at file /var/db/scripts/import/junos.xsl, line 255
```

The script processing is then allowed to continue, causing the input prompt from the jcs:get-input() function to be displayed. Once the new hostname is entered, the script continues on to the first breakpoint:

```
junos.xsl:31      event-script-input/junos-context"/>
sdbg> continue
Enter new host-name: lab-srx
Reached breakpoint 1, at /var/db/scripts/op/change-host-name.slax:24

change-host-name.slax:24       var $results := { call jcs:load-configuration($connection,
$configuration); }
sdbg>
```

At this point, the following variables might be of interest: `$host-name`, `$configuration`, and `$connection`, so they are each displayed by using the print command:

```
change-host-name.slax:24       var $results := { call jcs:load-configuration($connection,
$configuration); }
sdbg> print $host-name
(Local) host-name => lab-srx (String)

change-host-name.slax:24       var $results := { call jcs:load-configuration($connection,
$configuration); }
sdbg> print $configuration
(Local) configuration => (RTF)
```

```
<configuration>
  <system>
    <host-name>lab-srx</host-name>
  </system>
</configuration>

change-host-name.slax:24       var $results := { call jcs:load-configuration($connection,
$configuration); }
sdbg> print $connection
(Local) connection => (Nodeset)
<cookie></cookie>
```

The script is then allowed to continue until it reaches the jcs:load-configuration breakpoint. At that point, the code is stepped through the template's processing, and the $load-config variable, which is defined within the jcs:load-configuration template, is checked:

```
change-host-name.slax:24       var $results := { call jcs:load-configuration($connection,
$configuration); }
sdbg> continue
Reached breakpoint 2, at /var/db/scripts/import/junos.xsl:255

junos.xsl:255  <xsl:template name="jcs:load-configuration">
sdbg> next
Reached breakpoint 2, at /var/db/scripts/import/junos.xsl:255

junos.xsl:255  <xsl:template name="jcs:load-configuration">
sdbg> next

junos.xsl:261    <xsl:choose>
sdbg> next

junos.xsl:262      <xsl:when test="not($connection) or not($configuration)">
sdbg> next

junos.xsl:267      <xsl:otherwise>
sdbg> next

junos.xsl:268        <xsl:variable name="options-temp-1">
sdbg> next

junos.xsl:269          <xsl:copy-of select="$commit-options"/>
sdbg> next

junos.xsl:271        <xsl:variable name="options" select="ext:node-set($options-temp-1)"/>
sdbg> next

junos.xsl:272        <xsl:variable name="commit-options-error-temp-2">
sdbg> next

junos.xsl:273          <xsl:choose>
sdbg> next

junos.xsl:274            <xsl:when test="$options/commit-options">
sdbg> next

junos.xsl:287            <xsl:otherwise>
sdbg> next

junos.xsl:288              <xsl:if test="$commit-options">
sdbg> next

junos.xsl:296        <xsl:variable name="commit-options-error" select="ext:node-set($commit-
options-error-temp-2)"/>
```

```
sdbg> next

junos.xsl:297       <xsl:choose>
sdbg> next

junos.xsl:298         <xsl:when test="jcs:empty($commit-options-error/*)">
sdbg> next

junos.xsl:299           <xsl:variable name="lock-result" select="jcs:execute($connection, 'lock-
configuration')"/>
sdbg> next

junos.xsl:301           <xsl:copy-of select="$lock-result/..//xnm:warning"/>
sdbg> next

junos.xsl:303           <xsl:choose>
sdbg> next

junos.xsl:304             <xsl:when test="$lock-result/..//xnm:error">
sdbg> next

junos.xsl:307             <xsl:otherwise>
sdbg> next

junos.xsl:309               <xsl:variable name="load-config">
sdbg> next

junos.xsl:310                 <load-configuration format="xml" action="{$action}">
sdbg> next

junos.xsl:311                   <xsl:copy-of select="$configuration"/>
sdbg> next

junos.xsl:314               <xsl:variable name="load-result" select="jcs:execute($connection,
$load-config)"/>
sdbg> print $load-config
(Local) load-config => (RTF)
<load-configuration format="xml" action="">
  <configuration>
    <system>
      <host-name>lab-srx</host-name>
    </system>
  </configuration>
</load-configuration>

junos.xsl:314               <xsl:variable name="load-result" select="jcs:execute($connection,
$load-config)"/>
sdbg>
```

Now, rather than continuing to step through the jcs:load-configuration template, an additional breakpoint is added in the change-host-name.slax file, so that the results of the template can be analyzed. The script processing is then allowed to continue until that point and the `$results` variable are displayed:

```
junos.xsl:314               <xsl:variable name="load-result" select="jcs:execute($connection,
$load-config)"/>
sdbg> break change-host-name.slax:25
Breakpoint 3 at file /var/db/scripts/op/change-host-name.slax, line 25

junos.xsl:314               <xsl:variable name="load-result" select="jcs:execute($connection,
$load-config)"/>
sdbg> continue
Reached breakpoint 3, at /var/db/scripts/op/change-host-name.slax:25
```

```
change-host-name.slax:25        copy-of $results;
sdbg> print $results
(Local) results => (Nodeset)
<commit-configuration/>
```

No specific success message is provided from jcs:load-configuration; rather, the absence of <xnm:error> elements indicates that the commit was successful.

NOTE The inclusion of the <commit-configuration> Junos API element within the results from jcs:load-configuration is a bug in the current code and can be ignored.

The debugging session can now be closed by using the `quit` command, allowing the script to finish processing normally.

```
change-host-name.slax:25        copy-of $results;
sdbg> quit

user@junos>
```

Using the <libxslt:debug> Debug Element

The op script debugger provides good debugging capabilities to op scripts, but is not supported for commit scripts or event scripts; however, another option exists to provide some minimal debugging capability to these other script types: the <libxslt:debug> extension element, which lacks much of the debugging capability of the op script debugger but is capable of displaying the template backtrace at a particular point in the script's code.

Before using the <libxslt:debug> extension element, the `libxslt` namespace must first be defined as an extension namespace:

```
ns libxslt extension = "http://xmlsoft.org/XSLT/namespace";
```

Next, the element is simply added wherever the debug output is desired.

```
var $ns := {
    <node> "one";
    <node> "two";
    <node> "three";
}
<libxslt:debug>;
apply-templates $ns/node;
```

When the script processor reaches the <libxslt:debug> element within the script, it realizes that it should be treated as an extension element rather than XML data (because the `libxslt` namespace was defined as an extension namespace), so it executes the operation associated with that element, causing the template backtrace at that current point in the script's code to be displayed to the CLI user as an error message, and also to be logged to the script's trace file.

NOTE Event scripts have to configure an output file in order to see the results of <libxslt:debug> (other than within the trace file), and if the output file is in XML format then the results will be included as <xnm:error> messages.

This is an example of an op script that is configured to use the <libxslt:debug> element, and the results that are displayed to the user:

```
version 1.0;

ns junos = "http://xml.juniper.net/junos/*/junos";
ns xnm = "http://xml.juniper.net/xnm/1.1/xnm";
ns jcs = "http://xml.juniper.net/junos/commit-scripts/1.0";
ns libxslt extension = "http://xmlsoft.org/XSLT/namespace";

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

match / {
    call template-1();
}

template template-1() {
    call template-2();
}

template template-2() {
    call template-3();
}

template template-3() {
    call template-4();
}

template template-4() {
    call template-5();
}

template template-5() {
    <libxslt:debug>;
}
```

```
jnpr@srx210> op libxslt_debug
error: Templates:
error: #0
error: name template-5
error: #1
error: name template-4
error: #2
error: name template-3
error: #3
error: name template-2
error: #4
error: name template-1
error: #5
error: name /
error: Variables:
```

ALERT! The output from `<libxslt:debug>` is interpreted as a commit error by commit scripts, causing the commit operation to fail, so it should only be included while troubleshooting as it will always cause the commit process to halt.

Any variables or parameters that are defined within the current template are displayed as well, though their values are not included in the output, making this section less important than the template backtrace:

```
user@junos> op test
error: Templates:
error: #0
error: name /
error: Variables:
```

```
error: #0
error: var
error: ns
```

Although the information provided by `<libxslt:debug>` is minimal, it might be a useful debugging tool for script writers who wish to better understand the template calling order in their commit script-or event script-because neither of those script types supports the normal script debugger. But, when working with op scripts in Junos 10.4 or later, the op script debugger would be the better debugging option.

Using the slaxproc Utility

The slaxproc utility is available for developers to use on their host computers (Unix/Mac/Windows). It provides a number of features and capabilities including debugging SLAX code, converting between XSLT and SLAX, and performing syntax checking. Figure A.1 shows the slaxproc utility with its output from the help option.

TIP To download the slaxproc utility and its complete feature set go to http://code.google.com/p/libslax/downloads/detail?name=libslax-0.2.2.tar.gz.

```
$ slaxproc --help
Usage: slaxproc [options] [stylesheet] [file]
    Options:
        --slax-to-xslt OR -x: turn SLAX into XSLT
        --xslt-to-slax OR -s: turn XSLT into SLAX
        --run OR -r: run a SLAX script
        --check OR -c: check syntax and content for a SLAX
script
        --debug OR -d: enable the SLAX/XSLT debugger
        --exslt OR -e: enable the EXSLT library
        --help OR -h: display this help message
        --input <file> OR -i <file>: take input from the given
file
        --name <file> OR -n <file>: read the script from the
given file
        --output <file> OR -o <file>: make output into the
given file
        --param name value OR -a name value: pass parameters
        --partial OR -p: allow partial SLAX input to --slax-to-
xslt
```

Figure A.1 The slaxproc Utility in Action

One of the more common dev-cycle activities for using the slaxproc utility is debugging SLAX syntax errors. Common errors in creating automation scripts are forgetting a curly-brace or a semi-colon. Rather than copying the script to a Junos device and attempting to run the script only to discover syntax errors, you can use `slaxproc` to check for syntax errors prior to copying the script to the device. It can save significant amounts of time. For now, let's turn to Hello, World:

```
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!";
```

```
      /* ERROR: missing } to terminate <op-script-results> */
}
```

Notice that there should be a right-curly (}) on line nine, but it is missing. Running this file through `slaxproc` using the `--check` option results in the following:

```
admin@myPC$ slaxproc --check hello-world.slax
hello-world.slax:13: error: hello-world.slax:12: syntax error, unexpected $end(null)
hello-world.slax:13: error: hello-world.slax: 1 error detected during parsing
slaxproc cannot parse: 'hello-world.slax'
```

Correcting the mistake on line nine results in a successful result:

```
admin@myPC$ slaxproc --check hello-world.slax
script check succeeds
```

You will also need to convert XSLT code into SLAX code, a common use-case being when adapting an XSLT cookbook recipe that you've found, or were given, into SLAX.

The following XSLT cookbook recipe defines a template that is used to find the index of a substring within a string:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template name="string-index-of">
  <xsl:param name="input"/>
  <xsl:param name="substr"/>
<xsl:choose>
  <xsl:when test="contains($input, $substr)">
    <xsl:value-of select="string-length(substring-before($input, $substr))+1"/>
  </xsl:when>
  <xsl:otherwise>0</xsl:otherwise>
</xsl:choose>
</xsl:template>

</xsl:sytlesheet>
```

…and the slaxproc utility can quickly convert this to SLAX using the "`-s`" and "`-p`" flags as illustrated here:

```
template string-index-of ($input, $substr) {
   if (contains($input, $substr)) {
      expr string-length(substring-before($input, $substr)) + 1;

   } else {
      expr "0";
   }
}
```

NOTE This could also be done on the CLI through the `request system scripts convert` command.

"Man page" for jcs:printf()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

string jcs:printf(string [format])

string jcs:printf(string [format], string* [arguments])

Description

The jcs:printf() function returns a formatted string that is created by inserting string arguments into a format string. The returned string can then be displayed through a standard output function such as jcs:output(), because jcs:printf() only creates the formatted string; it does not display it.

The first argument is the format string, followed by a variable number of string arguments, which are inserted into the format string according to its included format specifications. The format string consists of plain text as well as format specifications. Plain text is copied directly from the format string to the returned formatted string. Format specifications begin with '%' and end with 's' and are used to insert string arguments into the returned string.

Function call

```
jcs:printf( "First: %s Second: %s", "1st", "2nd" )
```

Returned string

```
"First: 1st Second: 2nd"
```

To include a '%' character in the formatted text, include '%%' in the format string.

Function call

```
jcs:printf( "%s%%", "50" )
```

Returned string

```
"50%"
```

Format specifications consist of optional flags, width, and precision, and are terminated by a mandatory conversion specifier, which should be set to 's' because only string arguments are supported, but any of the following characters will terminate the format specification as well: d, e, E, f, g, G, i, o, u, x. These alternate characters are handled the same as 's', however, because only strings are supported, so it is best to always terminate format specifications with 's' to prevent confusion with script writers that are accustomed to the *printf* function in other programming languages.

Each format specification corresponds to a string argument, so there must be at least as many string arguments as there are format specifications (the below sections on width and precision include scenarios where there will be more arguments than format specifications), and the arguments are inserted in the order that the format specifications appear in the format string.

Multiple flags can be used within the same format specification. These are the supported flags:

- The minus sign flag '-' causes the string to be left-aligned. Alignment determines whether padding is appended or prepended to the string. A right-aligned string, which is the default, has padding prepended, but a left-aligned string, which is indicated by using the '-' flag, has padding appended.

- Alignment has no effect on string truncation. Right truncation is always performed when a specified precision forces the string to be truncated.

Function call

```
jcs:printf( "|%-10s|", "Left" )
```

Returned string

```
"|Left     |"
```

A zero flag '0' causes the string to be padded by zeros rather than spaces, but it is only valid for right-aligned strings and is ignored if the '-' flag is present.

Function call

```
jcs:printf( "%08s", "5" )
```

Returned string

```
"00000005"
```

The j1 flag 'j1' is used to indicate that a string argument should not be inserted if the preceding call to jcs:printf() used the same format string, and the string argument's value has not changed. This flag is useful if a duplicate column value should not be displayed in subsequent rows.

Function call

```
jcs:printf( "%j1-10s %s", "xe-0/0/0", "inet" )
jcs:printf( "%j1-10s %s", "xe-0/0/0", "inet6" )
jcs:printf( "%j1-10s %s", "xe-1/0/0", "inet" )
```

Returned string

```
"xe-0/0/0  inet"
"         inet6"
"xe-1/0/0  inet"
```

The jc flag 'jc' causes the first letter of the string argument to be capitalized.

- The first character of the string is capitalized, following minimum width padding but prior to tag prepending, so capitalization will not work for right-aligned strings that have spaces or zeros prepended due to their minimum width.

Function call

```
jcs:printf( "%jcs", "example" )
```

Returned string

```
"Example"
```

The jt{TAG} flag is used to prepend a tag string to the inserted argument string if the argument string is not blank. The tag string consists of all the characters inside the { } curly brackets within the flag.

- A '}' cannot be included within the tag string, because it terminates the tag string.
- If the argument is a node-set or a result tree fragment that has element nodes but no text content, then it will be converted into a blank string, so the tag string will not be added.
- The tag is prepended after the string argument has been formatted to the correct width and precision, which could cause columns to not align correctly, so this

flag should only be used within the last format specification of a row if column alignment must be maintained over multiple rows.

Function call

```
jcs:printf( “%-10s %jt{>>}12s”, “10.0.0.1”, “192.168.1.1” )
jcs:printf( “%-10s %jt{>>}12s”, “10.0.0.1”, ““ )
```

Returned string

```
“10.0.0.1  >> 192.168.1.1”
“10.0.0.1            “
```

The minimum width to which a string will be padded, if necessary, can be indicated by including a numeric value within the format specification, or a * can be included, indicating that the width should be taken from the argument list rather than from the format specification; this causes the next argument in the argument list to be converted into an integer, which is used as the width, and the following argument is used as the string argument that is inserted into the string.

Function call

```
jcs:printf( “|%10s|%-10s|”, “Right”, “Left” )
jcs:printf( “|%*s|%-*s|”, 5, “1”, 3, “2” )
```

Returned string

```
“|    Right|Left     |”
“|   1|2  |”
```

The precision of the string, or the maximum length, is indicated by including a period followed by a numeric value within the format specification. If the string argument is longer than the indicated precision, then the string will be right-truncated. A * can be included, instead of a number, to indicate that the precision should be taken from the argument list rather than from the format specification. This causes the next argument in the argument list to be converted into an integer, which is used as the precision, and then the following field is used as the string argument that is inserted into the string.

Function call

```
jcs:printf( “|%5.5s|%.4s|”, “1234567890”, “abcdefg” )
jcs:printf( “|%*.*s|”, 5, 1, “12345” )
```

Returned string

```
“|12345|abcd|”
“|   1|”
```

The following escape characters can be used:

- \n – Newline
- \r – Carriage Return
- \t - Tab
- \\ - Backslash (As of Junos 10.2)
- \” – Double-quote (As of Junos 10.1R2)
- \’ – Single-quote

Example

This op script demonstrates how to create formatted output with the jcs:printf() function.

Code

```
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> {

      /* Display the parameters */
      expr jcs:output( jcs:printf( "%15s %-10s", "Parameter", "Value" ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$user", $user ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$hostname", $hostname ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$product", $product ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$script", $script ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$localtime", $localtime ) );
      expr jcs:output( jcs:printf( "%15s %-10s", "$localtime-iso", $localtime-iso ) );

      /* Retrieve string to display */
      var $string = jcs:get-input( "Enter string: " );

      /* Retrieve width */
      var $width = jcs:get-input( "Enter width: " );

      /* Retrieve precision */
      var $precision = jcs:get-input( "Enter precision: " );

      expr jcs:output( jcs:printf( "|%*.*s|", $width, $precision, $string ) );
   }
}
```

Output

```
     Parameter Value
        $user jnpr
     $hostname srx210
      $product srx210h
       $script jcs_printf.slax
    $localtime Tue Jun 21 09:56:27 2011
 $localtime-iso 2011-06-21 09:56:27 UTC
Enter string: 1234567890
Enter width: 5
Enter precision: 3
|  123|
```

"Man Page" for jcs:regex()

Source: Junos
Namespace: http://xml.juniper.net/junos/commit-scripts/1.0
Common Prefix: jcs
Minimum Version: Junos 8.2

Syntax

node-set [matches] jcs:regex(string [pattern], string [target])

Description

The jcs:regex() function is used to match strings based on a regular expression. The first string argument is the regular expression pattern in POSIX extended regular expression format, and the second string argument is the target string that is searched for a match. A node-set is returned, which contains the string that matches the entire regular expression, as well as up to eight included sub-expressions.

Supported Regular Expression Operators

Match any character - "." – Matches any character, including newlines.

Function call

```
jcs:regex( "srx2.0", "srx210" )
```

Returned node-set

```
<match> "srx210"
```

Match zero or more – "*" – Matches the prior character or subexpression zero or more times.

Function call

```
jcs:regex( "srx.*", "srx210" )
```

Returned node-set

```
<match> "srx210"
```

Match one or more – "+" – Matches the prior character or subexpression one or more times.

Function call

```
jcs:regex( "s+rx210", "srx210" )
```

Returned node-set

```
<match> "srx210"
```

Match zero or one times – "?" – Matches the prior character or subexpression either zero or one times.

Function call

```
jcs:regex( "srx21?0", "srx210" )
```

Returned node-set

```
<match> "srx210"
```

Matching interval – "{ ... }" – Comes in three forms:

- Match exactly N times: {N}
- Match N times or more: {N,}
- Match between N1 and N2 times: {N1,N2}

- N/N1/N2 must be 255 or less

Function call

```
jcs:regex( “srx[0-9]{3}”, “srx210” )
```

Returned node-set

```
<match> “srx210”
```

Alternation – “|” – Matches one or the other regular expressions. The longer left-most match is preferred.

Function call

jcs:regex(“20|2011”, “2011-02-22”)

Returned node-set

```
<match> “2011”
```

Matching list – “[...]” – Matches one character from within the list.

- Ranges are expressed by using the hyphen:

Function call

```
jcs:regex( “srx[0-9]*”, “srx210” )
```

Returned node-set

```
<match> “srx210”
```

- Most special characters within a list are treated as normal characters (*, +, ?, etc) and do not require escaping.
- To include a hyphen “-” as a normal character, include it as the first or last character.
- To include a closing bracket “]” as a normal character, include it as the first character.

Non-matching list – “[^ ...]” – Matches any character that is not included in the list. The syntax rules are the same as a matching list.

Function call

```
jcs:regex( “[^ 0-9]*”, “JUNOS 10.4R1.9” )
```

Returned node-set

```
<match> “JUNOS”
```

Grouping – “(...)” – Creates a group or subexpression, allowing operators to be applied to groups rather than characters, and returns the matching value as part of the returned node-set.

Function call

```
jcs:regex( “([0-9.]*).([0-9.]*)”, “10.4R1.9” )
```

Returned node-set

```
<match> “10.4R1.9”;
```

```
<match> “10.4”;
<match> “1.9”;
```

Supported regular expression anchors:

Start of string – “^” – Anchors the regular expression to the start of the string.

Function call

```
jcs:regex( “^[^ ]*”, “JUNOS Software Release [10.4R1.9]” )
```

Returned node-set

```
<match> “JUNOS”
```

End of string – “$” – Anchors the regular expression to the end of the string.

Function call

```
jcs:regex( “[^ ]*$”, “JUNOS Software Release [10.4R1.9]” )
```

Returned node-set

```
<match> “[10.4R1.9]”
```

Start of word –”[[:<:]]” – Anchors the regular expression to the start of a word, where a word is a sequence of alphanumeric characters or underscores.

Function call

```
jcs:regex( “[[:<:]]S[^ ]*”, “JUNOS Software Release [10.4R1.9]” )
```

Returned node-set

```
<match> “Software”
```

End of word –”[[:>:]]” – Anchors the regular expression to the end of a word, where a word is a sequence of alphanumeric characters or underscores.

Function call

```
jcs:regex( “.*S[[:>:]]”, “JUNOS Software Release [10.4R1.9]” )
```

Returned node-set

```
<match> “JUNOS”
```

Escaping

Most standard escape characters are expressed in the normal fashion: \n \t \r \” \’

- Exception is \\ which must be expressed as \\\\ within a regular expression, except within a list. This is necessary because the escape must be present both for the SLAX to XSLT conversion as well as for the jcs:regex() processing. In other words, the SLAX to XSLT conversion translates “\\\\” to “\\”, which jcs:regex() treats correctly as an escaped backslash.

Function call

```
jcs:regex( “1\\\\2”, “1\\2” )
```

Returned node-set

```
<match> “1\2”
```

- Double-quote escape \” is supported as of Junos 10.1R2.

Regular expression special characters (operators and anchors) must be escaped with two backslashes, because the SLAX to XSLT conversion will remove one of them.

Function call

```
jcs:regex( "\\.\\^\\$\\*\\+\\(\\)\\[\\]\\{\\}\\|\\?", ".^$*+()[]{}|?" )
```

Returned node-set

```
<match> ".^$*+()[]{}|?"
```

Character classes can be used within a list to indicate a class of characters that should be included instead of specifying the individual characters. The following character classes are supported by jcs:regex():

- [:alnum:] – Alphanumeric characters: A-Z, a-z, and 0-9
- [:alpha:] – Alpha characters: A-Z, and a-z
- [:blank:] – Blank characters: space and tab
- [:cntrl:] – Control characters: ASCII values 0x0-0x19 and 0x7F
- [:digit:] – Numeric characters: 0-9
- [:graph:] – Printable characters, except for space: ASCII values 0x21-0x7E
- [:lower:] – Lowercase letters: a-z
- [:print:] – Printable characters: ASCII values 0x20-0x7E
- [:punct:] – Punctuation: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~
- [:space:] – Whitespace: space, tab, newline, carriage return, vertical tab, form feed
- [:upper:] – Uppercase letters: A-Z
- [:xdigit:] – Hexadecimal digit characters: A-F, a-f, 0-9

Function call

```
jcs:regex( "[[:alnum:][:punct:]]*", "*[Direct/0] 05:08:47" )
```

Returned node-set

```
<match> "*[Direct/0]"
```

While no multi-character collating sequences are supported by jcs:regex(), single character collating elements can be used within lists and could be used to represent special characters or to indicate a specific control character. Collating elements for each non-null ASCII character can be created by enclosing the character within "[." and ".]". For example, to match a "]" within a list, the collating sequence "[.].]" could be used:

Function call

```
jcs:regex( "[[[:digit:][.].]]*", "[10]" )
```

Returned node-set

```
<match> "[10]"
```

In addition, the following collating sequences are defined:

Sequence	ASCII	Sequence	ASCII	Sequence	ASCII
[.SOH.]	0x01	[.SUB.]	0x1A	[.three.]	0x33 “3”
[.STX.]	0x02	[.ESC.]	0x1B	[.four.]	0x34 “4”
[.ETX.]	0x03	[.IS4.] [.FS.]	0x1C	[.five.]	0x35 “5”
[.EOT.]	0x04	[.IS3.] [.GS.]	0x1D	[.six.]	0x36 “6”
[.ENQ.]	0x05	[.IS2.] [.RS.]	0x1E	[.seven.]	0x37 “7”
[.ACK.]	0x06	[.IS1.] [.US.]	0x1F	[.eight.]	0x38 “8”
[.BEL.] [.alert.]	0x07	[.space.]	0x20 “ “	[.nine.]	0x39 “9”
[.BS.] [.backspace.]	0x08	[.exclamation-mark.]	0x21 “!”	[.colon.]	0x3A “:”
[.HT.] [.tab.]	0x09	[.quotation-mark.]	0x22 “	[.semicolon.]	0x3B “;”
[.LF.] [.newline.]	0x0A	[.number-sign.]	0x23 “#”	[.less-than-sign.]	0x3C “<”
[.VT.] [.vertical-tab.]	0x0B	[.dollar-sign.]	0x24 “$”	[.equals-sign.]	0x3D “=”
[.FF.] [.form-feed.]	0x0C	[.percent-sign.]	0x25 “%”	[.greater-than-sign.]	0x3E “>”
[.CR.] [.carriage-return.]	0x0D	[.ampersand.]	0x26 “&”	[.question-mark.]	0x3F “?”
[.SO.]	0x0E	[.apostrophe.]	0x27 ‘	[.commercial-at.]	0x40 “@”
[.SI.]	0x0F	[.left-parenthesis.]	0x28 “(“	[.left-square-bracket.]	0x5B “[“
[.DLE.]	0x10	[.right-parenthesis.]	0x29 “)”	[.backslash.] [.reverse-solidus.]	0x5C “\”
[.DC1.]	0x11	[.asterisk.]	0x2A “*”	[.right-square-bracket.]	0x5D “]”
[.DC2.]	0x12	[.plus-sign.]	0x2B “+”	[.circumflex.] [.circumflex-accent.]	0x5E “^”
[.DC3.]	0x13	[.comma.]	0x2C “,”	[.underscore.] [.low-line.]	0x5F “_”
[.DC4.]	0x14	[.hyphen.] [.hyphen-minus.]	0x2D “-”	[.grave-accent.]	0x60 “`”
[.NAK.]	0x15	[.period.] [.full-stop.]	0x2E “.”	[.left-brace.] [.left-curly-bracket.]	0x7B “{“

[.SYN.]	0x16	[.slash.] [.solidus.]	0x2F "/"	[.vertical-line.]	0x7C "|"
[.ETB.]	0x17	[.zero.]	0x30 "0"	[.right-brace.] [.right-curly-bracket.]	0x7D "}"
[.CAN.]	0x18	[.one.]	0x31 "1"	[.tilde.]	0x7E "~"
[.EM.]	0x19	[.two.]	0x32 "2"	[.DEL.]	0x7F

Function call

```
jcs:regex( "[[:digit:][.period.]]*", "10.4R1.9" )
```

Returned node-set

```
<match> "10.4"
```

There are no defined character equivalence classes so enclosing collating elements in "[=" and "=]" has the same effect as enclosing them in "[." and ".]".

The node-set returned by jcs:regex() consists of <match> element nodes with the appropriate match assigned as the text content. If no match is found then an empty node-set is returned. If there is an error with the regular expression then an error message is displayed and an empty node-set is returned. The order of the nodes within the returned node-set is deterministic, so the node-set can be treated similar to an array in other programming languages, meaning that the nodes can be retrieved based on their numerical order. Node number 1 is always the match of the entire regular expression, and nodes 2 through 9 contain subexpression matches, if appropriate. The subexpression nodes occur within the node-set in the same order as they appear in the regular expression pattern, but they are only included if they have a match, or if they do not have a match but a latter subexpression has a match, in which case they are included with an empty string as their text contents. Examples of returned node-sets follow.

Function call

```
jcs:regex( "123*", "other" )
```

Returned node-set

```
Empty
```

Function call

```
jcs:regex( "[A-Z]*", "JUNOS 10.4R1.9" )
```

Returned node-set

```
<match> "JUNOS"
```

Function call

```
jcs:regex( "([0-9]{4})-([0-9]{2})-([0-9]{2})", "2010-12-04" )
```

Returned node-set

```
<match> "2010-12-04"
<match> "2010"
<match> "12"
<match> "04"
```

Function call

```
jcs:regex( “(1?)(2?)(3?)(4?)(5)(6)(7)(8)(9)”, “56789” )
```

Returned node-set

```
<match> “56789”
<match> ““
<match> ““
<match> ““
<match> “5”
<match> “6”
<match> “7”
<match> “8”
```

Regular expression features not discussed above should be considered unsupported, in particular the following are either not supported or do not work at the time of this writing: backreferences, shorthand character classes, buffer operators, look-around, non-capturing groups, and non-greedy (lazy) repetition.

Example

This op script demonstrates how to use jcs:regex() to extract specific substrings from within a larger string.

Code

```
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> {

       /* parse the $localtime-iso parameter */
       var $regex =
       “([[:digit:]]*)-0?([[:digit:]]*)-0?([[:digit:]]*) 0?([0-9]*):0?([0-9]*):0?([0-9]*).*”;
       var $result = jcs:regex($regex, $localtime-iso );

       /* Display the complete match */
       <output> “Time: “ _ $result[1];

       /* Display all the captured subexpressions */
       <output> “Year: “ _ $result[2];
       <output> “Month: “ _ $result[3];
       <output> “Day: “ _ $result[4];
       <output> “Hour: “ _ $result[5];
       <output> “Minute: “ _ $result[6];
       <output> “Second: “ _ $result[7];
   }
}
```

Output

```
Time: 2011-02-23 16:14:06 UTC
Year: 2011
Month: 2
Day: 23
Hour: 16
Minute: 14
Second: 6
```

What to Do Next & Where to Go

http://www.juniper.net/dayone

Get all the *Day One* books and new *This Week* titles, too. All from Juniper Networks Books. Check for new automation books as they get published.

http://www.juniper.net/automation

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

http://forums.juniper.net/jnet

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

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

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

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

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