

O'REILLY®

Free Sampler



# Learning Puppet 4

A GUIDE TO CONFIGURATION MANAGEMENT AND AUTOMATION

Jo Rhett

# Learning Puppet 4

If you're a system administrator, developer, or site reliability engineer responsible for handling hundreds or even thousands of nodes in your network, the Puppet configuration management tool will make your job a whole lot easier. This practical guide shows you what Puppet does, how it works, and how it can provide significant value to your organization.

Through hands-on tutorials, DevOps engineer Jo Rhett demonstrates how Puppet manages complex and distributed components to ensure service availability. You'll learn how to secure configuration consistency across servers, clients, your router, and even that computer in your pocket by setting up your own testing environment.

- Learn exactly what Puppet is, why it was created, and what problems it solves
- Tailor Puppet to your infrastructure with environment layouts that meet your specific needs
- Write declarative Puppet policies to produce consistency in your systems
- Build, test, and publish your own Puppet modules
- Manage network devices such as routers and switches with *puppet device* and integrated Puppet agents
- Scale Puppet servers for high availability and performance
- Explore web dashboards and orchestration tools that supplement and complement Puppet

**Jo Rhett** is a DevOps engineer with 20 years of experience conceptualizing and delivering large-scale Internet services. He focuses on creating automation and infrastructure to accelerate deployment and minimize outages.

SYSTEM ADMINISTRATION

US \$49.99

CAN \$57.99

ISBN: 978-1-491-90766-5



Twitter: @oreillymedia  
facebook.com/oreilly

---

# Learning Puppet 4

*A Guide to Configuration Management  
and Automation*

*Jo Rhett*

Beijing • Boston • Farnham • Sebastopol • Tokyo

**O'REILLY**®

## Learning Puppet 4

by Jo Rhett

Copyright © 2016 Jo Rhett. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editor:** Brian Anderson

**Production Editor:** Kristen Brown

**Copyeditor:** Rachel Monaghan

**Proofreader:** Jasmine Kwityn

**Indexer:** Judy McConville

**Interior Designer:** David Futato

**Cover Designer:** Karen Montgomery

**Illustrator:** Rebecca Demarest

April 2016: First Edition

### Revision History for the First Edition

2016-03-22: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491907665> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Learning Puppet 4*, the cover image of an European polecat, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-90766-5

[LSI]

---

# Table of Contents

Foreword.....	xxi
Preface.....	xxiii
Introduction.....	xxxix

---

## Part I. Controlling with Puppet Apply

<b>1. Thinking Declarative.....</b>	<b>3</b>
Handling Change	3
Using Idempotence	4
Declaring Final State	5
Reviewing Declarative Programming	6
<b>2. Creating a Learning Environment.....</b>	<b>7</b>
Installing Vagrant	8
Installing Vagrant on Mac	8
Installing Git Tools on Windows	10
Installing VirtualBox on Windows	11
Installing Vagrant on Windows	14
Starting a Bash Shell	17
Downloading a Box	18
Cloning the Learning Repository	19
Install the Vagrant vbguest Plugin	19
Initializing the Vagrant Setup	19
Verifying the /vagrant Filesystem	21
Initializing Non-Vagrant Systems	22

Installing Some Helpful Utilities	22
Choosing a Text Editor	22
On the Virtual System	23
On Your Desktop	24
In Your Profile	24
Reviewing the Learning Environment	25
<b>3. Installing Puppet.....</b>	<b>27</b>
Adding the Package Repository	27
What Is a Puppet Collection?	28
Installing the Puppet Agent	28
Reviewing Dependencies	29
Reviewing Puppet 4 Changes	30
Linux and Unix	30
Windows	32
Making Tests Convenient	32
Running Puppet Without sudo	33
Running Puppet with sudo	34
Reviewing Puppet Installation	35
<b>4. Writing Manifests.....</b>	<b>37</b>
Implementing Resources	37
Applying a Manifest	38
Declaring Resources	39
Viewing Resources	40
Executing Programs	41
Was That Idempotent?	42
Managing Files	43
Finding File Backups	44
Restoring Files	45
Avoiding Imperative Manifests	45
Testing Yourself	47
Reviewing Writing Manifests	47
<b>5. Using the Puppet Configuration Language.....</b>	<b>49</b>
Defining Variables	50
Defining Numbers	51
Creating Arrays and Hashes	52
Mapping Hash Keys and Values	52
Using Variables in Strings	53
Using Braces to Limit Problems	53
Preventing Interpolation	54

Using Unicode Characters	55
Avoiding Redefinition	55
Avoiding Reserved Words	56
Learning More	57
Finding Facts	57
Calling Functions in Manifests	59
Using Variables in Resources	60
Defining Attributes with a Hash	62
Declaring Multiple Resource Titles	63
Declaring Multiple Resource Bodies	63
Modifying with Operators	64
Adding to Arrays and Hashes	65
Removing from Arrays and Hashes	65
Order of Operations	66
Using Comparison Operators	66
Evaluating Conditional Expressions	68
Matching Regular Expressions	70
Building Lambda Blocks	71
Looping Through Iterations	72
each()	73
filter()	75
map()	76
reduce()	76
slice()	78
with()	79
Capturing Extra Parameters	79
Iteration Wrap-Up	80
Reviewing Puppet Configuration Language	80
<b>6. Controlling Resource Processing.....</b>	<b>81</b>
Adding Aliases	81
Specifying an Alias by Title	82
Adding an Alias Metaparameter	82
Preventing Action	82
Auditing Changes	83
Defining Log Level	84
Filtering with Tags	84
Skipping Tags	85
Limiting to a Schedule	86
Utilizing periodmatch	87
Avoiding Dependency Failures	90
Declaring Resource Defaults	91

Reviewing Resource Processing	91
<b>7. Expressing Relationships.....</b>	<b>93</b>
Managing Dependencies	93
Referring to Resources	94
Ordering Resources	95
Assuming Implicit Dependencies	95
Triggering Refresh Events	96
Chaining Resources with Arrows	97
Processing with Collectors	98
Understanding Puppet Ordering	99
Debugging Dependency Cycles	100
Avoiding the Root User Trap	101
Utilizing Stages	103
Reviewing Resource Relationships	103
<b>8. Upgrading Puppet 3 Manifests.....</b>	<b>105</b>
Replacing Deprecated Features	105
Junking the Ruby DSL	105
Upgrading Config Environments	106
Removing Node Inheritance	107
Disabling puppet kick	107
Qualifying Relative Class Names	107
Losing the Search Function	108
Replacing Import	108
Documenting Modules with Puppet Strings	108
Installing the Tagmail Report Processor	109
Querying PuppetDB	109
Preparing for the Upgrade	109
Validating Variable Names	109
Quoting Strings	110
Preventing Numeric Assignment	110
Testing Boolean Facts	112
Qualifying Defined Types	112
Adding Declarative Permissions	113
Removing Cron Purge	114
Replacing MSI Package Provider	114
Adjusting Networking Facts	114
Testing with the Future Parser	115
Using Directory Environments	115
Duplicating a Master or Node	116
Enhancing Older Manifests	116



Adding else to unless	116
Calling Functions in Strings	116
Matching String Regexps	117
Letting Expressions Stand Alone	117
Chaining Assignments	118
Chaining Expressions with a Semicolon	118
Using Hash and Array Literals	118
Configuring Error Reporting	119
<b>9. Wrap-Up of Puppet Basics.....</b>	<b>121</b>
Best Practices for Writing Manifests	122
Learning More About Puppet Manifests	122

---

## Part II. Creating Puppet Modules

<b>10. Creating a Test Environment.....</b>	<b>125</b>
Verifying the Production Environment	125
Creating the Test Environment	125
Changing the Base Module Path	126
Skipping Ahead	126
<b>11. Separating Data from Code.....</b>	<b>127</b>
Introducing Hieradata	127
Creating Hieradata Backends	128
Hieradata in YAML	128
Hieradata in JSON	129
Puppet Variable and Function Lookup	130
Configuring Hieradata	130
Backends	131
Backend Configuration	131
Logger	131
Hierarchy	132
Merge Strategy	133
Complete Example	134
Looking Up Hieradata	134
Checking Hieradata Values from the Command Line	135
Performing Hieradata Lookups in a Manifest	136
Testing Merge Strategy	136
Providing Global Data	139

<b>12. Using Modules.....</b>	<b>141</b>
Finding Modules	141
Puppet Forge	141
Public GitHub Repositories	142
Internal Repositories	143
Evaluating Module Quality	143
Puppet Supported	144
Puppet Approved	144
Quality Score	145
Community Rating	148
Installing Modules	149
Installing from a Puppet Forge	149
Installing from GitHub	150
Testing a Single Module	151
Defining Config with Hiera	152
Assigning Modules to Nodes	153
Using Hiera for Module Assignment	153
Assigning Classes to Every Node	154
Altering the Class List per Node	155
Avoiding Node Assignments in Manifests	155
Upgrading from Puppet 2 or 3	157
Examining a Module	158
Reviewing Modules	159
<b>13. Designing a Custom Module.....</b>	<b>161</b>
Choosing a Module Name	161
Avoiding Reserved Names	162
Generating a Module Skeleton	162
Modifying the Default Skeleton	163
Understanding Module Structure	164
Installing the Module	164
Creating a Class Manifest	164
What Is a Class?	165
Declaring Class Resources	165
Accepting Input	166
Sharing Files	168
Testing File Synchronization	169
Synchronizing Directories	170
Parsing Templates	171
Common Syntax	171
Using Puppet EPP Templates	172
Using Ruby ERB Templates	175

Creating Readable Templates	177
Testing the Module	177
Peeking Beneath the Hood	178
Best Practices for Module Design	179
Reviewing Custom Modules	179
<b>14. Improving the Module.....</b>	<b>181</b>
Validating Input with Data Types	181
Valid Types	182
Validating Values	184
Testing Values	186
Comparing Strings with Regular Expressions	187
Matching a Regular Expression	188
Revising the Module	188
Looking Up Input from Hiera	189
Naming Parameters Keys Correctly	189
Using Array and Hash Merges	190
Understanding Lookup Merge	191
Specifying Merge Strategy in Data	192
Replacing Direct Hiera Calls	192
Building Subclasses	194
Creating New Resource Types	195
Understanding Variable Scope	197
Using Out-of-Scope Variables	197
Understanding Top Scope	198
Understanding Node Scope	199
Understanding Parent Scope	199
Tracking Resource Defaults Scope	199
Avoiding Resource Default Bleed	200
Redefining Variables	201
Calling Other Modules	202
Sourcing a Common Dependency	202
Using a Different Module	204
Ordering Dependencies	205
Depending on Entire Classes	205
Placing Dependencies Within Optional Classes	206
Notifying Dependencies from Dynamic Resources	207
Solving Unknown Resource Dependencies	208
Containing Classes	210
Creating Reusable Modules	211
Avoiding Fixed Values in Attribute Values	211
Ensuring Fixed Values for Resource Names	212

Defining Defaults in a Params Manifest	213
Best Practices for Module Improvements	214
Reviewing Module Improvements	215
<b>15. Extending Modules with Plugins.....</b>	<b>217</b>
Adding Custom Facts	217
External Facts	218
Custom (Ruby) Facts	220
Debugging	224
Understanding Implementation Issues	225
Defining Functions	225
Puppet Functions	226
Ruby Functions	227
Using Custom Functions	230
Creating Puppet Types	230
Defining Ensurable	231
Accepting Params and Properties	231
Validating Input Values	232
Defining Implicit Dependencies	234
Learning More About Puppet Types	234
Adding New Providers	235
Determining Provider Suitability	235
Assigning a Default Provider	236
Defining Commands for Use	237
Ensure the Resource State	237
Adjusting Properties	238
Providing a List of Instances	239
Taking Advantage of Caching	240
Learning More About Puppet Providers	241
Identifying New Features	241
Binding Data Providers in Modules	242
Using Data from a Function	242
Using Data from Hiera	243
Performing Lookup Queries	244
Requirements for Module Plugins	244
Reviewing Module Plugins	245
<b>16. Documenting Modules.....</b>	<b>247</b>
Learning Markdown	247
Writing a Good README	248
Documenting the Classes and Types	248
Installing YARD and Puppet Strings	248

Fixing the Headers	249
Listing Parameters	250
Documenting Variable References	251
Showing Examples	251
Listing Authors and Copyright	252
Documenting Functions	252
Generating Documentation	253
Updating Module Metadata	254
Identifying the License	254
Promoting the Project	255
Indicating Compatibility	255
Defining Requirements	256
Listing Dependencies	257
Identifying a Module Data Source	257
Updating Old Metadata	258
Maintaining the Change Log	258
Evolving and Improving	258
Best Practices for Documenting Modules	259
<b>17. Testing Modules.....</b>	<b>261</b>
Installing Dependencies	261
Installing Ruby	261
Adding Beaker	262
Bundling Dependencies	262
Preparing Your Module	263
Defining Fixtures	263
Defining RSpec Unit Tests	264
Defining the Main Class	264
Passing Valid Parameters	265
Failing Invalid Parameters	266
Testing File Creation	267
Validating Class Inclusion	268
Using Facts in Tests	268
Using Hiera Input	269
Defining Parent Class Parameters	270
Testing Functions	270
Adding an Agent Class	271
Testing Other Types	271
Creating Acceptance Tests	272
Installing Ruby for System Tests	272
Defining the Nodeset	272
Configuring the Test Environment	273

Creating an Acceptance Test	274
Running Acceptance Tests	275
Using Skeletons with Testing Features	278
Finding Documentation	280
Reviewing Testing Modules	280
<b>18. Publishing Modules.....</b>	<b>283</b>
Updating the Module Metadata	283
Packaging a Module	284
Uploading a Module to the Puppet Forge	284
Publishing a Module on GitHub	285
Automating Module Publishing	287
Getting Approved Status from Puppet Labs	287

---

## Part III. Using a Puppet Server

<b>19. Preparing for a Puppet Server.....</b>	<b>291</b>
Understanding the Catalog Builder	291
Node	291
Agent	292
Server	293
Planning for Puppet Server	294
The Server Is Not the Node	294
The Node Is Not the Server	295
Store Server Data Files Separately	296
Functions Run on the Server	297
Choosing Puppet Master Versus Puppet Server	297
Upgrading Easily with Puppet Master	298
Embracing the Future with Puppet Server	298
Why There's Really No Choice	299
Ensuring a High-Performance Server	299
<b>20. Creating a Puppet Master.....</b>	<b>301</b>
Starting the puppetmaster VM	301
Installing the Puppet Master	302
Configuring a Firewall for the Puppet Master	303
Running the WEBrick Server	303
Testing with the Puppet Master Service	304
Scaling the Puppet Master with Passenger	305
Installing Apache	306
Installing Phusion Passenger	306

Configuring the Puppet Master	307
IPv6 Dual-Stack Puppet Master	309
Debugging Puppet Master	309
<b>21. Creating a Puppet Server.....</b>	<b>311</b>
Starting the puppetserver VM	311
Installing Puppet Server	312
Configuring a Firewall for Puppet Server	312
Configuring Puppet Server	313
Defining Server Paths	314
Limiting Memory Usage	315
Configuring TLS Certificates	315
Avoiding Obsolete Settings	317
Configuring Server Logs	318
Configuring Server Authentication	319
Running Puppet Server	321
Adding Ruby Gems	321
IPv6 Dual-Stack Puppet Server	322
<b>22. Connecting a Node.....</b>	<b>323</b>
Creating a Key Pair	323
Authorizing the Node	324
Downloading the First Catalog	324
Installing Hieradata and Modules	325
Testing with a Client Node	326
Learning More About Puppet Server	326
<b>23. Migrating an Existing Puppet Master.....</b>	<b>329</b>
Migrating the Puppet Master Config	329
Synchronizing All Environments	330
Copying Hieradata	331
Moving the MCollective Config Directory	332
Removing Node Inheritance	332
Testing a Client Node	333
Upgrading Clients	334
<b>24. Utilizing Advantages of a Puppet Server.....</b>	<b>335</b>
Using Server Data in Your Manifests	335
Trusted Facts	335
Server Facts	336
Server Configuration Settings	337
Backing Up Files Changed on Nodes	337

Processing Puppet Node Reports	338
Enabling Transmission of Reports	339
Running Audit Inspections	339
Storing Node Reports	340
Logging Node Reports	341
Transmitting Node Reports via HTTP	342
Transmitting Node Reports to PuppetDB	342
Emailing Node Reports	342
Creating a Custom Report Processor	344
<b>25. Managing TLS Certificates.....</b>	<b>347</b>
Reviewing Node Authentication	347
Autosigning Agent Certificates	348
Name-Based Autosigning	348
Policy-Based Autosigning	349
Naive Autosigning	353
Using an External Certificate Authority	353
Distributing Certificates Manually	354
Installing Certificates on the Server	355
Disabling CA on a Puppet Server	355
Disabling CA on a Puppet Master	355
Using Different CAs for Servers and Agents	356
Distributing the CA Revocation List	357
Learning More About TLS Authentication	357
<b>26. Growing Your Puppet Deployment.....</b>	<b>359</b>
Using a Node Terminus	359
Running an External Node Classifier	360
Querying LDAP	361
Starting with Community Examples	363
Deploying Puppet Servers at Scale	364
Keeping Distinct Domains	364
Sharing a Single Puppet CA	364
Using a Load Balancer	365
Managing Geographically Dispersed Servers	366
Managing Geographically Dispersed Nodes	367
Falling Back to Cached Catalogs	368
Making the Right Choice	368
Best Practices for Puppet Servers	369
Reviewing Puppet Servers	369



---

## Part IV. Integrating Puppet

<b>27. Tracking Puppet Status with Dashboards.....</b>	<b>373</b>
Using Puppet Dashboard	373
Installing Dashboard Dependencies	374
Enabling Puppet Dashboard	381
Viewing node status	389
Using Dashboard as a Node Classifier	394
Implementing Dashboard in Production	401
Evaluating Alternative Dashboards	405
Puppetboard	405
Puppet Explorer	406
PanoPuppet	408
ENC Dashboard	409
Foreman	410
Upgrading to the Enterprise Console	412
Viewing Status	412
Classifying Nodes	413
Inspecting Events	413
Tracking Changes	414
Controlling Access	415
Evaluating Puppet Enterprise	416
Finding Plugins and Tools	416
<b>28. Running the Puppet Agent on Windows.....</b>	<b>417</b>
Creating a Windows Virtual Machine	417
Creating a VirtualBox Windows VM	418
Adding an Internal Network Adapter	418
Connecting the Windows Installation Media	419
Configuring the Internal Network Adapter	420
Installing Puppet on Windows	421
Configuring Puppet on Windows	422
Running Puppet Interactively	423
Starting the Puppet Service	424
Debugging Puppet Problems	425
Writing Manifests for Windows	425
Finding Windows-Specific Modules	426
Concluding Thoughts on Puppet Windows	427
<b>29. Customizing Environments.....</b>	<b>429</b>
Understanding Environment Isolation	429
Enabling Directory Environments	430

Assigning Environments to Nodes	431
Configuring an Environment	431
Choosing a Manifest Path	433
Utilizing Hiera Hierarchies	433
Binding Data Providers in Environments	434
Querying Data from a Function	434
Querying Data from Hiera	435
Strategizing How to Use Environments	436
Promoting Change Through Layers	436
Solving One-Off Problems Using Environments	437
Supporting Diverse Teams with Environments	438
Managing Environments with r10k	439
Listing Modules in the Puppetfile	439
Creating a Control Repository	440
Configuring r10k Sources	442
Adding New Environments	442
Populating a New Installation	443
Updating a Single Environment	444
Replicating Hiera Data	444
Invalidating the Environment Cache	445
Restarting JRuby When Updating Plugins	447
Reviewing Environments	448
<b>30. Controlling Puppet with MCollective.....</b>	<b>449</b>
Configuring MCollective	449
Enabling the Puppet Labs Repository	450
Installing the MCollective Module	450
Generating Passwords	450
Configuring Hiera for MCollective	451
Enabling the Middleware	452
Connecting MCollective Servers	453
Validating the Installation	454
Creating Another Client	455
Installing MCollective Agents and Clients	456
Sharing Facts with Puppet	457
Pulling the Puppet Strings	458
Viewing Node Inventory	458
Checking Puppet Status	460
Disabling the Puppet Agent	461
Invoking Ad Hoc Puppet Runs	461
Limiting Targets with Filters	464
Providing a List of Targets	466

Limiting Concurrency	466
Manipulating Puppet Resource Types	467
Comparing to Puppet Application Orchestration	469
Learning More About MCollective	470
<b>31. Managing Network Infrastructure with Puppet</b> .....	<b>471</b>
Managing Network Devices with Puppet Device	472
Enabling SSH on the Switch	472
Configuring the Puppet Proxy Agent	473
Installing the Device_Hiera Module	475
Defining Resource Defaults in Hier	475
Centralizing VLAN Configuration	477
Applying Default Configs to Interfaces	477
Customizing Interface Configurations	478
Testing Out the Switch Configuration	479
Adding Resource Types and Providers	479
Merging Defaults with Other Resources	480
Using the NetDev Standard Library	481
Finding NetDev Vendor Extensions	481
Creating a NetDev Device Object	482
Reducing Duplication with Device_Hiera	482
Puppetizing Cisco Nexus Switches	483
Configuring the Puppet Server	483
Preparing the NX-OS Device	484
Installing the NX-OS Puppet Agent	485
Enabling the NX-OS Puppet Agent	485
Managing Configuration	486
Puppetizing Juniper Devices	486
Supported Devices	487
Installing Modules on the Puppet Server	488
Preparing the Junos Device	489
Installing the Junos Puppet Agent	489
Creating the Puppet User	490
Adjusting Physical Interface Settings	491
Simplifying Layer-2 VLANs	492
Enabling Link Aggregation	493
Defining Ad Hoc Configuration Parameters	494
Distributing Junos Event Scripts	495
Running Puppet Automatically	496
Troubleshooting	496
Best Practices for Network Devices	497
Reviewing Network Devices	497

<b>32. Assimilating Puppet Best Practices.....</b>	<b>499</b>
Managing Change	499
Expecting Change	499
Controlling Rate of Change	500
Tracking Change	500
Choosing Puppet Apply Versus Puppet Server	501
Benefits of Puppet Apply	502
Benefits of Puppet Server	503
Benefits Shared	504
Summarizing the Differences	505
Creating a Private Puppet Forge	506
Pulp	506
Puppet Forge Server	506
Django Forge	506
Good Practices	507
Indenting Heredoc	507
Splaying Puppet Agent Cron Jobs	507
Cleaning Puppet Reports	508
Trimming the File Bucket	509
Drinking the Magic Monkey Juice	509
Hating on Params.pp	510
Disabling Environments	511
Tracking Providers	511
Breaking the Rules	512
Working Good, Fast, Cheap	513
Choosing Fight or Flight	513
Letting the Strings Pull You	513
Leveraging Puppet for Small Changes	513
Tossing Declarative to the Wind	514
Allowing Anyone to sudo puppet	515
<b>33. Finding Support Resources.....</b>	<b>517</b>
Accessing Community Support	517
Engaging Puppet Labs Support	518
Contacting the Author	518
<b>Afterword.....</b>	<b>521</b>
<b>A. Installing Puppet on Other Platforms.....</b>	<b>523</b>
<b>B. Configuring Firewalls on Other Platforms.....</b>	<b>525</b>

<b>C. Installing Ruby</b> .....	<b>527</b>
<b>Index</b> .....	<b>531</b>

---

# Thinking Declarative

If you have experience writing shell, Ruby, Python, or Perl scripts that make changes to a system, you've very likely been performing *imperative programming*. Imperative programming issues commands that change a target's state, much as the imperative grammatical mood in natural language expresses commands for people to act on.

You may be using *procedural programming* standards, where state changes are handled within procedures or subroutines to avoid duplication. This is a step toward declarative programming, but the main program still tends to define each operation, each procedure to be executed, and the order in which to execute them in an imperative manner.

While it can be useful to have a background in procedural programming, a common mistake is to attempt to use Puppet to make changes in an imperative fashion. The very best thing you can do is forget everything you know about imperative or procedural programming.

If you are new to programming, don't feel intimidated. People without a background in imperative or procedural programming can often learn good Puppet practices faster.

Writing good Puppet manifests requires declarative programming. When it comes to maintaining configuration on systems, you'll find declarative programming to be easier to create, easier to read, and easier to maintain. Let's show you why.

## Handling Change

The reason that you need to cast aside imperative programming is to handle change better.

When you write code that performs a sequence of operations, that sequence will make the desired change the first time it is run. If you run the same code the second time in a row, the same operations will either fail or create a different state than desired. Here's an example:

```
$ sudo useradd -u 1001 -g 1001 -c "Joe User" -m joe
$ sudo useradd -u 1001 -g 1000 -c "Joe User" -m joe
useradd: user 'joe' already exists
```

So then you need to change the code to handle that situation:

```
# bash excerpt
getent passwd $USERNAME > /dev/null 2> /dev/null
if [ $? -ne 0 ]; then
    useradd -u $UID -g $GID -c "$COMMENT" -s $SHELL -m $USERNAME
else
    usermod -u $UID -g $GID -c "$COMMENT" -s $SHELL -m $USERNAME
fi
```

OK, that's six lines of code and all we've done is ensure that the username isn't already in use. What if we need to check to ensure the UID is unique, the GID is valid, and that the password expiration is set? You can see that this will be a very long script even before we adjust it to ensure it works properly on multiple operating systems.

This is why we say that imperative programming doesn't handle change very well. It takes a lot of code to cover every situation you need to test.

## Using Idempotence

When managing computer systems, you want the operations applied to be *idempotent*, where the operation achieves the same results every time it executes. Idempotence allows you to apply and reapply (or converge) a configuration manifest and always achieve the desired state.

In order for imperative code to be idempotent, it needs to have instructions for how to compare, evaluate, and apply not just every resource, but also each attribute of the resource. As you saw in the previous section, even the simplest of operations will quickly become ponderous and difficult to maintain.

## What is an Idempotent Operation?

In mathematics and computer science, idempotent operations are those that can be applied multiple times without changing the result beyond the initial application. The word literally means “[the quality of] having the same power,” from the Latin roots *idem* + *potent* “same” + “power.”<sup>1</sup> Here are some examples of idempotent and non-idempotent math and code:

<code>any number^1</code>	<b>Idempotent</b>	A number to the power of 1 is the same
<code>value = value * 2</code>	<b>Non-idempotent</b>	Will double every time
<code>value = value * 2 / 2</code>	<b>Idempotent</b>	Remains the same value
<code>echo "Good!" &gt;&gt; /some/file</code>	<b>Non-idempotent</b>	File will keep growing
<code>echo "Good!" &gt; /some/file</code>	<b>Idempotent</b>	File will always have the same content

The simplistic final example avoids having to compare the state of the item by simply overwriting it every time. This only works in a limited set of situations. Most changes require evaluation to determine what changes are necessary.

## Declaring Final State

As we mentioned in **Introduction**, for a configuration state to be achieved no matter the conditions, the configuration language must avoid describing the actions required to reach the desired state. Instead, the configuration language should describe the desired state itself, and leave the actions up to the interpreter. Language that declares the final state is called *declarative*.

Rather than writing extensive imperative code to handle every situation, it is much simpler to declare what you want the final state to be. In other words, instead of including dozens of lines of comparison, the code reflects only the desired final state of the resource (a user account, in this example). Here we will introduce you to your first bit of Puppet configuration language, a resource declaration for the same user we created earlier:

---

<sup>1</sup> First seen in George Boole’s book *The Mathematical Analysis of Logic*, originally published in 1847.



```
user { 'joe':  
  ensure => present,  
  uid    => '1001',  
  gid    => '1000',  
  comment => 'Joe User',  
  managehome => true,  
}
```

As you can see, the code is not much more than a simple text explanation of the desired state. A user named *Joe User* should be *present*, a home directory for the user should be created, and so on. It is very clear, very easy to read. Exactly how the user should be created is not within the code, nor are instructions for handling different operating systems.

Declarative language is much easier to read, and less prone to breakage due to environment differences. Puppet was designed to achieve consistent and repeatable results. You describe what the final state of the resource should be, and Puppet will evaluate the resource and apply any necessary changes to reach that state.

## Reviewing Declarative Programming

Conventional programming languages create change by listing exact operations that should be performed. Code that defines each state change and the order of changes is known as *imperative programming*.

Good Puppet manifests are written with declarative programming. Instead of defining exactly how to make changes, in which you must write code to test and compare the system state before making that change, you instead declare how it should be. It is up to the Puppet agent to evaluate the current state and apply the necessary changes.

As this chapter has demonstrated, declarative programming is easier to create, easier to read, and easier to maintain.