



Learning Puppet

(Generated on September 16, 2011, from git revision
8830f89918de137c687bd1616c0054e57ca66394)

Learning Puppet

The web (including this site) is full of guides for how to solve specific problems with Puppet and how to get Puppet running. This is something slightly different. □

Start: [Resources and the RAL](#) →

Latest: [Templates](#) and [Modules \(Part Two\)](#) →

Welcome

This is Learning Puppet, and it's part of the Puppet documentation. Specifically, it's the first part. □

By which I don't mean it's about getting Puppet installed, or making sure your SSL certificates got issued correctly; that's the other first part. To be a little gnomic about it — because why not — this series is less about how to use Puppet than it is about how to become a Puppet user. If you've heard good things about Puppet but don't know where to start, this, hopefully, is it.

It's a work in progress, and I'd love to read your feedback at nick.fagerlund@puppetlabs.com.

Get Equipped

You can't make a knowledge omelette without breaking... stuff. Possibly eggs, maybe your system's entire configuration! Such is life. □

So to learn Puppet effectively, you need a virtual machine you can experiment on fearlessly. And to learn Puppet fast, you want a virtual machine with Puppet already installed, so you don't have to learn to debug SSL problems before you know how to classify a node.

In short, you want this virtual machine:

Get the Learning Puppet VM

The root user's password is puppet, and for your convenience, the system is configured to write its current IP address to the login screen about ten seconds after it boots.

If you'd rather cook up your own VM than download one from the web, you can imitate it fairly easily: this is a stripped-down CentOS 5.5 system with a hostname of "learn.puppet.demo," [Puppet Enterprise](#) installed using all default answers, iptables turned off, and the `pe-puppet` and `pe-httpd` services stopped and disabled. (It also has Puppet language modes installed for Vim and Emacs, but that's not strictly necessary.)

To begin with, you won't need separate agent and master VMs; you'll be running Puppet in its

serverless mode on a single machine. When we get to agent/master Puppet, we'll walk through turning on the puppet master and duplicating this system into a new agent node.

Compatibility Notes

The Learning Puppet VM is available in VMWare .vmx format and the cross-platform OVF format, and has been tested with VMWare Fusion and VirtualBox.

Getting the VM working with VMWare is fairly simple, but some extra effort is necessary on VirtualBox — the IP address it prints isn't externally reachable, and by default you'll be unable to SSH to the VM. You can enable SSH by turning on port forwarding, but since several examples (and the eventual agent/master exercises) will require more network access than simply port 22, it's wiser to configure two network interfaces:

- Before starting the VM, modify its network settings to add "Host Only" network access on adapter 2.
- Next, run `ifconfig` on your host machine and confirm the subnet assigned to the `vboxnet0` interface. (By default, this is 192.168.56.x, and your host machine's IP address is 192.168.56.1.)
- Once the VM is running, log in on its console and run `ifconfig eth1 192.168.56.2` (or some other IP address compatible with the relevant subnet); this should let you ping and SSH the box from your host machine, and you can add an entry to your host machine's `/etc/hosts` file to make things more convenient.

Beyond this, teaching the use of virtualization software is outside the scope of this introduction, but [let me know](#) if you run into trouble, and we'll try to refine our approach over time.

Hit the Gas

And with that, you're ready to start.

Part one: Serverless Puppet

- Begin with [Resources and the RAL](#), where you'll learn about the fundamental building blocks of system configuration.
- After that, move on to [Manifests](#) and start controlling your system by writing actual Puppet code.
- Next, in [Ordering](#), learn about dependencies and refresh events, manage the relationships between resources, and discover the most useful Puppet design pattern.
- In [Variables, Conditionals, and Facts](#), make your manifests versatile by reading system information.
- In [Modules and Classes \(Part One\)](#), take the first step to a knowable and elegant site design and start turning your manifests into self-contained modules.
- In [Templates](#), use ERB to make your config files as flexible as your Puppet manifests.
- In [Modules and \(Parameterized\) Classes \(Part Two\)](#), learn how to pass parameters to classes and make your modules more adaptable.

And come back soon, because there are more chapters on the way.

Learning — Resources and the RAL

Resources are the building blocks of Puppet, and the division of resources into types and providers is what gives Puppet its power.

You are at the beginning. — [Index](#) — [Manifests](#) →

Molecules

Imagine a system's configuration as a collection of molecules; call them “resources.”

These pieces vary in size, complexity, and lifespan: a user account can be a resource, as can a specific file, a software package, a running service, or a scheduled cron job. Even a single invocation of a shell command can be a resource.

Any resource is very similar to a class of related things: every file has a path and an owner, and every user has a name, a UID, and a group. Which is to say: similar resources can be grouped into types. Furthermore, the most important attributes of a resource type are usually conceptually identical across operating systems, regardless of how the implementations differ. That is, the description of a resource can be abstracted away from its implementation.

These two insights form Puppet's resource abstraction layer (RAL). The RAL splits resources into types (high-level models) and providers (platform-specific implementations), and lets you describe resources in a way that can apply to any system.

Sync: Read, Check, Write

Puppet uses the RAL to both read and modify the state of resources on a system. Since it's a declarative system, Puppet starts with an understanding of what state a resource should have. To sync the resource, it uses the RAL to query the current state, compares that against the desired state, then uses the RAL again to make any necessary changes.

Anatomy of a Resource

In Puppet, every resource is an instance of a resource type and is identified by a title; it has a number of attributes (which are defined by its type), and each attribute has a value.

The Puppet language represents a resource like this:

```
user { 'dave':  
  ensure => present,  
  uid    => '507',  
}
```

```
gid      => 'admin',
shell    => '/bin/zsh',
home     => '/home/dave',
managehome => true,
}
```

This syntax is the heart of the Puppet language, and you'll be seeing it a lot. Hopefully you can already see how it lays out all of the resource's parts (type, title, attributes, and values) in a fairly straightforward way.

The Resource Shell

Puppet ships with a tool called `puppet resource`, which uses the RAL to let you query and modify your system from the shell. Use it to get some experience with the RAL before learning to write and apply manifests.

`puppet resource`'s first argument is a resource type. If executed with no further arguments...□

```
$ puppet resource user
```

... it will query the system and return every resource of that type it can recognize in the system's current state.

You can retrieve a specific resource's state by providing a resource name as a second argument.□

```
$ puppet resource user root

user { 'root':
  home => '/var/root',
  shell => '/bin/sh',
  uid => '0',
  ensure => 'present',
  password => '*',
  gid => '0',
  comment => 'System Administrator'
}
```

Note that `puppet resource` returns Puppet code when it reads a resource from the system! You can use this code later to restore the resource to the state it's in now.

If any attribute=value pairs are provided as additional arguments to `puppet resource`, it will modify the resource, which can include creating it or destroying it:

```
$ puppet resource user dave ensure=present shell="/bin/zsh" home="/home/dave"
managehome=true

notice: /User[dave]/ensure: created
```

```
user { 'dave':  
  ensure => 'present',  
  home => '/home/dave',  
  shell => '/bin/zsh'  
}
```

(Note that this command line assignment syntax differs from the Puppet language's normal `attribute => value` syntax.)

Finally, if you specify a resource and use the `--edit` flag, you can change that resource in your text editor; after the buffer is saved and closed, puppet resource will modify the resource to match your changes.

The Core Resource Types

Puppet has a number of built-in types, and new native types can be distributed with modules. Puppet's core types, the ones you'll get familiar with first, are [notify](#), [file](#), [package](#), [service](#), [exec](#), [cron](#), [user](#), and [group](#). Don't worry about memorizing them immediately, since we'll be covering various resources as we use them, but do take a second to print out a copy of the [core types cheat sheet](#), a double-sided page covering these eight types. It is doctor-recommended¹ and has been clinically shown to treat reference inflammation.

Documentation for all of the built-in types [can always be found in the reference section of this site](#), and can be generated on the fly with the puppet describe utility.

An Aside: puppet describe -s

You can get help for any of the Puppet executables by running them with the `--help` flag, but it's worth pausing for an aside on puppet describe's `-s` flag.

```
$ puppet describe -s user
```

```
user  
====
```

```
Manage users.  This type is mostly built to manage system  
users, so it is lacking some features useful for managing normal  
users.
```

```
This resource type uses the prescribed native tools for creating  
groups and generally uses POSIX APIs for retrieving information  
about them.  It does not directly modify `/etc/passwd` or anything.
```

```
Parameters
```

```
-----
```

```
allowdupe, auth_membership, auths, comment, ensure, expiry, gid, groups,  
home, key_membership, keys, managehome, membership, name, password,  
password_max_age, password_min_age, profile_membership, profiles,
```

```
project, role_membership, roles, shell, uid
```

```
Providers
```

```
-----
```

```
directoryservice, hpuxuseradd, ldap, pw, user_role_add, useradd
```

-s makes puppet describe dump a compact list of the given resource type’s attributes and providers. This isn’t useful when learning about a type for the first time or looking up allowed values, but it’s fantastic when you have the name of an attribute on the tip of your tongue or you can’t remember which two out of “group,” “groups,” and “gid” are applicable for the user type.

Next

Puppet resource can be useful for one-off jobs, but Puppet was born for greater things. [Time to write some manifests](#).

1. The core types cheat sheet is not actually doctor-recommended. If you’re a sysadmin with an M.D., please email me so I can change this footnote. [↪](#)

Learning — Manifests

You understand the RAL; now learn about manifests and start writing and applying Puppet code.

← [Resources and the RAL](#) — [Index](#) — [Resource Ordering](#) →

No Strings Attached

You probably already know that Puppet usually runs in an agent/master (that is, client/server) configuration, but ignore that for now. It’s not important yet and you can get a lot done without it, so for the time being, we have no strings on us.

Instead, we’re going to use [puppet apply](#), which applies a manifest on the local system. It’s the simplest way to run Puppet, and it works like this:

```
$ puppet apply my_test_manifest.pp
```

Yeah, that easy.

You can use puppet — that is, without any subcommand — as a shortcut for puppet apply; it has the rockstar parking in the UI because of how often it runs at an interactive command line. I’ll mostly be saying “puppet apply” for clarity’s sake.

The behavior of Puppet’s man pages is currently in flux. You can always get help for Puppet’s

command line tools by running the tool with the `--help` flag; in the Learning Puppet VM, which uses Puppet Enterprise, you can also run `pe-man puppet apply` to get the same help in a different format. Versions of Puppet starting with the upcoming 2.7 will use Git-style man pages (`man puppet-apply`) with improved formatting.

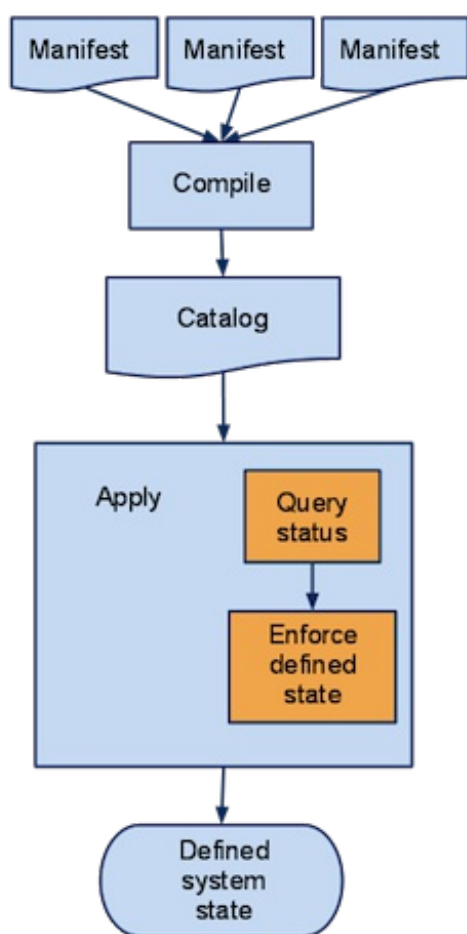
Manifests

Puppet programs are called “manifests,” and they use the `.pp` file extension.

The core of the Puppet language is the resource declaration, which represents the desired state of one resource. Manifests can also use conditional statements, group resources into collections, generate text with functions, reference code in other manifests, and do many other things, but it all ultimately comes down to making sure the right resources are being managed the right way.

An Aside: Compilation

Manifests don’t get used directly when Puppet syncs resources. Instead, the flow of a Puppet run goes a little like this:



Before being applied, manifests get compiled into a “catalog,” which is a [directed acyclic graph](#) that only represents resources and the order in which they need to be synced. All of the conditional

logic, data lookup, variable interpolation, and resource grouping gets computed away during compilation, and the catalog doesn't have any of it.

Why? Several really good reasons, which we'll get to once we rediscover agent/master Puppet;¹ it's not urgent at the moment. But I'm mentioning it now as kind of an experiment: I think there are several things in Puppet that are easy to explain if you understand that split and quite baffling if you don't, so try keeping this in the back of your head and we'll see if it pays off later.□

OK, enough about that; let's write some code! This will all be happening on your main Learning Puppet VM, so log in as root now; you'll probably want to stash these test manifests somewhere convenient, like `/root/learning-manifests`.

Resource Declarations

Let's start by just declaring a single resource:

```
# /root/training-manifests/1.file.pp

file {'testfile':
  path    => '/tmp/testfile',
  ensure  => present,
  mode    => 0640,
  content => "I'm a test file.",
}
```

And apply!

```
# puppet apply 1.file.pp
notice: /Stage[main]/File[testfile]/ensure: created
# cat /tmp/testfile
I'm a test file.
# ls -lah /tmp/testfile
-rw-r----- 1 root root 16 Feb 23 13:15 /tmp/testfile
```

You've seen this syntax before, but let's take a closer look at the language here.

- First, you have the type ("file"), followed by...□
- ...a pair of curly braces that encloses everything else about the resource. Inside those, you have...
 - ...the resource title, followed by a colon...
 - ...and then a set of attribute => value pairs describing the resource.


A few other notes about syntax:

- Missing commas and colons are the number one syntax error made by learners. If you take out the comma after `ensure => present` in the example above, you'll get an error like this:

```
Could not parse for environment production: Syntax error at 'mode';
expected '}' at /root/manifests/1.file.pp:6 on node barn2.magpie.lan
```

Missing colons do about the same thing. So watch for that. Also, although you don't strictly need the comma after the final attribute `=>` value pair, you should always put it there anyhow. Trust me.

- Capitalization matters! You can't declare a resource with `File {'testfile:' ...`, because that does something entirely different. (Specifically, it breaks. But it's kind of similar to what we use to tweak an existing resource, which we'll get to later.)
- Quoting values matters! Built-in values like `present` shouldn't be quoted, but normal strings should be. For all intents and purposes, everything is a string, including numbers. Puppet uses the same rules for single and double quotes as everyone else:
 - Single quotes are completely literal, except that you write a literal quote with `\'` and a literal backslash with `\\`.
 - Double quotes let you interpolate `$variables` and add newlines with `\n`.
- Whitespace is fungible for readability. Lining up the `=>` arrows (sometimes called "fat commas") is good practice if you ever expect someone else to read this code — note that future and mirror universe versions of yourself count as "someone else."

 *Exercise: Declare a file resource in a manifest and apply it! Try changing the login message by setting the content of `/etc/motd`.*

Once More, With Feeling!

Okay, you sort of have the idea by now. Let's make a whole wad of totally useless files! (And throw in some notify resources for good measure.)

```
# /root/training-manifests/2.file.pp

file {'/tmp/test1':
  ensure => present,
  content => "Hi.",
}

file {'/tmp/test2':
  ensure => directory,
  mode   => 0644,
}

file {'/tmp/test3':
  ensure => link,
  target => '/tmp/test1',
}
```

```
}

notify {"I'm notifying you."} # Whitespace is fungible, remember.
notify {"So am I!"}
```

```
# puppet apply 2.file.pp
notice: /Stage[main]//File[/tmp/test2]/ensure: created
notice: /Stage[main]//File[/tmp/test3]/ensure: created
notice: /Stage[main]//File[/tmp/test1]/ensure: created
notice: I'm notifying you.
notice: /Stage[main]//Notify[I'm notifying you.]/message: defined 'message' as
'I'm notifying you.'
notice: So am I!
notice: /Stage[main]//Notify[So am I!]/message: defined 'message' as 'So am I!'

# ls -lah /tmp/test*
-rw-r--r--  1 root root    3 Feb 23 15:54 test1
lrwxrwxrwx  1 root root   10 Feb 23 15:54 test3 -> /tmp/test1
-rw-r-----  1 root root   16 Feb 23 15:05 testfile

/tmp/test2:
total 16K
drwxr-xr-x  2 root root  4.0K Feb 23 16:02 .
drwxrwxrwt  5 root root  4.0K Feb 23 16:02 ..

# cat /tmp/test3
Hi.
```

That was totally awesome. What just happened?

Titles and Namevars

All right, notice how we left out some important attributes there and everything still worked? Almost every resource type has one attribute whose value defaults to the resource's title. For the file resource, that's path; with notify, it's message. A lot of the time (user, group, package...), it's plain old name.

To people who occasionally delve into the Puppet source code, the one attribute that defaults to the title is called the "namevar," which is a little weird but as good a name as any. It's almost always the attribute that amounts to the resource's identity, the one thing that should always be unique about each instance.

This can be a convenient shortcut, but be wary of overusing it; there are several common cases where it makes more sense to give a resource a symbolic title and assign its name (`-var`) as a normal attribute. In particular, it's a good idea to do so if a resource's name is long or you want to assign the name conditionally depending on the nature of the system.

```
notify {'bignotify':
  message => "I'm completely enormous, and will mess up the formatting of
```

```
your
    code! Also, since I need to fire before some other resource, you'll
need
    to refer to me by title later using the Notify['title'] syntax, and
you
    really don't want to have to type this all over again.",
}
```

The upshot is that our notify {"I'm notifying you."} resource above has the exact same effect as:

```
notify {'other title':
  message => "I'm notifying you.",
}
```

... because the message attribute just steals the resource title if you don't give it anything of its own.

You can't declare the same resource twice: Puppet will always keep you from making resources with duplicate titles, and will almost always keep you from making resources with duplicate name/namevar values. (exec resources are the main exception.)

And finally, you don't need an encyclopedic memory of what the namevar is for each resource —□ when in doubt, just choose a descriptive title and specify the attributes you need.

644 = 755 For Directories

We said /tmp/test2/ should have permissions mode 0644, but our `ls -lah` showed mode 0755. That's because Puppet groups the read bit and the traverse bit for directories, which is almost always what you actually want. The idea is to let you recursively manage whole directories as mode 0644 without making all their files executable.□

New Ensure Values

The file type has several different values for its ensure attribute: `present`, `absent`, `file`, `directory`, and `link`. They're listed on the [core types cheat sheet](#) whenever you need to refresh your memory, and they're fairly self-explanatory.

The Destination

Here's a pretty crucial part of learning to think like a Puppet user. Try applying that manifest again.

```
# puppet apply 2.file.pp
notice: I'm notifying you.
notice: /Stage[main]//Notify[I'm notifying you.]/message: defined 'message' as
'I'm notifying you.'
notice: So am I!
notice: /Stage[main]//Notify[So am I!]/message: defined 'message' as 'So am I!'
```

And again!

```
# rm /tmp/test3
# puppet apply 2.file.pp
notice: I'm notifying you.
notice: /Stage[main]//Notify[I'm notifying you.]/message: defined 'message' as
'I'm notifying you.'
notice: /Stage[main]//File[/tmp/test3]/ensure: created
notice: So am I!
notice: /Stage[main]//Notify[So am I!]/message: defined 'message' as 'So am I!'
```

The notifies are firing every time, because that's what they're for, but Puppet doesn't do anything with the file resources unless they're wrong on disk; if they're wrong, it makes them right. Remember how I said Puppet was declarative? This is how that pays off: You can apply the same configuration every half hour without having to know anything about how the system currently looks. Manifests describe the destination, and Puppet handles the journey.

Exercise: Write and apply a manifest that'll make sure Apache (httpd) is running, use a web browser on your host OS to view the Apache welcome page, then modify the manifest to turn Apache back off. (Hint: You'll have to check the [cheat sheet](#) or the [types reference](#), because the service type's ensure values differ from the ones you've seen so far.)

Slightly more difficult exercise: Write and apply a manifest that uses the [ssh_authorized_key](#) type to let you log into the learning VM as root without a password. You'll need to already have an SSH key.

Next

Resource declarations: Check! You know how to use the fundamental building blocks of Puppet code, so now it's time to learn [how those blocks fit together](#)

1. There are also a few I can mention now, actually. If you drastically refactor your manifest code and want to make sure it still generates the same configurations, you can just intercept the catalogs and use a special diff tool on them; if the same nodes get the same configurations, you can be sure the code acts the same without having to model the execution of the code in your head. Compiling to a catalog also makes it much easier to simulate applying a configuration, and since the catalog is just data, it's relatively easy to parse and analyze with your own tool of choice.

Learning — Resource Ordering

You understand manifests and resource declarations; now learn about metaparameters, resource

ordering, and one of the most useful patterns in Puppet.

← [Manifests](#) — [Index](#) — [Variables](#) →

Disorder

Let's look back on one of our manifests from the last page:

```
# /root/training-manifests/2.file.pp

file {'/tmp/test1':
  ensure => present,
  content => "Hi.",
}

file {'/tmp/test2':
  ensure => directory,
  mode   => 644,
}

file {'/tmp/test3':
  ensure => link,
  target => '/tmp/test1',
}

notify {"I'm notifying you."}
notify {"So am I!"}
```

Although we wrote these declarations one after another, Puppet might sync them in any order: unlike with a procedural language, the physical order of resources in a manifest doesn't imply a logical order.

But some resources depend on other resources. So how do we tell Puppet which ones go first?□

Metaparameters, Resource References, and Ordering

```
file {'/tmp/test1':
  ensure => present,
  content => "Hi.",
}

notify {'/tmp/test1 has already been synced.':
  require => File['/tmp/test1'],
}
```

Each resource type has its own set of attributes, but there's another set of attributes, called [metaparameters](#), which can be used on any resource. (They're meta because they don't describe any feature of the resource that you could observe on the system after Puppet finishes; they only□

describe how Puppet should act.)

There are four metaparameters that let you arrange resources in order: `before`, `require`, `notify`, and `subscribe`. All of them accept a resource reference (or an array¹ of them). Resource references look like this:

```
Type['title']
```

(Note the square brackets and capitalized resource type!)

AN ASIDE: CAPITALIZATION

The easy way to remember this is that you only use the lowercase type name when declaring a new resource. Any other situation will always call for a capitalized type name.

This will get more important in another couple lessons, so I'll mention it again later.

Before and Require

`before` and `require` make simple dependency relationships, where one resource must be synced before another. `before` is used in the earlier resource, and lists resources that depend on it; `require` is used in the later resource and lists the resources that it depends on.

These two metaparameters are just different ways of writing the same relationship — our example² above could just as easily be written like this:

```
file { ['/tmp/test1':  
  ensure => present,  
  content => "Hi.",  
  before => Notify['/tmp/test1 has already been synced.'],  
  # (See what I meant about symbolic titles being a good idea?)  
}  
  
notify {'/tmp/test1 has already been synced.':}
```

Notify and Subscribe

A few resource types² can be “refreshed” — that is, told to react to changes in their environment. For a service, this usually means restarting when a config file has been changed; for an `exec` resource, this could mean running its payload if any user accounts have been changed. (Note that refreshes are performed by Puppet, so they only occur during Puppet runs.)

The `notify` and `subscribe` metaparameters make dependency relationships the way `before` and `require` do, but they also make refresh relationships. Not only will the earlier resource in the pair get synced first, but if Puppet makes any changes to that resource, it will send a refresh event to the later resource, which will react accordingly.

Chaining

```
file {'/tmp/test1':  
  ensure => present,  
  content => "Hi.",  
}  
  
notify {'after':  
  message => '/tmp/test1 has already been synced.',  
}  
  
File['/tmp/test1'] -> Notify['after']
```

There's one last way to declare relationships: chain resource references with the ordering (`->`) and notification (`~>`; note the tilde) arrows. The arrows can point in either direction (`<-` works too), and you should think of them as representing the flow of time: the resource at the blunt end of the `~>` arrow will be synced before the resource the arrow points at.

The example above yields the same dependency as the two examples before it. The benefit of this `~>` alternate syntax may not be obvious when we're working with simple examples, but it can be much more expressive and legible when we're working with resource collections.

Autorequire

Some of Puppet's resource types will notice when an instance is related to other resources, and they'll set up automatic dependencies. The one you'll use most often is between files and their `parent` directories: if a given file and its parent directory are both being managed as resources, `Puppet` will make sure to sync the parent directory before the file.

Don't sweat much about the details of autorequiring; it's fairly conservative and should generally do the right thing without getting in your way. If you forget it's there and make explicit dependencies, your code will still work.

Summary

So to sum up: whenever a resource depends on another resource, use the `before` or `require` metaparameter or chain the resources with `->`. Whenever a resource needs to refresh when another resource changes, use the `notify` or `subscribe` metaparameter or chain the resources with `~>`. Some resources will autorequire other resources if they see them, which can save you some effort.

Hopefully that's all pretty clear! But even if it is, it's rather abstract — making sure a notify fires after a file is something of a “hello world” use case, and not very illustrative. Let's break something!

Example: sshd

You've probably been using SSH and your favorite terminal app to interact with the Learning Puppet VM, so let's go straight for the most-annoying-case scenario: we'll pretend someone accidentally

gave the wrong person (i.e., us) sudo privileges, and have you ruin root's ability to SSH to this box. We'll use Puppet to bust it and Puppet to fix it.□

First, if you got the `ssh_authorized_key` exercise from the last page working, undo it.

```
# mv ~/.ssh/authorized_keys ~/old_ssh_authorized_keys
```

Now let's get a copy of the current `sshd` config file; going forward, we'll use our new copy as the canonical source for that file.□

```
# cp /etc/ssh/sshd_config ~/learning-manifests/
```

Next, edit our new copy of the file. There's a line in there that says `PasswordAuthentication yes`; find it, and change the `yes` to a `no`. Then start writing some Puppet!□

```
# /root/learning-manifests/break_ssh.pp
file { ['/etc/ssh/sshd_config']:
  ensure => file,
  mode   => 600,
  source => '/root/learning-manifests/sshd_config',
  # And yes, that's the first time we've seen the "source" attribute.
  # It accepts absolute paths and puppet:/// URLs, about which more later.
}
```

Except that won't work! (Don't run it, and if you did, read this footnote.³) If we apply this manifest, the config file will change, but `sshd` will keep acting on the old config file until it restarts... and if it's only restarting when the system reboots, that could be years from now.

If we want the service to change its behavior as soon as we change our policy, we'll have to tell it to monitor the config file.□

```
# /root/learning-manifests/break_ssh.pp, again
file { ['/etc/ssh/sshd_config']:
  ensure => file,
  mode   => 600,
  source => '/root/learning-manifests/sshd_config',
}

service { 'sshd':
  ensure      => running,
  enable      => true,
  hasrestart  => true,
  hasstatus   => true,
  # FYI, those last two attributes default to false, since
  # bad init scripts are more or less endemic.
  subscribe  => File['/etc/ssh/sshd_config'],
}
```

And that'll do it! Run that manifest with puppet apply, and after you log out, you won't be able to SSH into the VM again. Victory.

To fix it, you'll have to log into the machine directly — use the screen provided by your virtualization app. Once you're there, you'll just have to edit `/root/learning-manifests/sshd_config` again to change the `PasswordAuthentication` setting and re-apply the same manifest; Puppet will replace `/etc/ssh/sshd_config` with the new version, restart the service, and re-enable remote password logins. (And you can put your SSH key back now, if you like.)

Package/File/Service

The example we just saw was very close to a pattern you'll see constantly in production Puppet code, but it was missing a piece. Let's complete it:

```
# /root/learning-manifests/break_ssh.pp
package { 'openssh-server':
  ensure => present,
  before => File['/etc/ssh/sshd_config'],
}

file { '/etc/ssh/sshd_config':
  ensure => file,
  mode   => 600,
  source => '/root/learning-manifests/sshd_config',
}

service { 'sshd':
  ensure      => running,
  enable      => true,
  hasrestart  => true,
  hasstatus   => true,
  subscribe   => File['/etc/ssh/sshd_config'],
}
```

This is `package/file/service`, one of the most useful patterns in Puppet: the `package` resource makes sure the software is installed, the `config` file depends on the `package` resource, and the `service` subscribes to changes in the `config` file.

It's hard to understate the importance of this pattern; if this was all you knew how to do with Puppet, you could still do a fair amount of work. But we're not done yet.

Next

Now that you can sync resources in their proper order, it's time to make your manifests aware of the outside world with [variables, facts, and conditionals](#).

1. Arrays in Puppet are made with square brackets and commas, so an array of resource references would `[Notify['look'],`

Notify['like'], Notify['this']].[↩](#)

2. Of the built-in types, only `exec`, `service`, and `mount` can be refreshed.[↩](#)
3. If you DID apply the incomplete manifest, something interesting happened: your machine is now in a half-rolled-out condition that puts the lie to what I said earlier about not having to worry about the system's current state. Since the config file is now in sync with its desired state, Puppet won't change it during the next run, which means applying the complete manifest won't cause the service to refresh until either the source file or the file on the system changes one more time. [↩](#)

In practice, this isn't a huge problem, because only your development machines are likely to end up in this state; your production nodes won't have been given incomplete configurations. In the meantime, you have two options for cleaning up after applying an incomplete manifest: For a one-time fix, echo a bogus comment to the bottom of the file on the system (echo "# ignoreme" >> /etc/ssh/sshd_config), or for a more complete approach, make a comment in the source file that contains a version string, which you can update whenever you make significant changes to the associated manifest(s). Both of these approaches will mark the config file as out of sync, replace it during the Puppet run, and send the refresh event to the service.[↩](#)

Learning — Variables, Conditionals, and Facts

You can write manifests and order resources; now, add logic and flexibility with conditional statements and variables.

[← Ordering](#) — [Index](#) — [Modules \(Part One\)](#) →

Variables

Variables! I'm going to bet you pretty much know this drill, so let's move a little faster:

- `$variables` always start with a dollar sign. You assign to variables with the `=` operator.
- Variables can hold strings, numbers, special values (`false`, `undef`...), arrays, and hashes.
- If you've never assigned a variable, you can actually still use it — its value will be `undef`. (You can also explicitly assign `undef` as a value, although the use case for that is somewhat advanced.)
- You can use variables as the value for any resource attribute, or as the title of a resource.
- You can also interpolate variables inside strings, if you use double-quotes. To distinguish a `${variable}` from the surrounding text, you should wrap its name in curly braces.
- Every variable has a short local name and a long fully-qualified name. Fully qualified variables look like `$scope::variable`. Top scope variables are the same, but their scope is nameless. (For example: `$::top_scope_variable`.)
- If you reference a variable with its short name and it isn't present in the local scope, Puppet will also check the top scope;¹ this means you can effectively leave off the leading `::` on top scope variables and treat them as globals.
- You can only assign the same variable once in a given scope.²

```
$longthing = "Imagine I have something really long in here. Like an SSH key, let's say."
```

```
file {'authorized_keys':  
  path    => '/root/.ssh/authorized_keys',  
  content => $longthing,  
}
```

Pretty easy.

Facts

And now, a teaspoon of magic.

Before you even start writing your manifests, Puppet builds you a stash of pre-assigned variables. Check it out:

```
# hosts-simple.pp  
  
# Host type reference:  
# http://docs.puppetlabs.com/references/stable/type.html#host  
  
host {'self':  
  ensure => present,  
  name   => $::fqdn,  
  host_aliases => ['puppet', $::hostname],  
  ip      => $::ipaddress,  
}  
  
file {'motd':  
  ensure => file,  
  path   => '/etc/motd',  
  mode   => 0644,  
  content => "Welcome to ${::hostname},\na ${::operatingsystem} island in  
the sea of ${::domain}.\n",  
}
```

```
# puppet apply hosts-simple.pp  
  
notice: /Stage[main]//Host[puppet]/ensure: created  
notice: /Stage[main]//File[motd]/ensure: defined content as  
'{md5}d149026e4b6d747ddd3a8157a8c37679'  
  
# cat /etc/hosts  
# HEADER: This file was autogenerated at Mon Apr 25 14:39:11 -0700 2011  
# HEADER: by puppet. While it can still be managed manually, it  
# HEADER: is definitely not recommended.  
# Do not remove the following line, or various programs  
# that require network functionality will fail.  
127.0.0.1 localhost.localdomain localhost
```

```
::1 localhost6.localdomain6 localhost6
172.16.158.137 learn.puppet.demo puppet learn
```

Our manifests are starting to get versatile, with pretty much no real work on our part.

Hostname? IPaddress?

So where did those helpful variables come from? They're "facts." Puppet ships with a tool called `Facter`, which ferrets out your system information, normalizes it into a set of variables, and passes them off to Puppet. The compiler then has access to those facts when it's reading a manifest.□

There are a lot of different facts, and the easiest way to get a list of them is to simply run `facter` at your VM's command line. You'll get back a long list of key/value pairs separated by the familiar `=>` hash rocket. To use one of these facts in your manifests, just prepend a dollar sign to its name (along with a `:`, because being explicit about namespaces is a good habit).

Most kinds of system will have at least a few facts that aren't available on other kinds of system (e.g., try comparing `Facter`'s output on your CentOS VM to what it does on an OS X machine), and it can get fuzzier if you're extending `Facter` with [custom facts](#), but there's a general core of facts that give you the same info everywhere. You'll get a feel for them pretty quickly, and can probably guess most of them just by reading the list of names.

Conditional Statements

Puppet has a fairly complete complement of conditional syntaxes, and the info available in facts makes it really easy to code different behavior for different systems.□

If

We'll start with your basic `if` statement. Same as it ever was:

```
if condition {
  block of code
}
elsif condition {
  block of code
}
else {
  block of code
}
```

The `else` and any number of `elsif` statements are optional.

```
if $is_virtual {
  service {'ntpd':
    ensure => stopped,
    enable => false,
```

```

    }
  }
  else {
    service { 'ntpd':
      name      => 'ntpd',
      ensure    => running,
      enable    => true,
      hasrestart => true,
      require   => Package['ntp'],
    }
  }
}

```

The blocks of code for each condition can contain any Puppet code.

WHAT IS FALSE?

You'll notice I used a bare fact as the condition above. The Puppet language's data types are kind of loose, and a lot of things tend to get represented internally as strings, so it's worth mentioning that the following values will be treated as false by an if statement:

- undef (the value of an unassigned variable)
- '' (the empty string)
- false
- Any expression that evaluates to false.

In particular, be aware that 0 is true.

CONDITIONS

Conditions can get pretty sophisticated: you can use any valid [expression](#) in an if statement. Usually, this is going to mean using one of the standard comparison operators (==, !=, <, >, <=, >=), the regex match operators (=~ and !~), or the in operator (which tests whether the right operand contains the left one).

Case

Also probably familiar: the case statement. (Or switch, or whatever your language of choice calls it.)

```

case $operatingsystem {
  centos: { $apache = "httpd" }
  # Note that these matches are case-insensitive.
  redhat: { $apache = "httpd" }
  debian: { $apache = "apache2" }
  ubuntu: { $apache = "apache2" }
  default: { fail("Unrecognized operating system for webserver") }
  # "fail" is a function. We'll get to those later.
}
package {'apache':
  name    => $apache,
  ensure  => latest,
}

```

Instead of testing a condition up front, case matches a variable against a bunch of possible values. default is a special value, which does exactly what it sounds like.

CASE MATCHING

Matches can be simple strings (like above), regular expressions, or comma-separated lists of either.

String matching is case-insensitive, like the == comparison operator. Regular expressions are denoted with the slash-quoting used by Perl and Ruby; they're case-sensitive by default, but you can use the (?i) and (?-i) switches to turn case-insensitivity on and off inside the pattern. Regexp matches also assign captured subpatterns to \$1, \$2, etc. inside the associated code block, with \$0 containing the whole matching string.

Here's a regex example:

```
case $ipaddress_eth0 {
  /^127[\\d.]+$/: {
    notify {'misconfig':
      message => "Possible network misconfiguration: IP address of $0",
    }
  }
}
```

And here's the example from above, rewritten and more readable:

```
case $operatingsystem {
  centos, redhat: { $apache = "httpd" }
  debian, ubuntu: { $apache = "apache2" }
  default: { fail("Unrecognized operating system for webserver") }
}
```

Selectors

Selectors might be less familiar; they're kind of like the [ternary operator](#), and kind of like the case statement.

Instead of choosing between a set of code blocks, selectors choose between a group of possible values. You can't use them on their own; instead, they're usually used to assign a variable.

```
$apache = $operatingsystem ? {
  centos           => 'httpd',
  redhat           => 'httpd',
  /(?!i)(ubuntu|debian)/ => "apache2-$1",
  # (Don't actually use that package name.)
  default          => undef,
}
```

Careful of the syntax, there: it looks kind of like we're saying `$apache = $operatingsystem`, but we're not. The question mark flags `$operatingsystem` as the pivot of a selector, and the actual value that gets assigned is determined by which option `$operatingsystem` matches. Also note how the syntax differs from the case syntax: it uses hash rockets and line-end commas instead of colons and blocks, and you can't use lists of values in a match. (If you want to match against a list, you have to fake it with a regular expression.)

It can look a little awkward, but there are plenty of situations where it's the most concise way to get a value sorted out; if you're ever not comfortable with it, you can just use a case statement to assign the variable instead.

Selectors can also be used directly as values for a resource attribute, but try not to do that, because it gets ugly fast.

Exercises

Exercise: Use the `$operatingsystem` fact to write a manifest that installs a build environment on Debian-based ("debian" and "ubuntu") and Enterprise Linux-based ("centos," "redhat") machines. (Both types of system require the `gcc` package, but Debian-type systems also require `build-essential`.)

Exercise: Write a manifest that installs and configures NTP for Debian-based and Enterprise Linux-based Linux systems. This will be a `package/file/service` pattern where you'll be shipping different config files ([Debian version](#), [Red Hat version](#) — remember the `file` type's "source" attribute) and using different service names (`ntp` and `ntpd`, respectively).

(Use a second manifest to disable the NTP service after you've gotten this example working; NTP can behave kind of uselessly in a virtual machine.)

Next

Now that your manifests can adapt to different kinds of systems, it's time to start grouping resources and conditionals into meaningful units. Onward to [classes, defined resource types, and modules](#)!

1. It's actually a little more complicated than that, but don't worry about it for now. You can [read up on it](#) later. ↪
2. This has to do with the declarative nature of the Puppet language: the idea is that the order in which you read the file shouldn't matter, so changing a value halfway through is illegal, since it would make the results order-dependent.

In practice, this isn't the full story, because you can't currently read a variable from anywhere north of its assignment. We're

Learning — Modules and Classes (Part One)

You can write some pretty sophisticated manifests at this point, but they're still at a fairly low altitude, going resource-by-resource-by-resource. Now, zoom out with resource collections.

← [Variables, etc.](#) — [Index](#) — [Templates](#) →

Collecting and Reusing

At some point, you're going to have Puppet code that fits into a couple of different buckets: really general stuff that applies to all your machines, more specialized stuff that only applies to certain classes of machines, and very specific stuff that's meant for a few nodes at most.

So... you could just paste in all your more general code as boilerplate atop your more specific code. There are ways to do that and get away with it. But that's the road down into the valley of the 4,000-line manifest. Better to separate your code out into meaningful units and then call those units by name as needed.

Thus, resource collections and modules! In a few minutes, you'll be able to maintain your manifest code in one place and declare whole groups of it like this:

```
class {'security_base': }
class {'webserver_base': }
class {'appserver': }
```

And after that, it'll get even better. But first things first.

Classes

Classes are singleton collections of resources that Puppet can apply as a unit. You can think of them as blocks of code that can be turned on or off.

If you know any object-oriented programming, try to ignore it for a little while, because that's not the kind of class we're talking about. Puppet classes could also be called “roles” or “aspects;” they describe one part of what makes up a system's identity.

Defining

Before you can use a class, you have to define it, which is done with the `class` keyword, a name, and a block of code:

```
class someclass {
```

```
...  
}
```

Well, hey: you have a block of code hanging around from last chapter's exercises, right? Chuck it in!

```
# ntp-class1.pp  
  
class ntp {  
  case $operatingsystem {  
    centos, redhat: {  
      $service_name = 'ntpd'  
      $conf_file     = 'ntp.conf.el'  
    }  
    debian, ubuntu: {  
      $service_name = 'ntp'  
      $conf_file     = 'ntp.conf.debian'  
    }  
  }  
  
  package { 'ntp':  
    ensure => installed,  
  }  
  
  service { 'ntp':  
    name       => $service_name,  
    ensure     => running,  
    enable     => true,  
    subscribe  => File['ntp.conf'],  
  }  
  
  file { 'ntp.conf':  
    path       => '/etc/ntp.conf',  
    ensure     => file,  
    require    => Package['ntp'],  
    source     => "/root/learning-manifests/${conf_file}",  
  }  
}
```

Go ahead and apply that. In the meantime:

AN ASIDE: NAMES, NAMESPACES, AND SCOPE

Class names have to start with a lowercase letter, and can contain lowercase alphanumeric characters and underscores. (Just your standard slightly conservative set of allowed characters.)

Class names can also use a double colon (::<) as a namespace separator. (Yes, this should [look familiar](#).) This is a good way to show which classes are related to each other; for example, you can tell right away that something's going on between `apache::ssl` and `apache::vhost`. This will become more important about [two feet south of here](#).

Also, class definitions introduce new variable scopes. That means any variables you assign within won't be accessible by their short names outside the class; to get at them from elsewhere, you

would have to use the fully-qualified name (e.g. `$apache::ssl::certificate_expiration`). It also means you can localize — mask — variable short names in use outside the class; if you assign a `$fqdn` variable in a class, you would get the new value instead of the value of the `Facter`-supplied variable, unless you used the fully-qualified fact name (`$::fqdn`).

Declaring

Okay, back to our example, which you'll have noticed by now doesn't actually do anything.

```
# puppet apply ntp-class1.pp
(...silence)
```

The code inside the class was properly parsed, but the compiler didn't build any of it into the catalog, so none of the resources got synced. For that to happen, the class has to be declared.

You actually already know the syntax to do that. A class definition just enables a unique instance of the `class` resource type; once it's defined, you can declare it like any other resource:□

```
# ntp-class1.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file     = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file     = 'ntp.conf.debian'
    }
  }

  package { ['ntp']:
    ensure => installed,
  }

  service { ['ntp']:
    name       => $service_name,
    ensure     => running,
    enable     => true,
    subscribe => File['ntp.conf'],
  }

  file { ['ntp.conf']:
    path       => '/etc/ntp.conf',
    ensure     => file,
    require    => Package['ntp'],
    source     => "/root/learning-manifests/${conf_file}",
  }
}
```

```
# Then, declare it:
class {'ntp': }
```

This time, all those resources will end up in the catalog:

```
# puppet apply --verbose ntp-class1.pp

info: Applying configuration version '1305066883'
info: FileBucket adding /etc/ntp.conf as {md5}5baec8bdbf90f877a05f88ba99e63685
info: /Stage[main]/Ntp/File[ntp.conf]: Filebucketed /etc/ntp.conf to puppet
with sum 5baec8bdbf90f877a05f88ba99e63685
notice: /Stage[main]/Ntp/File[ntp.conf]/content: content changed
'{md5}5baec8bdbf90f877a05f88ba99e63685' to
'{md5}dc20e83b436a358997041a4d8282c1b8'
info: /Stage[main]/Ntp/File[ntp.conf]: Scheduling refresh of Service[ntp]
notice: /Stage[main]/Ntp/Service[ntp]/ensure: ensure changed 'stopped' to
'running'
notice: /Stage[main]/Ntp/Service[ntp]: Triggered 'refresh' from 1 events
```

Defining the class makes it available; declaring activates it.□

INCLUDE

There's another way to declare classes, but it behaves a little bit differently:□

```
include ntp
include ntp
include ntp
```

The include function will declare a class if it hasn't already been declared, and will do nothing if it has. This means you can safely use it multiple times, whereas the resource syntax can only be used once. The drawback is that include can't currently be used with parameterized classes, on which more later.

So which should you choose? Neither, yet: learn to use both, and decide later, after we've covered site design and parameterized classes.

Classes In Situ

You've probably already guessed that classes aren't enough: even with the code above, you'd still have to paste the ntp definition into all your other manifests. So it's time to meet the `module` autoloader!

AN ASIDE: PRINTING CONFIG

But first, we'll need to meet its friend, the `modulepath`.

```
# puppet apply --configprint modulepath
/etc/puppetlabs/puppet/modules
```

By the way, `--configprint` is wonderful. Puppet has a lot of [config options](#), all of which have default values and site-specific overrides in `puppet.conf`, and trying to memorize them all is a pain. You can use `--configprint` on most of the Puppet tools, and they'll print a value (or a bunch, if you use `--configprint all`) and exit.

(As for the [modulepath](#), it looks like a single directory at the moment but is actually a colon-separated¹ list of directories.)

Modules

So anyway, modules are re-usable bundles of code and data. Puppet autoloads manifests from the modules in its `modulepath`, which means you can declare a class stored in a module from anywhere. Let's just convert that last class to a module immediately, so you can see what I'm talking about:

```
# cd /etc/puppetlabs/puppet/modules
# mkdir ntp; cd ntp; mkdir manifests; cd manifests
# vim init.pp
```

```
# init.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name = 'ntpd'
      $conf_file     = 'ntp.conf.el'
    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_file     = 'ntp.conf.debian'
    }
  }

  package { ['ntp']:
    ensure => installed,
  }

  service { ['ntp']:
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }

  file { ['ntp.conf']:
    path      => '/etc/ntp.conf',
    ensure    => file,
    require   => Package['ntp'],
    source    => "/root/learning-manifests/${conf_file}",
  }
}
```

```
}  
  
# (Remember not to declare the class yet.)
```

And now, the reveal:²

```
# cd ~  
# puppet apply -e "include ntp"
```

Which works! You can now include the class from any manifest, without having to cut and paste anything.

But we're not quite done yet. See how the manifest is referring to some files stored outside the module? Let's fix that:

```
# mkdir /etc/puppetlabs/puppet/modules/ntp/files  
# mv /root/learning-manifests/ntp.conf.*  
/etc/puppetlabs/puppet/modules/ntp/files/  
# vim /etc/puppetlabs/puppet/modules/ntp/manifests/init.pp
```

```
# ...  
file { 'ntp.conf':  
  path      => '/etc/ntp.conf',  
  ensure    => file,  
  require   => Package['ntp'],  
  # source  => "/root/learning-manifests/${conf_file}",  
  source    => "puppet:///modules/ntp/${conf_file}",  
}  
}
```

There — our little example from last chapter has grown up into a self-contained blob of awesome.

Module Structure

A module is just a directory with stuff in it, and the magic comes from putting that stuff where Puppet expects to find it. Which is to say, arranging the contents like this:

- {module}/
 - files/
 - lib/
 - manifests/
 - init.pp
 - {class}.pp
 - {defined type}.pp
 - {namespace}/

- {class}.pp
- {class}.pp
- templates/
- tests/

The main directory should be named after the module. All of the manifests go in the manifests directory. Each manifest contains only one class (or defined type). There's a special manifest called `init.pp` that holds the module's main class, which should have the same name as the module. That's your barest-bones module: main folder, manifests folder, `init.pp`, just like we used in the `ntp` module above.

But if that was all a module was, it'd make more sense to just load your classes from one flat folder. Modules really come into their own with namespacing and grouping of classes.

Manifests, Namespacing, and Autoloading

The manifests directory can hold any number of other classes and even folders of classes, and Puppet uses [namespacing](#) to find them. Say we have a manifests folder that looks like this:

- foo/
 - manifests/
 - init.pp
 - bar.pp
 - bar/
 - baz.pp

The `init.pp` file should contain `class foo { ... }`, `bar.pp` should contain `class foo::bar { ... }`, and `baz.pp` should contain `class foo::bar::baz { ... }`.

This can be a little disorienting at first, but I promise you'll get used to it. Basically, `init.pp` is special, and all of the other classes (each in its own manifest) should be under the main class's namespace. If you add more levels of directory hierarchy, they get interpreted as more levels of namespace hierarchy. This lets you group related classes together, or split the implementation of a complex resource collection out into conceptually separate bits.

Files

Puppet can serve files from modules, and it works identically regardless of whether you're doing serverless or agent/master Puppet. Everything in the `files` directory in the `ntp` module is available under the `puppet:///modules/ntp/` URL. Likewise, a `test.txt` file in the `testing` module's `files` could be retrieved as `puppet:///modules/testing/test.txt`.

Tests

Once you start writing modules you plan to keep for more than a day or two, read our brief guide to [module smoke testing](#). It's pretty simple, and will eventually pay off.□

Templates

More on [templates](#) later.

Lib

Puppet modules can also serve executable Ruby code from their `lib` directories, to extend Puppet and Facter. (Remember how I mentioned extending Facter with custom facts? This is where they live.) It'll be a while before we cover any of that.

Module Scaffolding□

Since you'll be dealing with those same five subdirectories so much, consider adding a function for□ them to your `~/.bashrc` file.□

```
mkmod() {  
    mkdir "$1"  
    mkdir "$1/files" "$1/lib" "$1/manifests" "$1/templates" "$1/tests"  
}
```

Exercises

Exercise: Build an Apache2 module and class, which ensures Apache is installed and running and manages its config file. While you're at it, make□ Puppet manage the DocumentRoot and put a custom 404 page and default index.html in place.

Set any files or package/service names that might vary per distro conditionally,□ failing if we're not on CentOS; this'll let you cleanly shim in support for other distros once you need it.

We'll be using this module some more in future lessons.

Next

So what's with those static config files we're shipping around? If our classes can do different things□ on different systems, shouldn't our `ntp.conf` and `httpd.conf` files be able to do the same? [Yes. Yes they should.](#)

1. Well, system path separator-separated. On POSIX systems, that's a colon; on Windows, it's a semicolon. ↩
2. The `-e` flag lets you give puppet apply a line of manifest code instead of a file, same as with Perl or Ruby. ↗

Learning — Templates

File serving isn't the be-all/end-all of getting content into place. Before we get to parameterized classes and defined types, take a break to learn about templates, which let you make your config files as versatile as your manifests.

← [Modules \(part one\)](#) — [Index](#) — [Modules \(part two\)](#) →

Templating

Okay: in the last chapter, you built a module that shipped and managed a configuration file, which was pretty cool. And if you expect all your enterprise Linux systems to use the exact same set of NTP servers, that's probably good enough. Except let's say you decide most of your machines should use internal NTP servers — whose `ntpd` configurations are also managed by Puppet, and which should be asking for the time from an external source. The number of files you'll need to ship just multiplied, and they'll only differ by three or four lines, which seems redundant and wasteful.

It would be much better if you could use all the tricks you learned in [Variables, Conditionals, and Facts](#) to rummage around in the actual text of your configuration files. Thus: templates!

Puppet can use [ERB templates](#) anywhere a string is called for. (Like a file's `content` attribute, for instance, or the value of a variable.) Templates go in the (wait for it) `templates/` directory of a module, and will mostly look like normal configuration files (or what-have-you), except for the occasional `<%` tag with Ruby code `%>`.

Yes, Ruby — unfortunately you can't use the Puppet language in templates. But usually you'll only be printing a variable or doing a simple loop, which you'll get a feel for almost instantly. Anyway, let's cut to the chase:

Some Simple ERB

First, keep in mind that facts, global variables, and variables defined in the current scope are available to a template as standard Ruby local variables, which are plain words without a `$` sigil in front of them. Variables from other scopes are reachable, but to read them, you have to call the `lookupvar` method on the scope object. (For example, `scope.lookupvar('apache::user')`.)

Tags

ERB tags are delimited by angle brackets with percent signs just inside. (There isn't any HTML-like concept of opening or closing tags.)

```
<% document = "" %>
```

Tags contain one or more lines of Ruby code, which can set variables, munge data, implement control flow, or... actually, pretty much anything, except for print text in the rendered output.□

Printing an Expression

For that, you need to use a printing tag, which looks like a normal tag with an equals sign right after the opening delimiter:

```
<%= sectionheader %>
environment = <%= gitrevision[0,5] %>
```

The value you print can be a simple variable, or it can be an arbitrarily complicated Ruby expression.

Comments

A tag with a hash mark right after the opening delimiter can hold comments, which aren't interpreted as code and aren't displayed in the rendered output.

```
<## This comment will be ignored. %>
```

Suppressing Line Breaks

Regular tags don't print anything, but if you keep each tag of logic on its own line, the line breaks you use will show up as a swath of whitespace in the final file. If you don't like that, you can make□ ERB trim the line break by putting a hyphen directly before the closing delimiter.

```
<% document += thisline -%>
```

Rendering a Template

To render output from a template, use Puppet's built-in template function:

```
file {'/etc/foo.conf':
  ensure => file,
  require => Package['foo'],
  content => template('foo/foo.conf.erb'),
}
```

This evaluates the template and turns it into a string. Here, we're using that string as the content¹ of a file resource, but like I said above, we could be using it for pretty much anything. Note that the

path to the template doesn't use the same semantics as the path in a puppet:/// URL² — it should be in the form <module name>/<path relative to module's templates directory>. (That is, `template('foo/foo.conf.erb')` points to `/etc/puppetlabs/puppet/modules/foo/templates/foo.conf.erb`.)

As a sidenote: if you give more than one argument to the template function...

```
template('foo/one.erb', 'foo/two.erb')
```

...it will evaluate each of the templates, then concatenate their outputs and return a single string.

An Aside: Other Functions

Since we just went over the template function, this is as good a time as any to cover functions in general.

Most of the Puppet language consists of ways to say “Here is a thing, and this is what it is” — resource declarations, class definitions, variable assignments, and the like. Functions are ways to say “Do something.” They're a bucket for miscellaneous functionality. (You can even write new functions in Ruby and distribute them in modules, if you need to repeatedly munge data or modify your catalogs in some way.)

Puppet's functions are run during catalog compilation,³ and they're pretty intuitive to call; it's basically just `function(argument, argument, argument)`, and you can optionally leave off the parentheses. (Remember that `include` is also a function.) Some functions (like `template`) get replaced with a return value, and others (like `include`) take effect silently.□

You can read the full list of available functions at the [function reference](#). We won't be covering most of these for a while, but you might find `inline_template` and `regsubst` useful in the short term.

An Example: NTP Again

So let's modify your NTP module to use templates instead of static config files.□

First, we'll change the `init.pp` manifest:

```
# init.pp

class ntp {
  case $operatingsystem {
    centos, redhat: {
      $service_name      = 'ntpd'
      $conf_template     = 'ntp.conf.el.erb'
      $default_servers = [ "0.centos.pool.ntp.org",
                          "1.centos.pool.ntp.org",
                          "2.centos.pool.ntp.org", ]
    }
  }
}
```

```

    }
    debian, ubuntu: {
      $service_name = 'ntp'
      $conf_template = 'ntp.conf.debian.erb'
      $default_servers = [ "0.debian.pool.ntp.org iburst",
                          "1.debian.pool.ntp.org iburst",
                          "2.debian.pool.ntp.org iburst",
                          "3.debian.pool.ntp.org iburst", ]
    }
  }

  if $servers == undef {
    $servers_real = $default_servers
  }
  else {
    $servers_real = $servers
  }

  package { 'ntp':
    ensure => installed,
  }

  service { 'ntp':
    name      => $service_name,
    ensure    => running,
    enable    => true,
    subscribe => File['ntp.conf'],
  }

  file { 'ntp.conf':
    path      => '/etc/ntp.conf',
    ensure    => file,
    require   => Package['ntp'],
    content   => template("ntp/${conf_template}"),
  }
}

```

There are several things going on, here:

- We changed the `File['ntp.conf']` resource, as advertised.
- We're storing the servers in an array, mostly so I can demonstrate how to iterate over an array once we get to the template. If you wanted to, you could store them as a string with line breaks and per-line server statements instead; it comes down to a combination of personal style and the nature of the problem at hand.
- We'll be using that `$servers_real` variable in the actual template, which might seem odd now but will make more sense during the next chapter. Likewise with testing whether `$servers` is `undef`. (For now, it will always be `undef`, as are all variables that haven't been assigned yet.)

Next, copy the config files to the templates directory, add the `.erb` extension to their names, and replace the blocks of servers with some choice ERB code:

```
# ...

# Managed by Class['ntp']
<% servers_real.each do |server| -%>
server <%= server %>
<% end -%>

# ...
```

This snippet will iterate over each entry in the array and print it after a server statement, so, for example, the string generated from the Debian template will end up with a block like this:

```
# Managed by Class['ntp']
server 0.debian.pool.ntp.org iburst
server 1.debian.pool.ntp.org iburst
server 2.debian.pool.ntp.org iburst
server 3.debian.pool.ntp.org iburst
```

You can see the limitations here — the servers are still basically hardcoded. But we’ve moved them out of the config file and into the Puppet manifest, which gets us half of the way to a much more flexible NTP class.

Next

And as for the rest of the way, keep reading to learn about [parameterized classes](#).

1. This is a good time to remind you that filling a `content` attribute happens during catalog compilation, and serving a file with `puppet:/// URL` happens during catalog application. Again, this doesn’t matter right now, but it may make some things clearer later.
2. This inconsistency is one of those problems that tend to crop up over time when software grows organically. We’re working on it, and you can keep an eye on [ticket #4885](#) if that sort of thing interests you.
3. To jump ahead a bit, this means the agent never sees them.

Learning — Modules and (Parameterized) Classes (Part Two)

Now that you have basic classes and modules under control, it’s time for some more advanced code re-use.

← [Templates](#) — [Index](#) — TBA →

Investigating vs. Passing Data

Most classes have to do slightly different things on different systems. You already know some ways

to do that — all of the modules you’ve written so far have switched their behaviors by looking up system facts. Let’s say that they “investigate:” they expect some information to be in a specific place (in the case of facts, a top-scope variable), and go looking for it when they need it.

But this isn’t always the best way to do it, and it starts to break down once you need to switch a module’s behavior on information that doesn’t map cleanly to system facts. Is this a database server? A local NTP server?

You could still have your modules investigate; instead of looking at the standard set of system facts, you could just point them to an arbitrary variable and make sure it’s filled if you plan on using that class. But it might be better to just tell the class what it needs to know when you declare it.

Parameters

When defining a class, you can give it a list of parameters.

```
class mysql ($user, $port) { ... }
```

This is a doorway for passing information into the class. When you declare the class, those parameters appear as resource attributes; inside the definition, they appear as local variables.

```
# /etc/puppetlabs/puppet/modules/paramclassexample/manifests/init.pp
class paramclassexample ($value1, $value2 = "Default value") {
  notify {"Value 1 is ${value1}.":}
  notify {"Value 2 is ${value2}.":}
}

# ~/learning-manifests/paramclass1.pp
class {'paramclassexample':
  value1 => 'Something',
  value2 => 'Something else',
}

# ~/learning-manifests/paramclass2.pp
class {'paramclassexample':
  value1 => 'Something',
}
```

```
# puppet apply ~/learning-manifests/paramclass1.pp
notice: Value 2 is Something else.
notice: /Stage[main]/Paramclassexample/Notify[Value 2 is Something
else.]/message: defined 'message' as 'Value 2 is Something else.'
notice: Value 1 is Something.
notice: /Stage[main]/Paramclassexample/Notify[Value 1 is Something.]/message:
defined 'message' as 'Value 1 is Something.'
notice: Finished catalog run in 0.05 seconds

# puppet apply ~/learning-manifests/paramclass2.pp
notice: Value 1 is Something.
```

```
notice: /Stage[main]/Paramclassexample/Notify[Value 1 is Something.]/message:
defined 'message' as 'Value 1 is Something.'
notice: Value 2 is Default value.
notice: /Stage[main]/Paramclassexample/Notify[Value 2 is Default
value.]/message: defined 'message' as 'Value 2 is Default value.'
notice: Finished catalog run in 0.05 seconds
```

(As shown above, you can give any parameter a default value, which makes it optional when you declare the class. Parameters without defaults are required.)

So what's the benefit of all this? In a word, it [encapsulates](#) the class. You don't have to pick unique magic variable names to use as a dead drop, and since anything affecting the function of the class has to pass through the parameters, it's much more clear where the information is coming from. This pays off once you start having to think about how modules work with other modules, and it really pays off if you want to download or create reusable modules.

Example: NTP (Again)

So let's get back to our NTP module. The first thing we talked about wanting to vary was the set of servers, and we already did the heavy lifting back in the [templates](#) chapter, so that's a good place to start:

```
class ntp ($servers = undef) {
  ...
}
```

And... that's all it takes, actually. This will work. If you declare the class with no attributes...

```
class {'ntp':}
```

...it'll work the same way it used to. If you declare it with a servers attribute containing an array of servers (with or without appended `iburst` and `dynamic` statements)...

```
class {'ntp':
  servers => [ "ntp1.puppetlabs.lan dynamic", "ntp2.puppetlabs.lan
dynamic", ],
}
```

...it'll override the servers in the `ntp.conf` file. Nice.

There is a bit of trickery to notice: setting a variable or parameter to `undef` might seem odd, and we're only doing it because we want to be able to get the default servers without asking for them. (Remember, parameters can't be optional without an explicit default value.)

Also, remember the business with the `$servers_real` variable? That was because the Puppet

language won't let us re-assign the `$servers` variable within a given scope. If the default value we wanted was the same regardless of OS, we could just use it as the parameter default, but the extra logic to accomodate the per-OS defaults means we have to make a copy.

While we're in the NTP module, what else could we make into a parameter? Well, let's say you have a mixed environment of physical and virtual machines, and some of them occasionally make the transition between VM and metal. Since NTP behaves weirdly under virtualization, you'd want it turned off on your virtual machines — and you would have to manage the service as a resource to do that, because if you just didn't say anything about NTP (by not declaring the class, e.g.), it might actually still be running. So you could make a separate `ntp_disabled` class and declare it whenever you aren't declaring the `ntp` class... but it makes more sense to expose the service's attributes as class parameters. That way, when you move a formerly physical server into the cloud, you could just change that part of its manifests from this:

```
class {'ntp':}
```

...to this:

```
class {'ntp':  
  ensure => stopped,  
  enable => false,  
}
```

And making that work right is almost as easy as the last edit. Here's the complete class, with all of our modifications thus far:

```
#!/etc/puppetlabs/puppet/modules/ntp/manifests/init.pp  
class ntp ($servers = undef, $enable = true, $ensure = running) {  
  case $operatingsystem {  
    centos, redhat: {  
      $service_name = 'ntpd'  
      $conf_template = 'ntp.conf.el.erb'  
      $default_servers = [ "0.centos.pool.ntp.org",  
                           "1.centos.pool.ntp.org",  
                           "2.centos.pool.ntp.org", ]  
    }  
    debian, ubuntu: {  
      $service_name = 'ntp'  
      $conf_template = 'ntp.conf.debian.erb'  
      $default_servers = [ "0.debian.pool.ntp.org iburst",  
                           "1.debian.pool.ntp.org iburst",  
                           "2.debian.pool.ntp.org iburst",  
                           "3.debian.pool.ntp.org iburst", ]  
    }  
  }  
}  
  
if $servers == undef {
```



```

    $servers_real = $default_servers
  }
  else {
    $servers_real = $servers
  }

  package { 'ntp':
    ensure => installed,
  }

  service { 'ntp':
    name      => $service_name,
    ensure    => $ensure,
    enable    => $enable,
    subscribe => File['ntp.conf'],
  }

  file { 'ntp.conf':
    path      => '/etc/ntp.conf',
    ensure    => file,
    require   => Package['ntp'],
    content   => template("ntp/${conf_template}"),
  }
}

```

Is there anything else we could do to this class? Well, yes: its behavior under anything but Debian, Ubuntu, CentOS, or RHEL is currently undefined, so it'd be nice to, say, come up with some config templates to use under the BSDs and OS X and then fail gracefully on unrecognized OSes. And it might make sense to unify our two current templates; they were just based on the system defaults, and once you decide how NTP should be configured at your site, chances are it's going to look similar on any Unix. This could also let you simplify the default value and get rid of that undef and `$servers_real` dance. But as it stands, this module is pretty serviceable.

So hey, let's throw on some documentation and be done with it!

Module Documentation

```

# Class: ntp
#
# This class installs/configures/manages NTP. It can optionally disable NTP
# on virtual machines. Only supported on Debian-derived and Red Hat-derived
# OSes.
#
# Parameters:
#   - $servers:
#       An array of NTP servers, with or without +iburst+ and
#       +dynamic+ statements appended. Defaults to the OS's defaults.
#   - $enable:
#       Whether to start the NTP service on boot. Defaults to true. Valid
#       values: true and false.
#   - $ensure:
#       Whether to run the NTP service. Defaults to running. Valid

```

```

values:
  #           running and stopped.
  #
  # Requires:
  #   Nothing.
  #
  # Sample Usage:
  #   class {'ntp':
  #     servers => [ "ntp1.puppetlabs.lan dynamic",
  #                 "ntp2.puppetlabs.lan dynamic", ],
  #   }
  #   class {'ntp':
  #     enable => false,
  #     ensure => stopped,
  #   }
  #
  class ntp ($servers = undef, $enable = true, $ensure = running) {
    case $operatingsystem { ...
    ...

```

This doesn't have to be Tolstoy, but seriously, at least write down what the parameters are and what kind of data they take. Your future self will thank you. Also! If you put it in a comment block butted up directly against the start of the class definition, you can automatically generate a browsable Rdoc-style site with info for all your modules. You can test it now, actually:

```

# puppet doc --mode rdoc --outputdir ~/moduledocs --modulepath
/etc/puppetlabs/puppet/modules

```

(Then just upload that ~/moduledocs folder to some webspace you control, or grab it onto your desktop with SFTP.)

Some Important Notes From the Dep't of Foreshadowing

Parameterized classes are still pretty new — they were only added to Puppet in version 2.6.0 — and they changed the landscape of Puppet in some ways that aren't immediately obvious.

You probably noticed that the examples in this chapter are all using the [resource-like](#) declaration syntax instead of the [include](#) function. That's because `include` doesn't work¹ with parameterized classes, and likely never will. The problem is that the whole point of `include` conflicts with the idea that a class can change depending on how it's declared — if you declare a class multiple times and the attributes don't match precisely, which set of attributes wins?

Parameterized classes made the problem with that paradigm more explicit, but it already existed, and it was possible to run afoul of it without even noticing. A common pattern for passing information into a class was to choose an external variable and have the class retrieve it with [dynamically-scoped variable lookup](#).² If you were also having low-level classes manage their own dependencies by including anything they might need, then a given class might have several

potential scope chains resolving to different values, which would result in a race — whichever `include` took effect first would determine the behavior of the class.

However, there were and are a couple of other ways to get data into a class — let's lump them together and call them data separation — and if you used them well, your classes could safely manage their own dependencies with `include`. Using parameterized classes gives you new options for site design that we've come to believe are just plain better, but it closes off that option of self-managed dependencies.

I'm purposely getting ahead of myself a bit — this isn't going to be fully relevant until we start talking about class composition and site design, and we'll be covering data separation later as well. But since all these issues stem from ideas about what a class is and where it gets its information, it seemed worthwhile to mention some of these issues now, just so they don't seem so out-of-the-blue later.

Next

Okay, we can pass parameters into classes now and change their behavior. Great! But classes are still always singletons; you can't declare more than one copy and get two different sets of behavior simultaneously. And you'll eventually want to do that! What if you had a collection of resources that created a vhost definition for a web server, or cloned a Git repository, or managed a user account complete with group, SSH key, home directory contents, sudoers entry, and `.bashrc/.vimrc/etc.` files? What if you wanted more than one Git repo, user account, or vhost on a single machine?

Well, you'd whip up a defined resource type. Come back soon for Modules (part three)!

1. Yes, you can actually `include` a parameterized class if all of its parameters have defaults, but mixing and matching declaration styles for a class is not the best plan.
2. I haven't covered dynamic scope in Learning Puppet, both because it shouldn't be necessary for someone learning today and because its days are numbered.

© 2010 [Puppet Labs](http://puppetlabs.com) info@puppetlabs.com 411 NW Park Street / Portland, OR 97209 1-877-575-9775