In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 ‘Puppet Language and Style’
- A synopsis of the book’s content
- More information on Puppet Cookbook Third Edition

About the Authors

**Thomas Uphill** is an RHCA who has been using Puppet since 0.24. He has worked as a system administrator for almost 20 years, most recently with RHEL systems. He recently wrote Mastering Puppet, Packt Publishing a book for managing Puppet in large deployments. He has given tutorials on Puppet at LISA and LOPSA-East. When not at the Puppet User Group of Seattle (PUGS), he can be found at http://ramblings.narrabilis.com.

Thank you to John Arundel for the previous editions of this book, I had a great foundation for this rewrite. I would like to thank my wife Priya Fernandes for her support and encouragement while I was updating this book. Thanks to my fellow PUGS Andy and Justin for their suggestions. Thanks to my reviewers, Jeroen Hooyberghs, James Fryman, and Dhruv Ahuja for taking the time to find all the errors they did.

**John Arundel** has worked in the IT industry for most of his life, and during that time has done wrong (or seen others do wrong) almost everything that you can do wrong with computers. That comprehensive knowledge of what not to do, he feels, is one of his greatest assets as a consultant. He is still adding to it.

He spent much of his career working in very large corporations and, as a result, now likes to work with very small corporations. They like working with him too, not only because he can tell them about things that should not be done, but also because he can confidently inform them that big companies don't know what they're doing either.

Off the clock, he enjoys gardening, competitive rifle shooting, and other gentle hobbies. You can follow him on Twitter @bitfield. If your company is small enough, you can hire him there too.
Puppet Cookbook

Third Edition

Configuration management has become a requirement for system administrators. Knowing how to use configuration management tools, such as Puppet, enables administrators to take full advantage of automated provisioning systems and cloud resources. There is a natural progression from performing a task, scripting a task to creating a module in Puppet, or Puppetizing a task.

This book takes you beyond the basics and explores the full power of Puppet, showing you in detail how to tackle a variety of real-world problems and applications. At every step, it shows you exactly what commands you need to type and includes complete code samples for every recipe. It takes you from a rudimentary knowledge of Puppet to a more complete and expert understanding of Puppet's latest and most advanced features, community best practices, scaling, and performance. This edition of the book includes recipes for configuring and using Hiera, puppetdb and operating a centralized puppetmaster configuration.

This book also includes real examples from production systems and techniques that are in use in some of the world's largest Puppet installations. It will show you different ways to do things using Puppet, and point out some of the pros and cons of these approaches.

The book is structured so that you can dip in at any point and try out a recipe without having to work your way through from cover to cover. Whatever your level of Puppet experience, there's something for you—from simple workflow tips to advanced, high-performance Puppet architectures.

Puppet is an ever-changing ecosystem of tools. I've tried to include all the tools that I feel are important today, such as r10k. The #puppet IRC channel, puppetlabs blog (http://puppetlabs.com/blog), and the Forge (http://forge.puppetlabs.com) are great resources to stay up to date with the changes being made to Puppet.
What This Book Covers

Chapter 1, *Puppet Language and Style*, introduces the Puppet language and shows how to write manifests. The Puppet linting tool, puppet-lint, is introduced and we review best practices to write Puppet code. Metaparameters are shown with examples. We also preview proposed changes to the Puppet language by using the future parser.

Chapter 2, *Puppet Infrastructure*, is all about how to deploy Puppet in your environment. We cover the two main methods of installation, centralized and decentralized (masterless). We show you how to use Git to centrally manage your code. We also configure puppetdb and Hiera.

Chapter 3, *Writing Better Manifests*, deals with organizing your Puppet manifests. Manifests are used to build modules; we introduce the concept of roles and profiles to abstract how modules are applied to machines. Parameterized classes are introduced. We also show you how to efficiently define resources with arrays of resources and resource defaults.

Chapter 4, *Working with Files and Packages*, shows you how to manage files using snippets (fragments). We introduce the power of templating with both Ruby (ERB) and Puppet (EPP) templates. We also explore ways to secure information stored in your Puppet manifests.

Chapter 5, *Users and Virtual Resources*, deals with the advanced topic of virtual and exported resources. Virtual resources are a way of defining resources but not applying them by default. Exported resources are similar but are used to have resources from one machine, applied to one or more other machines.

Chapter 6, *Managing Resources and Files*, is about dealing with directories and purging resources not controlled by Puppet. We show you how to have file resources applied differently on different machines. Methods for managing host entries in /etc/hosts are shown with exported resources examples.

Chapter 7, *Managing Applications*, shows you how to use Puppet to manage your deployed applications. Using public Forge modules, we configure Apache, nginx, and MySQL.

Chapter 8, *Internode Coordination*, explores exported resources. We use exported resources to configure NFS, haproxy, and iptables.

Chapter 9, *External Tools and the Puppet Ecosystem*, shows you how to extend Puppet with your own types and providers, how to make your own facts, as well as some of the more advanced tools such as Puppet-librarian and r10k.

Chapter 10, *Monitoring, Reporting, and Troubleshooting*, is the final chapter where we show you how to leverage Puppet to see where the problems are in your infrastructure. Some of the more common problems are shown with solutions.
“Computer language design is just like a stroll in the park. Jurassic Park, that is.”

— Larry Wall

In this chapter, we will cover the following recipes:

- Adding a resource to a node
- Using Facter to describe a node
- Installing a package before starting a service
- Installing, configuring, and starting a service
- Using community Puppet style
- Creating a manifest
- Checking your manifests with Puppet-lint
- Using modules
- Using standard naming conventions
- Using inline templates
- Iterating over multiple items
- Writing powerful conditional statements
- Using regular expressions in if statements
- Using selectors and case statements
- Using the in operator
- Using regular expression substitutions
- Using the future parser
# Introduction

In this chapter, we'll start with the basics of Puppet syntax and show you how some of the syntactic sugar in Puppet is used. We'll then move on to how Puppet deals with dependencies and how to make Puppet do the work for you.

We'll look at how to organize and structure your code into modules following community conventions, so that other people will find it easy to read and maintain your code. I'll also show you some powerful features of Puppet language, which will let you write concise, yet expressive manifests.

## Adding a resource to a node

This recipe will introduce the language and show you the basics of writing Puppet code. A beginner may wish to reference *Puppet 3: Beginner's Guide*, John Arundel, Packt Publishing in addition to this section. Puppet code files are called manifests; manifests declare resources. A resource in Puppet may be a type, class, or node. A type is something like a file or package or anything that has a type declared in the language. The current list of standard types is available on puppetlabs website at [https://docs.puppetlabs.com/references/latest/type.html](https://docs.puppetlabs.com/references/latest/type.html). I find myself referencing this site very often. You may define your own types, either using a mechanism, similar to a subroutine, named **defined types**, or you can extend the language using a custom type. Types are the heart of the language; they describe the things that make up a node (node is the word Puppet uses for client computers/devices). Puppet uses resources to describe the state of a node; for example, we will declare the following package resource for a node using a site manifest (`site.pp`).

### How to do it...

Create a `site.pp` file and place the following code in it:

```puppet
node default {
    package { 'httpd':
        ensure => 'installed'
    }
}
```

### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at [http://www.packtpub.com](http://www.packtpub.com). If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files e-mailed directly to you.
Chapter 1

How it works...

This manifest will ensure that any node, on which this manifest is applied, will install a package called 'httpd'. The default keyword is a wildcard to Puppet; it applies anything within the node default definition to any node. When Puppet applies the manifest to a node, it uses a Resource Abstraction Layer (RAL) to translate the package type into the package management system of the target node. What this means is that we can use the same manifest to install the httpd package on any system for which Puppet has a Provider for the package type. Providers are the pieces of code that do the real work of applying a manifest. When the previous code is applied to a node running on a YUM-based distribution, the YUM provider will be used to install the httpd RPM packages. When the same code is applied to a node running on an APT-based distribution, the APT provider will be used to install the httpd DEB package (which may not exist, most debian-based systems call this package apache2; we'll deal with this sort of naming problem later).

Using Facter to describe a node

Facter is a separate utility upon which Puppet depends. It is the system used by Puppet to gather information about the target system (node); facter calls the nuggets of information facts. You may run facter from the command line to obtain real-time information from the system.

How to do it...

1. Use facter to find the current uptime of the system, the uptime fact:
   
   ```
   t@cookbook ~$ facter uptime
   0:12 hours
   ```

2. Compare this with the output of the Linux uptime command:
   
   ```
   t@cookbook ~$ uptime
   01:18:52 up 12 min,  1 user,  load average: 0.00, 0.00, 0.00
   ```

How it works...

When facter is installed (as a dependency for puppet), several fact definitions are installed by default. You can reference each of these facts by name from the command line.

There's more...

Running facter without any arguments causes facter to print all the facts known about the system. We will see in later chapters that facter can be extended with your own custom facts. All facts are available for you to use as variables; variables are discussed in the next section.
Variables

Variables in Puppet are marked with a dollar sign ($) character. When using variables within a manifest, it is preferred to enclose the variable within braces "${myvariable}" instead of "$myvariable". All of the facts from facter can be referenced as top scope variables (we will discuss scope in the next section). For example, the fully qualified domain name (FQDN) of the node may be referenced by "${::fqdn}". Variables can only contain alphabetic characters, numerals, and the underscore character (_). As a matter of style, variables should start with an alphabetic character. Never use dashes in variable names.

Scope

In the variable example explained in the There's more... section, the fully qualified domain name was referred to as ${::fqdn} rather than ${fqdn}; the double colons are how Puppet differentiates scope. The highest level scope, top scope or global, is referred to by two colons (::) at the beginning of a variable identifier. To reduce namespace collisions, always use fully scoped variable identifiers in your manifests. For a Unix user, think of top scope variables as the / (root) level. You can refer to variables using the double colon syntax similar to how you would refer to a directory by its full path. For the developer, you can think of top scope variables as global variables; however, unlike global variables, you must always refer to them with the double colon notation to guarantee that a local variable isn't obscuring the top scope variable.

Installing a package before starting a service

To show how ordering works, we'll create a manifest that installs httpd and then ensures the httpd package service is running.

How to do it...

1. We start by creating a manifest that defines the service:
   
   ```puppet
   service {'httpd':
     ensure => running,
     require => Package['httpd'],
   }
   ```

2. The service definition references a package resource named httpd; we now need to define that resource:
   
   ```puppet
   package {'httpd':
     ensure => 'installed',
   }
   ```
How it works...

In this example, the package will be installed before the service is started. Using `require` within the definition of the `httpd` service ensures that the package is installed first, regardless of the order within the manifest file.

**Capitalization**

Capitalization is important in Puppet. In our previous example, we created a package named `httpd`. If we wanted to refer to this package later, we would capitalize its type (`package`) as follows:

```
Package['httpd']
```

To refer to a class, for example, the `something::somewhere` class, which has already been included/defined in your manifest, you can reference it with the full path as follows:

```
Class['something::somewhere']
```

When you have a defined type, for example the following defined type:

```
example::thing {'one':}
```

The preceding resource may be referenced later as follows:

```
Example::Thing['one']
```

Knowing how to reference previously defined resources is necessary for the next section on metaparameters and ordering.

**Learning metaparameters and ordering**

All the manifests that will be used to define a node are compiled into a catalog. A catalog is the code that will be applied to configure a node. It is important to remember that manifests are not applied to nodes sequentially. There is no inherent order to the application of manifests. With this in mind, in the previous `httpd` example, what if we wanted to ensure that the `httpd` process started after the `httpd` package was installed?

We couldn't rely on the `httpd` service coming after the `httpd` package in the manifests. What we have to do is use metaparameters to tell Puppet the order in which we want resources applied to the node. Metaparameters are parameters that can be applied to any resource and are not specific to any one resource type. They are used for catalog compilation and as hints to Puppet but not to define anything about the resource to which they are attached. When dealing with ordering, there are four metaparameters used:

- `before`
- `require`
- `notify`
- `subscribe`
The before and require metaparameters specify a direct ordering; notify implies before and subscribe implies require. The notify metaparameter is only applicable to services; what notify does is tell a service to restart after the notifying resource has been applied to the node (this is most often a package or file resource). In the case of files, once the file is created on the node, a notify parameter will restart any services mentioned. The subscribe metaparameter has the same effect but is defined on the service; the service will subscribe to the file.

**Trifecta**

The relationship between package and service previously mentioned is an important and powerful paradigm of Puppet. Adding one more resource-type file into the fold, creates what puppeteers refer to as the *trifecta*. Almost all system administration tasks revolve around these three resource types. As a system administrator, you install a package, configure the package with files, and then start the service.

![Diagram of Trifecta (Files require package for directory, service requires files and package)](image)

**Idempotency**

A key concept of Puppet is that the state of the system when a catalog is applied to a node cannot affect the outcome of Puppet run. In other words, at the end of Puppet run (if the run was successful), the system will be in a known state and any further application of the catalog will result in a system that is in the same state. This property of Puppet is known as idempotency. **Idempotency** is the property that no matter how many times you do something, it remains in the same state as the first time you did it. For instance, if you had a light switch and you gave the instruction to turn it on, the light would turn on. If you gave the instruction again, the light would remain on.

**Installing, configuring, and starting a service**

There are many examples of this pattern online. In our simple example, we will create an Apache configuration file under `/etc/httpd/conf.d/cookbook.conf`. The `/etc/httpd/conf.d` directory will not exist until the `httpd` package is installed. After this file is created, we would want `httpd` to restart to notice the change; we can achieve this with a notify parameter.
Chapter 1

How to do it...

We will need the same definitions as our last example; we need the package and service installed. We now need two more things. We need the configuration file and index page (index.html) created. For this, we follow these steps:

1. As in the previous example, we ensure the service is running and specify that the service requires the httpd package:

   ```
   service {'httpd':
       ensure => running,
       require => Package['httpd'],
   }
   ```

2. We then define the package as follows:

   ```
   package {'httpd':
       ensure => installed,
   }
   ```

3. Now, we create the /etc/httpd/conf.d/cookbook.conf configuration file; the /etc/httpd/conf.d directory will not exist until the httpd package is installed. The require metaparameter tells Puppet that this file requires the httpd package to be installed before it is created:

   ```
   file {'/etc/httpd/conf.d/cookbook.conf':
       content => '<VirtualHost *:80>
Servername cookbook
DocumentRoot /var/www/cookbook
</VirtualHost>
',
       require => Package['httpd'],
       notify => Service['httpd'],
   }
   ```

4. We then go on to create an index.html file for our virtual host in /var/www/cookbook. This directory won't exist yet, so we need to create this as well, using the following code:

   ```
   file {'/var/www/cookbook':
       ensure => directory,
   }
   file {'/var/www/cookbook/index.html':
       content => '<html><h1>Hello World!</h1></html>',
       require => File['/var/www/cookbook'],
   }
   ```
How it works...

The `require` attribute to the file resources tell Puppet that we need the `/var/www/cookbook` directory created before we can create the `index.html` file. The important concept to remember is that we cannot assume anything about the target system (node). We need to define everything on which the target depends. Anytime you create a file in a manifest, you have to ensure that the directory containing that file exists. Anytime you specify that a service should be running, you have to ensure that the package providing that service is installed.

In this example, using metaparameters, we can be confident that no matter what state the node is in before running Puppet, after Puppet runs, the following will be true:

- `httpd` will be running
- The `VirtualHost` configuration file will exist
- `httpd` will restart and be aware of the `VirtualHost` file
- The `DocumentRoot` directory will exist
- An `index.html` file will exist in the `DocumentRoot` directory

Using community Puppet style

If other people need to read or maintain your manifests, or if you want to share code with the community, it's a good idea to follow the existing style conventions as closely as possible. These govern such aspects of your code as layout, spacing, quoting, alignment, and variable references, and the official puppetlabs recommendations on style are available at [http://docs.puppetlabs.com/guides/style_guide.html](http://docs.puppetlabs.com/guides/style_guide.html).

How to do it...

In this section, I'll show you a few of the more important examples and how to make sure that your code is style compliant.

Indentation

Indent your manifests using two spaces (not tabs), as follows:

```plaintext
service {'httpd':
  ensure => running,
}
```
**Quoting**

Always quote your resource names, as follows:

```erb
package { 'exim4':

We cannot do this as follows though:

```erb
package { exim4:

Use single quotes for all strings, except when:

- The string contains variable references such as "${::fqdn}" 
- The string contains character escape sequences such as "\n"

Consider the following code:

```erb
file { '/etc/motd':
  content => "Welcome to ${::fqdn}\n"
}
```

Puppet doesn't process variable references or escape sequences unless they're inside double quotes.

Always quote parameter values that are not reserved words in Puppet. For example, the following values are not reserved words:

```erb
name => 'Nucky Thompson',
mode => '0700',
owner => 'deploy',
```

However, these values are reserved words and therefore not quoted:

```erb
ensure => installed,
enable => true,
ensure => running,
```

**False**

There is only one thing in Puppet that is false, that is, the word `false` without any quotes. The string "false" evaluates to `true` and the string "true" also evaluates to `true`. Actually, everything besides the literal `false` evaluates to `true` (when treated as a Boolean):

```erb
if "false" {
  notify { 'True': }
}
if 'false' {
  notify { 'Also true': }
}
```
if false {
  notify { 'Not true': }
}

When this code is run through `puppet apply`, the first two notifies are triggered. The final notify is not triggered; it is the only one that evaluates to false.

**Variables**

Always include curly braces ({}) around variable names when referring to them in strings, for example, as follows:

```bash
source => "puppet:///modules/webserver/${brand}.conf",
```

Otherwise, Puppet's parser has to guess which characters should be a part of the variable name and which belong to the surrounding string. Curly braces make it explicit.

**Parameters**

Always end lines that declare parameters with a comma, even if it is the last parameter:

```bash
service { 'memcached':
  ensure => running,
  enable => true,
}
```

This is allowed by Puppet, and makes it easier if you want to add parameters later, or reorder the existing parameters.

When declaring a resource with a single parameter, make the declaration all on one line and with no trailing comma, as shown in the following snippet:

```bash
package { 'puppet': ensure => installed }
```

Where there is more than one parameter, give each parameter its own line:

```bash
package { 'rake':
  ensure => installed,
  provider => gem,
  require => Package['rubygems'],
}
```

To make the code easier to read, line up the parameter arrows in line with the longest parameter, as follows:

```bash
file { "/var/www/${app}/shared/config/rvmrc":
  owner => 'deploy',
  group => 'deploy',
}
content => template('rails/rvmrc.erb'),
require => File["/var/www/${app}/shared/config"],
}

The arrows should be aligned per resource, but not across the whole file, otherwise it can make it difficult for you to cut and paste code from one file to another.

**Symlinks**

When declaring file resources which are symlinks, use ensure => link and set the target attribute, as follows:

```erb
file { '/etc/php5/cli/php.ini':
  ensure => link,
  target => '/etc/php.ini',
}
```

---

**Creating a manifest**

If you already have some Puppet code (known as a Puppet manifest), you can skip this section and go on to the next. If not, we'll see how to create and apply a simple manifest.

**How to do it...**

To create and apply a simple manifest, follow these steps:

1. First, install Puppet locally on your machine or create a virtual machine and install Puppet on that machine. For YUM-based systems, use [https://yum.puppetlabs.com/](https://yum.puppetlabs.com/) and for APT-based systems, use [https://apt.puppetlabs.com/](https://apt.puppetlabs.com/). You may also use `gem` to install Puppet. For our examples, we'll install Puppet using `gem` on a Debian Wheezy system (hostname: `cookbook`). To use `gem`, we need the `rubygems` package as follows:

```bash
@cookbook:~$ sudo apt-get install rubygems
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  rubygems
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 0 B/597 kB of archives.
After this operation, 3,844 kB of additional disk space will be used.
```
Selecting previously unselected package rubygems.
(Reading database ... 30390 files and directories currently installed.)
Unpacking rubygems (from .../rubygems_1.8.24-1_all.deb) ...
Processing triggers for man-db ...
Setting up rubygems (1.8.24-1) ...

2. Now, use gem to install Puppet:

t@cookbook $ sudo gem install puppet
Successfully installed hiera-1.3.4
Fetching: facter-2.3.0.gem (100%)
Successfully installed facter-2.3.0
Fetching: puppet-3.7.3.gem (100%)
Successfully installed puppet-3.7.3
Installing ri documentation for hiera-1.3.4
Installing ri documentation for facter-2.3.0
Installing ri documentation for puppet-3.7.3
Done installing documentation for hiera, facter, puppet after 239 seconds

3. Three gems are installed. Now, with Puppet installed, we can create a directory to contain our Puppet code:

t@cookbook:~$ mkdir -p .puppet/manifests
t@cookbook:~$ cd .puppet/manifests
t@cookbook:~/.puppet/manifests$

4. Within your manifests directory, create the site.pp file with the following content:

    node default {
        file { '/tmp/hello':
            content => "Hello, world!\n",
        }
    }

5. Test your manifest with the puppet apply command. This will tell Puppet to read the manifest, compare it to the state of the machine, and make any necessary changes to that state:

    t@cookbook:~/.puppet/manifests$ puppet apply site.pp
Notice: Compiled catalog for cookbook in environment production in 0.14 seconds
Notice: /Stage[main]/Main/Node[default]/File[/tmp/hello]/ensure: defined content as '{md5}746308829575e17c3331bbcb00c0898b'
Notice: Finished catalog run in 0.04 seconds
6. To see if Puppet did what we expected (create the /tmp/hello file with the Hello, world! content), run the following command:

```bash
t@cookbook:-/puppet/manifests$ cat /tmp/hello
Hello, world!
t@cookbook:-/puppet/manifests$
```

Note that creating the file in /tmp did not require special permissions. We did not run Puppet via `sudo`. Puppet need not be run through `sudo`; there are cases where running via an unprivileged user can be useful.

**There's more...**

When several people are working on a code base, it's easy for style inconsistencies to creep in. Fortunately, there's a tool available which can automatically check your code for compliance with the style guide: `puppet-lint`. We'll see how to use this in the next section.

**Checking your manifests with Puppet-lint**

The puppetlabs official style guide outlines a number of style conventions for Puppet code, some of which we've touched on in the preceding section. For example, according to the style guide, manifests:

- Must use two-space soft tabs
- Must not use literal tab characters
- Must not contain trailing white space
- Should not exceed an 80 character line width
- Should align parameter arrows (=>) within blocks

Following the style guide will make sure that your Puppet code is easy to read and maintain, and if you're planning to release your code to the public, style compliance is essential.

The `puppet-lint` tool will automatically check your code against the style guide. The next section explains how to use it.
Getting ready

Here's what you need to do to install Puppet-lint:

1. We'll install Puppet-lint using the gem provider because the gem version is much more up to date than the APT or RPM packages available. Create a puppet-lint.pp manifest as shown in the following code snippet:

```puppet
package {'puppet-lint':
    ensure => 'installed',
    provider => 'gem',
}
```

2. Run `puppet apply` on the puppet-lint.pp manifest, as shown in the following command:

```
t@cookbook ~$ puppet apply puppet-lint.pp
Notice: Compiled catalog for node1.example.com in environment production in 0.42 seconds
Notice: /Stage[main]/Main/Package[puppet-lint]/ensure: created
Notice: Finished catalog run in 2.96 seconds
```

```
t@cookbook ~$ gem list puppet-lint
*** LOCAL GEMS ***
puppet-lint (1.0.1)
```

How to do it...

Follow these steps to use Puppet-lint:

1. Choose a Puppet manifest file that you want to check with Puppet-lint, and run the following command:

```
t@cookbook ~$ puppet-lint puppet-lint.pp
WARNING: indentation of => is not properly aligned on line 2
ERROR: trailing whitespace found on line 4
```

2. As you can see, Puppet-lint found a number of problems with the manifest file. Correct the errors, save the file, and rerun Puppet-lint to check that all is well. If successful, you'll see no output:

```
t@cookbook ~$ puppet-lint puppet-lint.pp
```

There's more...

You can find out more about Puppet-lint at https://github.com/rodjek/puppet-lint.
Should you follow Puppet style guide and, by extension, keep your code lint-clean? It's up to you, but here are a couple of things to think about:

- It makes sense to use some style conventions, especially when you're working collaboratively on code. Unless you and your colleagues can agree on standards for whitespace, tabs, quoting, alignment, and so on, your code will be messy and difficult to read or maintain.

- If you're choosing a set of style conventions to follow, the logical choice would be that issued by puppetlabs and adopted by the community for use in public modules.

Having said that, it's possible to tell Puppet-lint to ignore certain checks if you've chosen not to adopt them in your codebase. For example, if you don't want Puppet-lint to warn you about code lines exceeding 80 characters, you can run Puppet-lint with the following option:

```
t@cookbook ~$ puppet-lint --no-80chars-check
```

Run `puppet-lint --help` to see the complete list of check configuration commands.

**See also**

- The Automatic syntax checking with Git hooks recipe in Chapter 2, Puppet Infrastructure
- The Testing your Puppet manifests with rspec-puppet recipe in Chapter 9, External Tools and the Puppet Ecosystem

**Using modules**

One of the most important things you can do to make your Puppet manifests clearer and more maintainable is to organize them into modules.

Modules are self-contained bundles of Puppet code that include all the files necessary to implement a thing. Modules may contain flat files, templates, Puppet manifests, custom fact declarations, augeas lenses, and custom Puppet types and providers.

Separating things into modules makes it easier to reuse and share code; it's also the most logical way to organize your manifests. In this example, we'll create a module to manage memcached, a memory caching system commonly used with web applications.
How to do it...

Following are the steps to create an example module:

1. We will use Puppet's module subcommand to create the directory structure for our new module:

   ```bash
   t@cookbook:~$ mkdir -p .puppet/modules
   t@cookbook:~$ cd .puppet/modules
   t@cookbook:~/.puppet/modules$ puppet module generate thomas-memcached
   ```

   We need to create a metadata.json file for this module. Please answer the following questions; if the question is not applicable to this module, feel free to leave it blank. Puppet uses Semantic Versioning (semver.org) to version modules.

   What version is this module?  [0.1.0]
   --> Who wrote this module?  [thomas]
   --> What license does this module code fall under?  [Apache 2.0]
   --> How would you describe this module in a single sentence?
   --> A module to install memcached
   --> Where is this module's source code repository?
   --> Where can others go to learn more about this module?
   --> Where can others go to file issues about this module?
   -->

   ----------------------------------------

   ```json
   {
     "name": "thomas-memcached",
     "version": "0.1.0",
     "author": "thomas",
     "summary": "A module to install memcached",
     "license": "Apache 2.0",
     "source": "",
     "issues_url": null,
     "project_page": null,
     "dependencies": [
       {
         "version_range": ">= 1.0.0",
         "name": "puppetlabs-stdlib"
       }
     ]
   }
   ```
About to generate this metadata; continue? [n/Y] --> y

Notice: Generating module at /home/thomas/.puppet/modules/thomas-memcached...
Notice: Populating ERB templates...
Finished; module generated in thomas-memcached.

thomas-memcached/manifests
thomas-memcached/manifests/init.pp
thomas-memcached/spec
thomas-memcached/spec/classes
thomas-memcached/spec/classes/init_spec.rb
thomas-memcached/spec/spec_helper.rb
thomas-memcached/README.md
thomas-memcached/metadata.json
thomas-memcached/Rakefile
thomas-memcached/tests
thomas-memcached/tests/init.pp

This command creates the module directory and creates some empty files as starting points. To use the module, we'll create a symlink to the module name (memcached).

t@cookbook:~/.puppet/modules$ ln -s thomas-memcached memcached

2. Now, edit memcached/manifests/init.pp and change the class definition at the end of the file to the following. Note that puppet module generate created many lines of comments; in a production module you would want to edit those default comments:

```ruby
class memcached {
  package { 'memcached':
    ensure => installed,
  }

  file { '/etc/memcached.conf':
    source => 'puppet:///modules/memcached/memcached.conf',
    owner => 'root',
    group => 'root',
    mode  => '0644',
    require => Package['memcached'],
  }
}
```

---

Chapter 1

---
service { 'memcached':
  ensure  => running,
  enable  => true,
  require => [Package['memcached'],
               File['/etc/memcached.conf']],
}

3. Create the modules/thomas-memcached/files directory and then create a file named memcached.conf with the following contents:

   -m 64
   -p 11211
   -u nobody
   -l 127.0.0.1

4. Change your site.pp file to the following:

   node default {
     include memcached
   }

5. We would like this module to install memcached. We'll need to run Puppet with root privileges, and we'll use sudo for that. We'll need Puppet to be able to find the module in our home directory; we can specify this on the command line when we run Puppet as shown in the following code snippet:

   t@cookbook:-$ sudo puppet apply --modulepath=/home/thomas/.puppet/modules /home/thomas/.puppet/manifests/site.pp
   Notice: Compiled catalog for cookbook.example.com in environment production in 0.33 seconds
   Notice: /Stage[main]/Memcached/File[/etc/memcached.conf]/content: content changed '{md5}a977521922a151c959ac953712840803' to '{md5}9429eff3e3354c0be232a020bcf78f75'
   Notice: Finished catalog run in 0.11 seconds

6. Check whether the new service is running:

   t@cookbook:-$ sudo service memcached status
   [ ok ] memcached is running.

How it works...

When we created the module using Puppet's module generate command, we used the name thomas-memcached. The name before the hyphen is your username or your username on Puppet forge (an online repository of modules). Since we want Puppet to be able to find the module by the name memcached, we make a symbolic link between thomas-memcached and memcached.
Modules have a specific directory structure. Not all of these directories need to be present, but if they are, this is how they should be organized:

```
modules/
└MODULE_NAME/
  └examples/  example usage of the module
  └files/  flat files used by the module
  └lib/
    └facter/  define new facts for facter
    └puppet/
      └parser/
        └functions/  define a new puppet function, like sort()
    └provider/
    └util/
    └type/
  └manifests/
    └init.pp  class MODULE_NAME { }
  └spec/  rSpec  tests
  └templates/  erb template files used by the module
```

All manifest files (those containing Puppet code) live in the manifests directory. In our example, the memcached class is defined in the manifests/init.pp file, which will be imported automatically.

Inside the memcached class, we refer to the memcached.conf file:

```
file { '/etc/memcached.conf':
  source => 'puppet:///modules/memcached/memcached.conf',
}
```

The preceding source parameter tells Puppet to look for the file in:

```
MODULEPATH/ (/home/thomas/.puppet/modules)
└memcached/
  └files/
    └memcached.conf
```

**There's more...**

Learn to love modules because they'll make your Puppet life a lot easier. They're not complicated, however, practice and experience will help you judge when things should be grouped into modules, and how best to arrange your module structure. Modules can hold more than manifests and files as we'll see in the next two sections.
Templates

If you need to use a template as a part of the module, place it in the module's templates directory and refer to it as follows:

```puppet
file { '/etc/memcached.conf':
  content => template('memcached/memcached.conf.erb'),
}
```

Puppet will look for the file in:

```
MODULEPATH/memcached/templates/memcached.conf.erb
```

Facts, functions, types, and providers

Modules can also contain custom facts, custom functions, custom types, and providers.

For more information about these, refer to Chapter 9, External Tools and the Puppet Ecosystem.

Third-party modules

You can download modules provided by other people and use them in your own manifests just like the modules you create. For more on this, see Using Public Modules recipe in Chapter 7, Managing Applications.

Module organization

For more details on how to organize your modules, see puppetlabs website:

http://docs.puppetlabs.com/puppet/3/reference/modules_fundamentals.html

See also

- The Creating custom facts recipe in Chapter 9, External Tools and the Puppet Ecosystem
- The Using public modules recipe in Chapter 7, Managing Applications
- The Creating your own resource types recipe in Chapter 9, External Tools and the Puppet Ecosystem
- The Creating your own providers recipe in Chapter 9, External Tools and the Puppet Ecosystem

Using standard naming conventions

Choosing appropriate and informative names for your modules and classes will be a big help when it comes to maintaining your code. This is even truer if other people need to read and work on your manifests.
Here are some tips on how to name things in your manifests:

1. Name modules after the software or service they manage, for example, apache or haproxy.

2. Name classes within modules (subclasses) after the function or service they provide to the module, for example, apache::vhosts or rails::dependencies.

3. If a class within a module disables the service provided by that module, name it disabled. For example, a class that disables Apache should be named apache::disabled.

4. Create a roles and profiles hierarchy of modules. Each node should have a single role consisting of one or more profiles. Each profile module should configure a single service.

5. The module that manages users should be named user.

6. Within the user module, declare your virtual users within the class user::virtual (for more on virtual users and other resources, see the Using virtual resources recipe in Chapter 5, Users and Virtual Resources).

7. Within the user module, subclasses for particular groups of users should be named after the group, for example, user::sysadmins or user::contractors.

8. When using Puppet to deploy the config files for different services, name the file after the service, but with a suffix indicating what kind of file it is, for example:
   - Apache init script: apache.init
   - Logrotate config snippet for Rails: rails.logrotate
   - Nginx vhost file for mywizzoapp: mywizzoapp.vhost.nginx
   - MySQL config for standalone server: standalone.mysql

9. If you need to deploy a different version of a file depending on the operating system release, for example, you can use a naming convention like the following:
   ```
   memcached.lucid.conf
   memcached.precise.conf
   ```

10. You can have Puppet automatically select the appropriate version as follows:
    ```
        source = > "puppet:///modules/memcached
               /memcached.${::lsbdistrelease}.conf",
    ```

11. If you need to manage, for example, different Ruby versions, name the class after the version it is responsible for, for example, ruby192 or ruby186.
Puppet community maintains a set of best practice guidelines for your Puppet infrastructure, which includes some hints on naming conventions:

http://docs.puppetlabs.com/guides/best_practices.html

Some people prefer to include multiple classes on a node by using a comma-separated list, rather than separate include statements, for example:

```puppet
node 'server014' inherits 'server' {
  include mail::server, repo::gem, repo::apt, zabbix
}
```

This is a matter of style, but I prefer to use separate include statements, one on a line, because it makes it easier to copy and move around class inclusions between nodes without having to tidy up the commas and indentation every time.

I mentioned inheritance in a couple of the preceding examples; if you're not sure what this is, don't worry, I'll explain this in detail in the next chapter.

**Using inline templates**

Templates are a powerful way of using Embedded Ruby (ERB) to help build config files dynamically. You can also use ERB syntax directly without having to use a separate file by calling the `inline_template` function. ERB allows you to use conditional logic, iterate over arrays, and include variables.

**How to do it...**

Here's an example of how to use `inline_template`:

Pass your Ruby code to `inline_template` within Puppet manifest, as follows:

```puppet
cron { 'chkrootkit':
  command => '/usr/sbin/chkrootkit >
  /var/log/chkrootkit.log 2>&1',
  hour    => inline_template('<%= @hostname.sum % 24 %>'),
  minute  => '00',
}
```
How it works...

Anything inside the string passed to inline_template is executed as if it were an ERB template. That is, anything inside the <%= and %> delimiters will be executed as Ruby code, and the rest will be treated as a string.

In this example, we use inline_template to compute a different hour for this cron resource (a scheduled job) for each machine, so that the same job does not run at the same time on all machines. For more on this technique, see the Distributing cron jobs efficiently recipe in Chapter 6, Managing Resources and Files.

There's more...

In ERB code, whether inside a template file or an inline_template string, you can access your Puppet variables directly by name using an @ prefix, if they are in the current scope or the top scope (facts):

<%= @fqdn %>

To reference variables in another scope, use scope.lookupvar, as follows:

<%= "The value of something from otherclass is " + scope.lookupvar('otherclass::something') %>

You should use inline templates sparingly. If you really need to use some complicated logic in your manifest, consider using a custom function instead (see the Creating custom functions recipe in Chapter 9, External Tools and the Puppet Ecosystem).

See also

- The Using ERB templates recipe in Chapter 4, Working with Files and Packages
- The Using array iteration in templates recipe in Chapter 4, Working with Files and Packages

Iterating over multiple items

Arrays are a powerful feature in Puppet; wherever you want to perform the same operation on a list of things, an array may be able to help. You can create an array just by putting its content in square brackets:

$lunch = [ 'franks', 'beans', 'mustard' ]
How to do it...

Here's a common example of how arrays are used:

1. Add the following code to your manifest:
   ```
   $packages = [ 'ruby1.8-dev',
                 'ruby1.8',
                 'ri1.8',
                 'rdoc1.8',
                 'irb1.8',
                 'libreadline-ruby1.8',
                 'libruby1.8',
                 'libopenssl-ruby' ]
   ```
   ```
   package { $packages: ensure => installed }
   ```

2. Run Puppet and note that each package should now be installed.

How it works...

Where Puppet encounters an array as the name of a resource, it creates a resource for each element in the array. In the example, a new package resource is created for each of the packages in the $packages array, with the same parameters (ensure => installed). This is a very compact way to instantiate many similar resources.

There's more...

Although arrays will take you a long way with Puppet, it's also useful to know about an even more flexible data structure: the hash.

Using hashes

A hash is like an array, but each of the elements can be stored and looked up by name (referred to as the key), for example (hash.pp):

```
$interface = {
    'name' => 'eth0',
    'ip'   => '192.168.0.1',
    'mac'  => '52:54:00:4a:60:07'
}
```
When we run Puppet on this, we see the following notify in the output:

```
When we run Puppet on this, we see the following notify in the output:

```

```
t@cookbook:~/.puppet/manifests$ puppet apply hash.pp
Notice: (192.168.0.1) at 52:54:00:4a:60:07 on eth0
```

Hash values can be anything that you can assign to variables, strings, function calls, expressions, and even other hashes or arrays. Hashes are useful to store a bunch of information about a particular thing because by accessing each element of the hash using a key, we can quickly find the information for which we are looking.

**Creating arrays with the split function**

You can declare literal arrays using square brackets, as follows:

```puppet
define lunchprint() {
  notify { "Lunch included \$\{name\}"; }
}
```

```
$lunch = ['egg', 'beans', 'chips']
lunchprint { $lunch: }
```

Now, when we run Puppet on the preceding code, we see the following notice messages in the output:

```
Notice: Lunch included chips
Notice: Lunch included beans
Notice: Lunch included egg
```

However, Puppet can also create arrays for you from strings, using the **split** function, as follows:

```puppet
$menu = 'egg beans chips'
$items = split($menu, ' ')
lunchprint { $items: }
```

Running `puppet apply` against this new manifest, we see the same messages in the output:

```
Notice: Lunch included chips
Notice: Lunch included beans
Notice: Lunch included egg.
```
Note that `split` takes two arguments: the first argument is the string to be split. The second argument is the character to split on; in this example, a single space. As Puppet works its way through the string, when it encounters a space, it will interpret it as the end of one item and the beginning of the next. So, given the string 'egg beans chips', this will be split into three items.

The character to split on can be any character or string:

```puppet
$menu = 'egg and beans and chips'
$items = split($menu, ' and ')
```

The character can also be a regular expression, for example, a set of alternatives separated by a `|` (pipe) character:

```puppet
$lunch = 'egg:beans, chips'
$items = split($lunch, ':|,')
```

## Writing powerful conditional statements

Puppet's `if` statement allows you to change the manifest behavior based on the value of a variable or an expression. With it, you can apply different resources or parameter values depending on certain facts about the node, for example, the operating system, or the memory size.

You can also set variables within the manifest, which can change the behavior of included classes. For example, nodes in data center A might need to use different DNS servers than nodes in data center B, or you might need to include one set of classes for an Ubuntu system, and a different set for other systems.

### How to do it...

Here's an example of a useful conditional statement. Add the following code to your manifest:

```puppet
if $::timezone == 'UTC' {
   notify { 'Universal Time Coordinated': }
} else {
   notify { "$::timezone is not UTC": }
}
```

### How it works...

Puppet treats whatever follows an `if` keyword as an expression and evaluates it. If the expression evaluates to true, Puppet will execute the code within the curly braces.

Optionally, you can add an else branch, which will be executed if the expression evaluates to false.
There's more...

Here are some more tips on using if statements.

**Elsif branches**

You can add further tests using the elsif keyword, as follows:

```plaintext
if $::timezone == 'UTC' {
    notify { 'Universal Time Coordinated': }
} elsif $::timezone == 'GMT' {
    notify { 'Greenwich Mean Time': }
} else {
    notify { "$::timezone is not UTC": }
}
```

**Comparisons**

You can check whether two values are equal using the == syntax, as in our example:

```plaintext
if $::timezone == 'UTC' {

}
```

Alternatively, you can check whether they are not equal using !=:

```plaintext
if $::timezone != 'UTC' {
 ...
}
```

You can also compare numeric values using < and >:

```plaintext
if $::uptime_days > 365 {
    notify { 'Time to upgrade your kernel!': }
}
```

To test whether a value is greater (or less) than or equal to another value, use <= or >=:

```plaintext
if $::mtu_eth0 <= 1500 {
    notify { "Not Jumbo Frames": }
}
```
Combining expressions

You can put together the kind of simple expressions described previously into more complex logical expressions, using and, or, and not:

```
if ($::uptime_days > 365) and ($::kernel == 'Linux') {
  ...
}
```

```
if ($role == 'webserver') and { ($datacenter == 'A') or ($datacenter == 'B') } {
  ...
}
```

See also

- The Using the in operator recipe in this chapter
- The Using selectors and case statements recipe in this chapter

Using regular expressions in if statements

Another kind of expression you can test in if statements and other conditionals is the regular expression. A regular expression is a powerful way to compare strings using pattern matching.

How to do it...

This is one example of using a regular expression in a conditional statement. Add the following to your manifest:

```
if $::architecture =~ /64/ {
  notify { '64Bit OS Installed': }
} else {
  notify { 'Upgrade to 64Bit': }
  fail('Not 64 Bit')
}
```
How it works...

Puppet treats the text supplied between the forward slashes as a regular expression, specifying the text to be matched. If the match succeeds, the if expression will be true and so the code between the first set of curly braces will be executed. In this example, we used a regular expression because different distributions have different ideas on what to call 64bit; some use amd64, while others use x86_64. The only thing we can count on is the presence of the number 64 within the fact. Some facts that have version numbers in them are treated as strings to Puppet. For instance, $::facterversion. On my test system, this is 2.0.1, but when I try to compare that with 2, Puppet fails to make the comparison:

```
Error: comparison of String with 2 failed at /home/thomas/.puppet/manifests/version.pp:1 on node cookbook.example.com
```

If you wanted instead to do something if the text does not match, use !~ rather than =~:

```
if $::kernel !~ /Linux/ {
    notify { 'Not Linux, could be Windows, MacOS X, AIX, or ?': }
}
```

There's more...

Regular expressions are very powerful, but can be difficult to understand and debug. If you find yourself using a regular expression so complex that you can’t see at a glance what it does, think about simplifying your design to make it easier. However, one particularly useful feature of regular expressions is the ability to capture patterns.

Capturing patterns

You can not only match text using a regular expression, but also capture the matched text and store it in a variable:

```
$input = 'Puppet is better than manual configuration'
if $input =~ /(.*) is better than (.*)/ {
    notify { "You said '$0'. Looks like you're comparing $1 to $2!": }
}
```

The preceding code produces this output:

**You said 'Puppet is better than manual configuration'. Looks like you're comparing Puppet to manual configuration!**

The variable $0 stores the whole matched text (assuming the overall match succeeded). If you put brackets around any part of the regular expression, it creates a group, and any matched groups will also be stored in variables. The first matched group will be $1, the second $2, and so on, as shown in the preceding example.
Regular expression syntax

Puppet's regular expression syntax is the same as Ruby's, so resources that explain Ruby's regular expression syntax will also help you with Puppet. You can find a good introduction to Ruby's regular expression syntax at this website:

http://www.tutorialspoint.com/ruby/ruby_regular_expressions.htm

See also

- Refer to the Using regular expression substitutions recipe in this chapter

Using selectors and case statements

Although you could write any conditional statement using if, Puppet provides a couple of extra forms to help you express conditionals more easily: the selector and the case statement.

How to do it...

Here are some examples of selector and case statements:

1. Add the following code to your manifest:

   ```
   $systemtype = $::operatingsystem ? {
     'Ubuntu' => 'debianlike',
     'Debian' => 'debianlike',
     'RedHat' => 'redhatlike',
     'Fedora' => 'redhatlike',
     'CentOS' => 'redhatlike',
     default => 'unknown',
   }

   notify { "You have a $systemtype system": }
   ```

2. Add the following code to your manifest:

   ```
   class debianlike {
     notify { 'Special manifest for Debian-like systems': }
   }

   class redhatlike {
     notify { 'Special manifest for RedHat-like systems': }
   }

   case $::operatingsystem {
     'Ubuntu',
     'Debian',
     'RedHat',
     'Fedora',
     'CentOS',
     default => 'unknown',
   }
   ```
Our example demonstrates both the selector and the case statement, so let's see in detail how each of them works.

Selector

In the first example, we used a selector (the ?? operator) to choose a value for the $systemtype variable depending on the value of $::operatingsystem. This is similar to the ternary operator in C or Ruby, but instead of choosing between two possible values, you can have as many values as you like.

Puppet will compare the value of $::operatingsystem to each of the possible values we have supplied in Ubuntu, Debian, and so on. These values could be regular expressions (for example, for a partial string match, or to use wildcards), but in our case, we have just used literal strings.

As soon as it finds a match, the selector expression returns whatever value is associated with the matching string. If the value of $::operatingsystem is Fedora, for example, the selector expression will return the redhatlike string and this will be assigned to the variable $systemtype.

Case statement

Unlike selectors, the case statement does not return a value. case statements come in handy when you want to execute different code depending on the value of some expression. In our second example, we used the case statement to include either the debianlike or redhatlike class, depending on the value of $::operatingsystem.
Again, Puppet compares the value of $::operatingsystem to a list of potential matches. These could be regular expressions or strings, or as in our example, comma-separated lists of strings. When it finds a match, the associated code between curly braces is executed. So, if the value of $::operatingsystem is Ubuntu, then the code including debianlike will be executed.

**There's more...**

Once you've got a grip of the basic use of selectors and case statements, you may find the following tips useful.

**Regular expressions**

As with if statements, you can use regular expressions with selectors and case statements, and you can also capture the values of the matched groups and refer to them using $1, $2, and so on:

```puppet
case $::lsbdistdescription {
    /Ubuntu (.+)/: {
        notify { "You have Ubuntu version \$1": }
    }
    /CentOS (.+)/: {
        notify { "You have CentOS version \$1": }
    }
    default: {}
}
```

**Defaults**

Both selectors and case statements let you specify a default value, which is chosen if none of the other options match (the style guide suggests you always have a default clause defined):

```puppet
$lunch = 'Filet mignon.'
$lunchtype = $lunch ? {
    /fries/ => 'unhealthy',
    /salad/ => 'healthy',
    default => 'unknown',
}

notify { "Your lunch was \$lunchtype": }
```

The output is as follows:

```
t@mylaptop - $ puppet apply lunchtype.pp
```

Notice: Your lunch was unknown
Chapter 1

Notice: /Stage[main]/Main/Notify[Your lunch was unknown]/message: defined 'message' as 'Your lunch was unknown'

When the default action shouldn't normally occur, use the fail() function to halt the Puppet run.

Using the in operator

The in operator tests whether one string contains another string. Here's an example:

```python
if 'spring' in 'springfield'
```

The preceding expression is true if the spring string is a substring of springfield, which it is. The in operator can also test for membership of arrays as follows:

```python
if $crewmember in ['Frank', 'Dave', 'HAL ']
```

When in is used with a hash, it tests whether the string is a key of the hash:

```python
$ifaces = { 'lo' => '127.0.0.1', 'eth0' => '192.168.0.1' }
if 'eth0' in $ifaces {
    notify { "eth0 has address ${ifaces['eth0']}'": } }
```

How to do it...

The following steps will show you how to use the in operator:

1. Add the following code to your manifest:

   ```python
   if $::operatingsystem in ['Ubuntu', 'Debian'] {
       notify { 'Debian-type operating system detected': }
   } elsif $::operatingsystem in ['RedHat', 'Fedora', 'SuSE', 'CentOS'] {
       notify { 'RedHat-type operating system detected': }
   } else {
       notify { 'Some other operating system detected': }
   }
   ```

2. Run Puppet:

   ```bash
t@cookbook:~/.puppet/manifests$ puppet apply in.pp
   Notice: Compiled catalog for cookbook.example.com in environment production in 0.03 seconds
   Notice: Debian-type operating system detected
   ```
Puppet Language and Style

Notice: /Stage[main]/Main/Notify[Debian-type operating system detected]/message: defined 'message' as 'Debian-type operating system detected'
Notice: Finished catalog run in 0.02 seconds

There's more...

The value of an in expression is Boolean (true or false) so you can assign it to a variable:

```puppet
$debianlike = $::operatingsystem in [ 'Debian', 'Ubuntu' ]
```

```puppet
if $debianlike {
    notify { 'You are in a maze of twisty little packages, all alike': }
}
```

Using regular expression substitutions

Puppet's `regsubst` function provides an easy way to manipulate text, search and replace expressions within strings, or extract patterns from strings. We often need to do this with data obtained from a fact, for example, or from external programs.

In this example, we'll see how to use `regsubst` to extract the first three octets of an IPv4 address (the network part, assuming it's a /24 class C address).

How to do it...

Follow these steps to build the example:

1. Add the following code to your manifest:
   ```puppet
   $class_c = regsubst($::ipaddress, '(.*)\..*', '\1.0')
   notify { "The network part of $::ipaddress is $class_c": }
   ```
2. Run Puppet:
   ```bash
t@cookbook:-/.puppet/manifests$ puppet apply ipaddress.pp
   Notice: Compiled catalog for cookbook.example.com in environment production in 0.02 seconds
   Notice: The network part of 192.168.122.148 is 192.168.122.0
   ```
How it works...

The `regsubst` function takes at least three parameters: source, pattern, and replacement. In our example, we specified the source string as `$::ipaddress`, which, on this machine, is as follows:

```
192.168.122.148
```

We specify the pattern function as follows:

```
(.*).*
```

We specify the replacement function as follows:

```
\1.0
```

The pattern captures all of the string up to the last period (\.) in the \1 variable. We then match on \.*\, which matches everything to the end of the string, so when we replace the string at the end with \1.0\, we end up with only the network portion of the IP address, which evaluates to the following:

```
192.168.122.0
```

We could have got the same result in other ways, of course, including the following:

```
$class_c = regsubst($::ipaddress, '\.\d+$', '.0')
```

Here, we only match the last octet and replace it with .0, which achieves the same result without capturing.

There's more...

The pattern function can be any regular expression, using the same (Ruby) syntax as regular expressions in `if` statements.

See also

- The *Importing dynamic information* recipe in Chapter 3, *Writing Better Manifests*
- The *Getting information about the environment* recipe in Chapter 3, *Writing Better Manifests*
- The *Using regular expressions in if statements* recipe in this chapter
Using the future parser

Puppet language is evolving at the moment; many features that are expected to be included in the next major release (4) are available if you enable the future parser.

Getting ready

- Ensure that the \texttt{rgen} gem is installed.
- Set \texttt{parser = future} in the \texttt{[main]} section of your \texttt{puppet.conf} (/etc/puppet/puppet.conf for open source Puppet as root, /etc/puppetlabs/puppet/puppet.conf for Puppet Enterprise, and~/.puppet/puppet.conf for a non-root user running puppet).
- To temporarily test with the future parser, use \texttt{--parser=future} on the command line.

How to do it...

Many of the experimental features deal with how code is evaluated, for example, in an earlier example we compared the value of the \texttt{:$::facterversion} fact with a number, but the value is treated as a string so the code fails to compile. Using the future parser, the value is converted and no error is reported as shown in the following command line output:

\begin{verbatim}
t@cookbook:~/.puppet/manifests$ puppet apply --parser=future version.pp
Notice: Compiled catalog for cookbook.example.com in environment production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
\end{verbatim}

Appending to and concatenating arrays

You can concatenate arrays with the \texttt{+} operator or append them with the \texttt{<<} operator. In the following example, we use the ternary operator to assign a specific package name to the \texttt{$apache} variable. We then append that value to an array using the \texttt{<<} operator:

\begin{verbatim}
$apache = $::osfamily ? {
    'Debian' => 'apache2',
    'RedHat' => 'httpd'
}
$packages = ['memcached'] << $apache
package {$packages: ensure => installed}
\end{verbatim}

If we have two arrays, we can use the \texttt{+} operator to concatenate the two arrays. In this example, we define an array of system administrators (\texttt{$sysadmins}) and another array of application owners (\texttt{$appowners}). We can then concatenate the array and use it as an argument to our allowed users:
$sysadmins = [ 'thomas','john','josko' ]
$appowners = [ 'mike', 'patty', 'erin' ]
$users = $sysadmins + $appowners
notice ($users)

When we apply this manifest, we see that the two arrays have been joined as shown in the following command line output:

t@cookbook:~/.puppet/manifests$ puppet apply --parser=future concat.pp
Notice: [thomas, john, josko, mike, patty, erin]
Notice: Compiled catalog for cookbook.example.com in environment production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds

Merging Hashes

If we have two hashes, we can merge them using the same + operator we used for arrays. Consider our $interfaces hash from a previous example; we can add another interface to the hash:

$iface = {
  'name' => 'eth0',
  'ip' => '192.168.0.1',
  'mac' => '52:54:00:4a:60:07'
} + {'route' => '192.168.0.254'}
notice ($iface)

When we apply this manifest, we see that the route attribute has been merged into the hash (your results may differ, the order in which the hash prints is unpredictable), as follows:

t@cookbook:~/.puppet/manifests$ puppet apply --parser=future hash2.pp
Notice: {route => 192.168.0.254, name => eth0, ip => 192.168.0.1, mac => 52:54:00:4a:60:07}
Notice: Compiled catalog for cookbook.example.com in environment production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds

Lambda functions

Lambda functions are iterators applied to arrays or hashes. You iterate through the array or hash and apply an iterator function such as each, map, filter, reduce, or slice to each element of the array or key of the hash. Some of the lambda functions return a calculated array or value; others such as each only return the input array or hash.

Lambda functions such as map and reduce use temporary variables that are thrown away after the lambda has finished. Use of lambda functions is something best shown by example. In the next few sections, we will show an example usage of each of the lambda functions.
Reduce
Reduce is used to reduce the array to a single value. This can be used to calculate the maximum or minimum of the array, or in this case, the sum of the elements of the array:

```ruby
$count = [1,2,3,4,5]
$sum = reduce($count) | $total, $i | {$total + $i }
notice("Sum is $sum")
```

This preceding code will compute the sum of the $count array and store it in the $sum variable, as follows:

```
t@cookbook:~/.puppet/manifests$ puppet apply --parser future lambda.pp
Notice: Sum is 15
Notice: Compiled catalog for cookbook.example.com in environment production in 0.36 seconds
Notice: Finished catalog run in 0.03 seconds
```

Filter
Filter is used to filter the array or hash based upon a test within the lambda function. For instance to filter our $count array as follows:

```ruby
$filter = filter ($count) | $i | {$i > 3 }
notice("Filtered array is $filter")
```

When we apply this manifest, we see that only elements 4 and 5 are in the result:

```
Notice: Filtered array is [4, 5]
```

Map
Map is used to apply a function to each element of the array. For instance, if we wanted (for some unknown reason) to compute the square of all the elements of the array, we would use map as follows:

```ruby
$map = map ($count) | $i | {$i * $i }
notice("Square of array is $map")
```

The result of applying this manifest is a new array with every element of the original array squared (multiplied by itself), as shown in the following command line output:

```
Notice: Square of array is [1, 4, 9, 16, 25]
```

Slice
Slice is useful when you have related values stored in the same array in a sequential order. For instance, if we had the destination and port information for a firewall in an array, we could split them up into pairs and perform operations on those pairs:
$firewall_rules = ['192.168.0.1','80','192.168.0.10','443']
slice ($firewall_rules,2) |$ip, $port| { notice("Allow $ip on $port") }  

When applied, this manifest will produce the following notices:

Notice: Allow 192.168.0.1 on 80
Notice: Allow 192.168.0.10 on 443

To make this a useful example, create a new firewall resource within the block of the slice instead of notice:

slice ($firewall_rules,2) |$ip, $port| {
  firewall {"$port from $ip":
    dport => $port,
    source => "$ip",
    action => "accept",
  }
}

Each

Each is used to iterate over the elements of the array but lacks the ability to capture the results like the other functions. Each is the simplest case where you simply wish to do something with each element of the array, as shown in the following code snippet:

each ($count) |$c| { notice($c) }

As expected, this executes the notice for each element of the $count array, as follows:

Notice: 1
Notice: 2
Notice: 3
Notice: 4
Notice: 5

Other features

There are other new features of Puppet language available when using the future parser. Some increase readability or compactness of code. For more information, refer to the documentation on puppetlabs website at http://docs.puppetlabs.com/puppet/latest/reference/experiments_future.html.
Where to buy this book
You can buy Puppet Cookbook Third Edition from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.