Chapter 5. Using the Puppet Configuration Language

Now that you've been introduced to the Puppet manifest and the resource building block, you are ready to meet the Puppet configuration language.

This chapter will introduce you to the data types, operators, conditionals, and iterations that can be used to build manifests for Puppet 4. Writing manifests well is the single most important part of using Puppet. You'll find yourself returning to this chapter again and again as you develop your own manifests.

> **TIP**
>
> Don't stop here if you don't understand how or why to use one piece of the language just yet. Just take in what is possible, and refer to this chapter as you build and grow your skills.

Consider this chapter a reference for the Puppet configuration language. The reasoning behind why and when to use particular bits of the language will become clear as specific implementations are discussed later in the book.

> **PUPPET 4 VERSUS THE FUTURE PARSER**
>
> You'll find statements throughout this book about new features only available in Puppet 4. However, most of these features were available in Puppet 3, if the *future parser* was enabled. So when I say that a language change is "only available in Puppet 4," you can read this to include "or when the future parser is enabled in Puppet 3."
>
> Puppet 4 contains all features and changes previously developed in the future parser. The future is here.
>
> We'll cover how to test Puppet 3 manifests with the future parser in "Testing with the Future Parser".

Defining Variables

Puppet makes use of *variables* (named storage of data) much like any other language you've learned to write code in.

Like many scripting languages, variables are prefaced with a `$` in Puppet. The variable name must start with a lowercase letter or underscore, and may contain lowercase letters, numbers, and underscores. Let's take a look at a few examples of both valid and invalid variable names:

```
$myvar          # valid
$MyVar          # invalid

$my_var         # valid
$my-var         # invalid

$my3numbers     # valid
$3numbers       # invalid
```

> **WARNING**
>
> Previous versions of Puppet allowed uppercase letters, periods, and dashes with inconsistent results. Puppet 4 has improved reliability by enforcing these standards.

Variables starting with underscores should only be used within the local scope (we'll cover variable scope in "Understanding Variable Scope"):

```
$_myvar         # valid inside a local scope
$prog::_myvar   # deprecated: no underscore-prefixed variables out of scope
```

Variables are assigned values with an equals sign. Every value has a data type. The most common types are `Boolean`, `Numeric`, and `String`. As I'm sure you've seen these data types many times before, we'll jump straight to some examples:

```
$my_name        = 'Jo'          # string
$num_tokens     = 115           # numeric
$not_true       = false         # boolean
```

> **NOTE**
>
> Anything after the # (comment mark) is ignored by the interpreter.

A variable that has not been initialized will evaluate as undefined, or `undef`. You can also explicitly assign `undef` to a variable:

```
$notdefined = undef
```

The `puppetlabs/stdlib` module provides a function you can use to see a variable's type:

```
include stdlib
$nametype = type_of( $my_name )      # String
$numtype  = type_of( $num_tokens )   # Integer
```

Defining Numbers

In Puppet 4, unquoted numerals are evaluated as a Numeric data type. Numbers are assigned specific numeric types based on the characters at the start of or within the number:

- Decimal numbers start with 1 through 9.
- Floating-point numbers contain a single period.
- Octal numbers (most commonly used for file modes) start with a `0`.
- Hexadecimal numbers (used for memory locations or colors) start with `0x`.

> **WARNING**
>
> In previous versions of Puppet, bare numbers were evaluated as strings. Best practice as of Puppet 3 was to quote all numbers to ensure they were evaluated as a `String`. This was preparation for Puppet 4, where unquoted numbers are the `Numeric` data type, and validation is performed on the value.

Any time an unquoted word starts with numbers, it will be validated as a Numeric type.

```
$decimal     = 1234      # valid Integer decimal assignment
$decimal     = 12.34     # valid Float decimal assignment
$octal       = 0775      # valid Integer octal assignment
$hexadecimal = 0xFFAA    # valid Integer hexadecimal assignment
$string      = '001234'  # string containing a number with leading zeros
```

Always quote numbers that need to be represented intact, such as decimals with leading zeros.

Creating Arrays and Hashes

It is possible to declare an `Array` (ordered list) that contains many values. As I'm sure you've used arrays in other languages, we'll jump straight to some examples:

```
$my_list  = [1,3,5,7,11]          # array of Numeric values
$my_names = ['Amy','Sam','Jen']   # array of String values
```

```
$mixed_up = ['Alice',3,true]      # String, Number, Boolean
$trailing = [4,5,6,]              # trailing commas OK, unlike JSON
$embedded = [4,5,['a','b']]       # number, number, array of strings
```

Array-to-array assignment works if there are equal variables and values:

```
[$first,$middle,$last] = ['Jo',undef,'Rhett']  # good
[$first,$middle,$last] = ['Jo','Rhett']        # error
```

Some functions require a list of input values, instead of an array. In a function call, the splat operator (`*`) operator will convert an `Array`into a comma-separated list of values:

```
myfunction( *$Array_of_arguments ) { ... }
```

This provides a concise, readable way to pass a list of unknown size to a function.

Mapping Hash Keys and Values

You can also create an unordered, random-access hash where member values are associated with a key value. At assignment time, the key and value should be separated by a *hash rocket* (`=>`), as shown here:

```
# small hash of user home directories
$homes = { 'Jack' => '/home/jack', 'Jill' => '/home/jill', }

# Multiline definition
$user = {
  'username' => 'Jill',  # String value
  'uid'      => 1001,    # Integer value
  'create'   => true,    # Boolean value
}
```

`Hash` keys must be `Scalar` (string or number), but values can be any data type. This means that the value assigned to a key could be a nested `Array` or `Hash`.

Using Variables in Strings

Strings with pure data should be surrounded by single quotes:

```
$my_name = 'Dr. Evil'
$how_much = '100 million'
```

Use double quotes when interpolating variables into strings, as shown here:

```
notice( "Hello ${username}, glad to see you today!" )
```

For very large blocks of text, you may want to use the *heredoc* multiline format. Start the block with an end tag surrounded by a `@( )` start tag:

```
$message_text = @(END)
This is a very long message,
which will be composed over
many lines of text.
END
```

By default, heredoc syntax is not interpolated, so this is the same as a single-quoted block of text. You can have the block be interpolated for variables by placing the end tag within double quotes. For example, the following contains variables to customize the message:

```
$message_text = @("END")
Dear ${user},
  Your password is ${password}.

  Please login at ${site_url} to continue.
END
```

Many common shell and programming escape sequences are available for you to use within interpolated strings:

Sequence	Expands to
`\n`	Line feed (end of line terminator)
`\r`	Carriage return (necessary in Windows files)
`\s`	Space
`\t`	Tab character

Using Braces to Limit Problems

As with most scripting languages, curly braces should be used to delineate variable boundaries:

```
$the_greeting = "Hello ${myname}, you have received ${num_tokens} tokens!"

notice( "The second value in the list is ${my_list[1]}" )
```

BEST PRACTICE

Use curly braces to delineate the beginning and end of a variable name within a string.

Use curly braces any time you use a variable within a string, but not when using the variable by itself. As shown here, the variables are used directly by the resource without interpolation, so it reads easier without braces:

```
# This time we define the strings in advance
$file_name = "/tmp/testfile2-${my_name}.txt"
$the_greeting = "Hello ${myname}, you have received ${num_tokens} tokens!"

# Don't use braces for variables that stand alone
file { $file_name:
  ensure  => present,
  mode    => '0644',
  replace => true,
  content => $the_greeting,
}
```

In particular, the following array index will not resolve correctly:

```
notice( "The second value in the list is $my_list[1]" )
```

This will actually output every value in the array, followed by the string `[1]`. To interpolate a specific array index or hash key within a string, you must enclose the array and index or hash and key both inside curly braces, like so:

```
# Output value from an array index 1
notice( "The second value in the list is ${my_list[1]}" )

# Output value stored in hash key alpha
notice( "The value stored with key alpha is ${my_hash['alpha']}" )
```

Preventing Interpolation

You will sometimes want to utilize characters that generally have special meaning within your strings, such as dollar signs and quotes. The simplest way is to place the entire string within single quotes, but at times you need to use the special characters in combination with interpolation.

In most cases, you can simply preface the character with the escape character (a backslash or `\`) to avoid interpolation. Interpolation happens only once, even if a string is used within another string:

```
# Work around the need for both types of quotes in a variable
$the_greeting = "we need 'single quotes' and \"double quotes\" here."

# Place backslashes before special characters to avoid interpolation
$describe = "\$user uses a \$ dollar sign and a \\ backslash"

# Previously interpolated values won't be interpolated again
$inform = "${describe}, and resolves to the value '${user}'."
```

Using single quotes avoids the need for backslashes:

```
$num_tokens = '$100 million'    # dollars, not a variable
$cifs_share = '\\server\drive'  # windows share, not escape chars
```

Using Unicode Characters

You can safely assign Unicode characters to strings utilizing `\u` followed by either their UTF-16 four-digit number or the UTF-8 hex value in curly braces. Here is a very small sample of Unicode characters and the way to represent them in Puppet:

Character	Description	UTF-16	UTF-8 hex
€	Euro currency	`\u20AC`	`\u{E282AC}`
¥	Yen currency	`\u00A5`	`\u{C2A5}`
Ä	Umlaut A	`\u00C4`	`\u{C384}`
©	Copyright sign	`\u00A9`	`\u{C2A9}`

Unicode is documented at length at *www.unicode.org*; however, the site doesn't have an easy search mechanism. You can find many UTF-16 Unicode numbers at Wikipedia's "List of Unicode Characters". Micha Köllerwirth maintains a more comprehensive list along with the UTF-8 equivalents, and you can also check out the character search available on FileFormat.info.

Avoiding Redefinition

Variables may not be redefined in Puppet within a given namespace or scope. We'll cover the intricacies of scope in Part II, "Creating Puppet Modules", but understand that a manifest has a single namespace, and a variable cannot receive a new value within that namespace.

This is one of the hardest things for experienced programmers to grow accustomed to. However, if you consider the nature of declarative programming, it makes a lot of sense.

In imperative programming, you have a specific order of events and an expected state of change as you pass through the code:

```
myvariable = 10
print myvariable  # prints 10
myvariable = 20
print myvariable  # prints 20
```

In a declarative language, the interpreter handles variable assignment independently of usage within resources. Which assignment would be performed prior to the resource application? This can change as more resources are added to manifests, and more manifests are applied to a node. Likewise, it would change for each node that did or did not have certain resources applied to it. This means that the value could change from one node to the other, or even from one evaluation to another on the same node.

To avoid this problem, Puppet kicks out an error if you attempt to change a variable's value:

```
[vagrant@client ~]$ cat double-assign.pp
$myvar = 5
$myvar = 10

[vagrant@client ~]$ puppet apply double-assign.pp
Error: Cannot reassign variable myvar at double-assign.pp:2
```

This is simply part of the learning process for thinking declaratively.

Avoiding Reserved Words

As mentioned previously, variables may be assigned string values as bare words, without quotes. A bare word that begins with a letter is usually evaluated the same as if it were single-quoted.

There are a number of reserved words that have special meaning for the interpreter, and must be quoted when used as string values. These are all fairly obvious words that are reserved within many other programming languages:

and	elsif	node
attr	false	private
case	function	or

`class`	`if`	`true`
`default`	`in`	`type`
`define`	`import`	`undef`
`else`	`inherits`	`unless`

There really aren't any surprises in this list. Any language primitive (as just shown), resource type (e.g., `file`, `exec`), or function name cannot be used as a bare word string.

You can find a complete list of all reserved words at "Language: Reserved Words and Acceptable Names" on the Puppet docs site.

> **TIP**
>
> Previously working unquoted strings can yield surprising results when functions or resource types with the same name are added to the catalog. Make code future-proof by quoting string values every time:
>
> ```
> $my_variable = somestring    # valid
> $my_variable = 'somestring'  # best practice
> ```

Learning More

You can find more details about using variables at the Puppet docs site:

- Language: Variables
- Language: About Values and Data Types

Some specific data types worth reading about in more detail on the Puppet docs site include the following:

- Data Types: Numbers
- Data Types: Arrays
- Data Types: Hashes

Finding Facts

Speaking of variables, Facter provides many variables for you containing node-specific information. These are always available for use in your manifests. Go ahead and run `facter` and look at the output:

```
[vagrant@client ~]$ facter
architecture => x86_64
augeasversion => 1.0.0
blockdevice_sda_model => VBOX HARDDISK
blockdevice_sda_size => 10632560640
blockdevice_sda_vendor => ATA
blockdevices => sda
domain => example.com
facterversion => 3.1.2
filesystems => ext4,iso9660
fqdn => client.example.com
gid => vagrant
hardwareisa => x86_64
hardwaremodel => x86_64
hostname => client
id => vagrant
interfaces => eth0,eth1,lo
ipaddress => 10.0.2.15
...etc
```

As you can see, Facter produces a significant number of useful facts about the system, from facts that won't change over the lifetime of a system (e.g., platform and architecture) to information that can change from moment to moment (e.g., free memory). Try the following commands to find some of the more variable fact information provided:

```
[vagrant@client ~]$ facter | grep version

[vagrant@client ~]$ facter | grep mb

[vagrant@client ~]$ facter | grep free
```

Puppet adds several facts for use within Puppet modules. You can run the following command to see all facts used by Puppet, and installed on the node from Puppet modules:

```
[vagrant@client ~]$ facter --puppet
```

Puppet always adds the following facts above and beyond system facts provided by Facter:

`$facts['clientcert']`
: The client-reported value of the node's `certname` configuration value.

`$facts['clientversion']`
: The client-reported version of Puppet running on the node.

`$facts['clientnoop']`
: The client-reported value of whether `noop` was enabled in the configuration or on the command line to perform the comparison without actually making changes.

`$facts['agent_specified_environment']`
: The environment requested by the client. When using a Puppet server, the server could override the environment selection. This value contains the requested value. If this value is blank, the *production* environment is used by default.

All of these facts can be found in the `$facts` hash.

> **NOTE**
>
> Puppet servers provide server-validated or *trusted facts* in a `$trusted` hash that is more reliable than the information provided by the client. You'll find information about trusted facts in Part III.

You can also use Puppet to list out Puppet facts in JSON format:

```
[vagrant@client ~]$ puppet facts find
$ puppet facts find
{
  "name": "client.example.com",
  "values": {
    "puppetversion": "4.4.0",
    "virtual": "virtualbox",
    "is_virtual": true,
    "architecture": "x86_64",
    "augeasversion": "1.4.0",
    "kernel": "Linux",
    "domain": "example.com",
    "hardwaremodel": "x86_64",
    "operatingsystem": "CentOS",
```

Facter can provide the data in different formats, useful for passing to other programs. The following options output Facter data in the common YAML and JSON formats:

```
[vagrant@client ~]$ facter --yaml
[vagrant@client ~]$ puppet facts --render-as yaml

[vagrant@client ~]$ facter --json
[vagrant@client ~]$ puppet facts --render-as json
```

Calling Functions in Manifests

A *function* is executable code that may accept input parameters, and may output a return value. A function that returns a value can be used to provide a value to a variable:

```
$zero_or_one = bool2num( $facts['is_virtual'] );
```

The function can also be used in place of a value, or interpolated into a string:

```
# md5() function provides the value for the message attribute
notify { 'md5_hash':
  message => md5( $facts['fqdn'] )
}

# Include the MD5 hash in the result string
$result = "The MD5 hash for the node name is ${md5( $facts['fqdn'] )}"
```

Functions can also take action without returning a value. Previous examples used the `notice()` function, which sends a message at `notice` log level but does not return a value. In fact, there is a function for logging a message at each level, including:

- `debug( message )`
- `info( message )`
- `notice( message )`
- `warning( message )`
- `err( message )`

Puppet executes functions when building the catalog; thus, functions can change the catalog. Some of the more common uses for this level of power are:

- Look up data from external sources
- Add, modify, or remove items from the catalog
- Dynamically generate and execute code segments

Functions can be written in the common prefix format or in the Ruby-style chained format. The following two calls will return the same result:

```
# Common prefix format
notice( 'this' )

# Ruby-style chained format
'this'.notice()
```

As always, use the form that is easier to read where it is used in the code.

Using Variables in Resources

Now let's cover how to use variables in resources. Each data type has different methods, and sometimes different rules, about how to access its values.

Constant strings without variables in them should be surrounded by single quotes. Strings containing variables to be interpolated should be surrounded by double quotes. No other type should be quoted. Here's an example:

```
notice( 'Beginning the program.' )
notice( "Hello, ${username}" )
notice( 1000000 )
notice( true )
```

You can access specific items within an Array by using a 0-based array index within square brackets. Two indices can be specified to retrieve a range of items:

```
$first_item = $my_list[1]
$four_items = $my_list[3,6]
```

You can access specific items within a Hash by using the hash key within square brackets as follows:

```
$username = $my_hash['username']
```

Use curly braces when interpolating variables into a double-quoted string. The curly braces must surround both the variable name and the index or key (within square brackets) when accessing hash keys or array indexes:

```
notice( "The user's name is ${username}" )
notice( "The second value in my list is ${my_list[1]}" )
notice( "The login username is ${my_hash['username']}" )
```

> **WARNING**
>
> Array variables and their index, and hash variables with a key, must be surrounded with curly braces for interpolation in strings.

Curly braces are only necessary when you're interpolating a variable within a string. Do not use braces or quotes when using the variable by itself. Here is an example of using predefined variables properly:

```
file { $filename:
  ensure  => present,
  mode    => '0644',
  replace => $replace_bool,
  content => $file['content'],
}
```

BEST PRACTICE

Don't surround standalone variables with curly braces or quotes.

Retrieve specific values from a `Hash` by assigning to an `Array` of variables named for the keys you'd like to retrieve. Read that sentence carefully—the name of the variable in the array identifies the hash key to get the value from:

```
[$Jack] = $homes           # identical to $Jack = $homes['Jack']
[$username,$uid] = $user   # gets the values assigned to keys "username" and "uid"
$Jill = $user              # oops, got the entire Hash!
```

The facts provided by Facter can be referenced like any other variable. The facts are available in a `$facts` hash. For example, to customize the message shown on login to each node, use a `file` resource like this:

```
file { '/etc/motd':
  ensure  => present,
  mode    => '0644',
  replace => true,
  content => "${facts['hostname']} runs ${facts['os']['release']['full']}",
}
```

Older Puppet manifests refer to facts using just the fact name as a variable, such as `$factname`. This is dangerous, as the fact could be overwritten either deliberately or accidentally within the scope in which the code is operating. A slight improvement is to refer to the fact explicitly in the top scope with `$::factname`. However, the variable could be overridden there as well. Finally, neither of these options informs a code reviewer whether the value was defined in a manifest, or by a fact.

> **BEST PRACTICE**
>
> Refer explicitly to a fact using the `$facts[]` hash. This ensures access to unaltered values supplied by Facter, and tells the reader where the value came from.

You can receive `Evaluation Error` exceptions when Puppet tries to use a variable that has never been defined by enabling the `strict_variables` configuration setting in */etc/puppetlabs/puppet/puppet.conf*:

```
[main]
  strict_variables = true
```

This will not cause an error when a variable has been explicitly set to `undef`. It will only throw an exception if the variable has never been declared:

```
$ puppet apply --strict_variables /vagrant/manifests/undefined.pp
Notice: Scope(Class[main]):
Error: Evaluation Error: Unknown variable: 'never_defined'.
  at /vagrant/manifests/undefined.pp:4:9 on node client.example.com
```

Defining Attributes with a Hash

It is also possible to pass a hash of attribute names and values in to a resource definition. You do this with an attribute name of `*` (called a *splat*) with a value of the hash.

Here's an example:

```
$resource_attributes = {
  ensure    => present,
  owner     => 'root',
  group     => 'root',
  'mode'    => '0644',
  'replace' => true,
}

file { '/etc/config/first.cfg':
  source => 'first.cfg',
  *      => $resource_attributes,
}

file { '/etc/config/second.cfg':
  source => 'config.cfg',
  *      => $resource_attributes,
}
```

The splat operator allows you to utilize a common set of values across many resources without repeating the same declaration. This is an essential strategy for *don't repeat yourself* (*DRY*) development.

Declaring Multiple Resource Titles

It is possible to declare multiple resources within a single declaration. The first way you can do so is by supplying an array of titles with the same resource body. The following example works because the file's name defaults to the title if not supplied as an attribute:

```
file { ['/tmp/file_one.txt','/tmp/file_two.txt']:
  ensure => present,
  owner  => 'vagrant',
}
```

This definition creates two different file resources in the same declaration by providing two different titles. This only works successfully when the title is reused as a parameter that makes the resource unique.

Declaring Multiple Resource Bodies

The title and attributes of a resource are called the *resource body*. A single resource declaration can have multiple resource bodies separated by semicolons. Here's an example of two `file` resources within a single resource declaration:

```
file {
  'file_one':
    ensure => present,
    owner  => 'vagrant',
    path   => 'file_one.txt',
  ;

  'file_two':
    ensure => present,
    owner  => 'vagrant',
    path   => 'file_two.txt',
  ;
}
```

> **TIP**
>
> It is common to use a different indentation style in multiple resource declaration to improve readability.

This format would create two resources, exactly as if these were done in two different resource declarations. As this is not necessarily easier to read, the single resource per declaration is considered a better practice, except when using default values.

If one of the resource bodies has the title `default`, it is not used to create a resource. Instead, it defines defaults for the other resource bodies. Here is an example where the preceding definition becomes shorter and easier to maintain:

```
file {
  default:
    ensure => present,
    owner  => 'vagrant',
  ;

  'file_one': path => 'file_one.txt';
  'file_two': path => 'file_two.txt';
}
```

Any of the resources can provide an override for an attribute, in which case the default value is ignored.

Modifying with Operators

You can use all of the standard arithmetic operators for variable assignment or evaluation. As before, we're going to provide examples and skip an explanation you've likely gotten many times in your life:

```
$added       = 10 + 5       # 15
$subtracted  = 10 - 5       # 5
$multiplied  = 10 * 5       # 50
$divided     = 10 / 5       # 2
$remainder   = 10 % 5       # 0
$two_bits_l  = 2 << 2       # 8
$two_bits_r  = 64 >> 2      # 16
```

> **TIP**
>
> If bit-shifting operators aren't something you're accustomed to, you can safely ignore them. You won't need the bit-shift operators unless you enjoy playing in binary.

We'll cover the comparison operators in the next section about conditionals.

Adding to Arrays and Hashes

New in Puppet 4, you can add items to arrays and hashes. You might remember these structured data types we defined in the previous section:

```
$my_list = [1,4,7]
$bigger_list = $my_list + [14,17]   # equals [1,4,7,14,17]

$key_pairs = {name => 'Joe', uid => 1001}
$user_definition = $key_pairs + { gid => 500 }  # hash now has name, uid, gid...
```

> **NOTE**
>
> The expression returns a new value from the operands; it does not change the existing data.

You can also append single values to arrays with the << operator. Watch out, as an array appended to an array creates a single entry in the array containing an array in the last position:

```
$longer_list = $my_list << 33      # equals [1,4,7,33]
$unintended = $my_list << [33,35]  # equals [1,4,7,[33,35]]
```

> **WARNING**
>
> Concatenation and append are new operators in Puppet 4; they are not available in earlier versions of Puppet.

Removing from Arrays and Hashes

New in Puppet 4, you can remove values from arrays and hashes with the removal operator (-). The following examples return a new array without the values on the righthand side:

```
# Remove a single value
$names = ['jill','james','sally','sam','tigger']
$no_tigger = $names - 'tigger'

# Remove multiple values
$no_boys = $names - ['james','sam']
```

> **NOTE**
>
> The expression returns a new value from the operands; it does not change the existing data.

Each of the following examples returns a hash without the keys listed on the right side:

```
# Remove a single key
$user = {name => 'Jo', uid => 1001, gid => 500 }
$no_name = $user - 'name'

# Remove multiple keys
$user = {name => 'Jo', uid => 1001, gid => 500 }
$only_name = $user - ['uid','gid']

# Remove all matching keys from another hash
$compare = {name => 'Jo', uid => 1001, home => '/home/jo' }
$difference = $user - $compare
```

> **WARNING**
>
> Removing values from arrays and hashes was only previously possible by using functions from the `stdlib` library.

Order of Operations

The operators have the precedence used by standard math and all other programming languages. If you find this statement vague, it is because I intended it to be. Very few people know all the rules for precedence.

Do yourself and whoever has to read your code a favor: use parentheses to make the ordering explicit. Explicit ordering is more readable and self-documenting:

```
# you don't need to know operator precedence to understand this
$myvar = 5 * (10 + $my_var)
```

If you are stuck reading code that was written by someone who didn't use parentheses, the implicit order of operations is documented at "Language: Expressions and Operators" on the Puppet docs site.

Using Comparison Operators

If you have any experience programming, you'll find Puppet's comparison operators familiar and easy to understand. Any expression using comparison operations will evaluate to boolean `true` or `false`. First, let's discuss all the ways to evaluate statements. Then we'll review how to use the boolean results.

Number comparisons operate much as you might expect:

```
4 != 4.1                  # number comparisons are simple equality match
$how_many_cups < 4        # any number less than 4.0 is true
$how_many_cups >= 3       # any number larger than or equal to 3.0 is true
```

String operators are a bit inconsistent. String equality comparisons are case insensitive, while substring matches are case sensitive:

```
 coffee  == 'coffee'     # bare word string is equivalent to quoted single word
'Coffee' == 'coffee'     # string comparisons are case insensitive

'tea' !in 'coffee'       # you can't find tea in coffee
'Fee' !in 'coffee'       # substring matches are case sensitive
'fee'  in 'coffee'       # you can pay your daily barista fee
```

Array and hash comparisons match only with complete equality of both length and value. The `in` comparison looks for value matches in arrays, and key matches in hashes:

```
[1,2,5] != [1,2]                    # array matching tests for identical arrays
5 in [1,2,5]                        # value found in array

{name => 'Joe'} != {name => 'Jo'}           # hashes aren't identical
'Jo' !in {fname => 'Jo', lname => 'Rhett'}  # Jo is a value and doesn't match
```

You can also compare values to data types, like so:

```
$not_true =~ Boolean        # true if true or false
$num_tokens =~ Integer      # true if an integer
$my_name !~ String          # true if not a string
```

As this feature has the most benefit for input validation in Puppet modules, we cover this topic extensively in "Validating Input with Data Types". For now, just be aware that it is possible.

When doing comparisons you'll find the standard boolean operators `and`, `or`, and `!` (not) work exactly as you might expect:

```
true and true               # true
true and false              # false
true or false               # true
true and !false             # true
true and !true              # false
```

> **TIP**
>
> These same operators (except `in`) can be used by the odd people who enjoy Backus–Naur form. Yes, you, we know about you. And no, I'm not going to initiate any innocents into your ranks. Enjoy your innocence if you don't know Backus–Naur. Just be aware that they work, should you need that particular perversion.

You can find a complete list of all operands and operators with example uses at "Language: Expressions and Operators" on the Puppet docs site.

Evaluating Conditional Expressions

Now, let's use these expressions you've learned with conditional statements. You have four different ways to utilize the boolean results of a comparison:

- `if/elsif/else` statements
- `unless/else` statements
- `case` statements
- Selectors

As you'd expect, there's always the basic conditional form I'm sure you know and love:

```
if ($coffee != 'drunk') {
  notify { 'best-to-avoid': }
}
elsif ('scotch' == 'drunk') {
  notify { 'party-time': }
}
else {
  notify { 'party-time': }
}
```

There's also `unless` to reverse comparisons for readability purposes:

```
unless $facts['kernel'] == Linux {
   notify { 'You are on an older machine.': }
}
else {
   notify { 'We got you covered.': }
}
```

> **WARNING**
>
> The use of `else` with `unless` is new to Puppet 4.

While `unless` is considered bad form by some, I recommend sticking with the most readable form. The following example shows why `unless` can be tricky reading with an `else` clause:

```
# The $id fact tells us who is running the Puppet agent
unless( $facts['id'] == 'root' ) {
  notify { 'needsroot':
    message => "This manifest must be executed as root.",
  }
}
else {
  notify { 'isroot':
    message => "Running as root.",
  }
}
```

The case operator can be used to do numerous evaluations, avoiding a long string of multiple `elsif`(s). You can test explicit values, match against another variable, use regular expressions, or evaluate the results of a function. The first successful match will execute the code within the block following a colon:

```
case $what_she_drank {
   'wine':            { include state::california }
   $stumptown:        { include state::portland   }
   /(scotch|whisky)/: { include state::scotland   }
   is_tea( $drink ):  { include state::england    }
   default:           {}
}
```

Always include a `default:` option when using `case` statements, even if the default does nothing, as shown in the preceding example.

Statements with `selectors` are similar to `case` statements, except they return a value instead of executing a block of code. This can be useful when you are defining variables. A selector looks like a normal assignment, but the value to be compared is followed by a question mark and a block of comparisons with fat commas identifying the matching values:

```
$native_of = $what_he_drinks ? {
   'wine'            => 'california',
   $stumptown        => 'portland',
   /(scotch|whisky)/ => 'scotland',
   is_tea( $drink )  => 'england',
   default           => 'unknown',
}
```

As a value must be returned in an assignment operation, a match is required. Always include a bare word `default` option with a value.

So the drinking comparisons have been fun, but let's examine some practical comparisons that you may actually use in a real manifest. Here's a long `if/then/else` chain:

```
# Explicit comparison
if( $facts['osfamily'] == 'redhat' ) {
   include yum
}
# Do a substring match
elsif( $facts['osfamily'] in 'debian-ubuntu' ) {
   include apt
}
# New package manager available with FreeBSD 9 and above
elsif( $facts['operatingsystem'] =~ /?i:freebsd/ )
  and ( $facts['os']['release']['major'] >= 9 ) {
   include pkgng
}
```

The same result can be had in a more compact `case` statement:

```
case $facts['osfamily'] {
  'redhat':                                          { include yum   }
  'debian', 'ubuntu':                                { include apt   }
  'freebsd' and ($facts['os']['release']['major'] >= 9) { include pkgng }
  default: {}
}
```

Selectors are also useful for handling heterogenous environments:

```
$libdir = $facts['osfamily'] ? {
  /(?i-mx:centos|fedora|redhat)/ => '/usr/libexec/mcollective',
  /(?i-mx:ubuntu|debian)/        => '/usr/share/mcollective/plugins',
  /(?i-mx:freebsd)/              => '/usr/local/share',
}
```

The splat operator (`*`) can turn an array of values into a list of choices. This can be very useful in `case` or `select` statements, as shown here:

```
$redhat_based = ['RedHat','Fedora','CentOS','Scientific','Oracle','Amazon']
$libdir = $facts['osfamily'] ? {
  *$redhat_based => '/usr/libexec/mcollective',
```

You can find a complete list of all conditional statements with more example uses at "Language: Conditional Statements and Expressions" on the Puppet docs site.

Matching Regular Expressions

Puppet supports standard Ruby regular expressions, as defined in the Ruby Regexp docs. The match operator (=~) requires a `String`value on the left, and a `Regexp` expression on the right:

```
$what_did_you_drink =~ /tea/        # likely true if English
$what_did_you_drink !~ /coffee/     # likely false if up late
$what_did_you_drink !~ "^coffee$"   # uses a string value for regexp
```

The value on the left must be a string. The value on the right can be a `/Regexp/` definition within slashes, or a string value within double quotes. The use of a string allows variable interpolation to be performed prior to conversion into a `Regexp`.

You can use regular expressions in four places:

- Conditional statements: `if` and `unless`
- `case` statements
- Selectors
- Node definitions (deprecated)

As regular expressions are well documented in numerous places, we won't spend time covering how to use them here, other than to provide some examples:

```
unless $facts['operatingsystem'] !~ /(?i-mx:centos|fedora|redhat)/ {
        include yum
}

case $facts['hostname'] {
  /^web\d/: { include role::webserver  }
  /^mail/ : { include role::mailserver }
  default : { include role::base       }
}

$package_name = $facts['operatingsystem'] ? {
  /(?i-mx:centos|fedora|redhat)/ => 'mcollective',
  /(?i-mx:ubuntu|debian)/        => 'mcollective',
  /(?i-mx:freebsd)/              => 'sysutils/mcollective',
}
```

You may find Tony Stubblebine's *Regular Expressions Pocket Reference* (O'Reilly) handy for day-to-day work with `Regexps`. To truly master regular expressions, there is no better book than, well, Jeffrey E.F. Friedl's *Mastering Regular Expressions* (O'Reilly).

Building Lambda Blocks

A *lambda* is a block of code that allows parameters to be passed in. You can think of them as functions without a name. You will use lambdas with the iterator functions (such as `each()`, introduced in the next

section) to perform a set of instructions on multiple values. If you are experienced with Ruby lambdas, you'll find the syntax similar.

> **WARNING**
>
> Lambdas are a new, advanced feature of Puppet 4 not available in any previous version of Puppet.

A lambda begins with one or more variable names between pipe operators | |. These name the variables that will contain the values passed into the block of code:

```
| $firstvalue, $secondvalue | {
  block of code that operates on these values.
}
```

The lambda has its own variable scope. This means that the variables named between the pipes exist only within the block of code. You can name these variables any name you want, as they will be filled by the values passed by the function into the lambda on each iteration. Other variables within the context of the lambda are also available, such as local variables or node facts.

The following example will output a list of disk partitions from the hash provided by Facter. Within the loop, we refer to the `hostname`fact on each iteration. The device name and a hash of values about each device are stored in the `$name` and `$device` variables during each loop:

```
$ cat /vagrant/manifests/mountpoints.pp
each( $facts['partitions'] ) |$name, $device| {
  notice( "${facts['hostname']} has device ${name} with size ${device['size']}" )
}

$ puppet apply /vagrant/manifests/mountpoints.pp
Notice: Scope(Class[main]): Host geode has device sda1 with size 524288
Notice: Scope(Class[main]): Host geode has device sda2 with size 3906502656
Notice: Scope(Class[main]): Host geode has device sdb1 with size 524288
Notice: Scope(Class[main]): Host geode has device sdb2 with size 3906502656
```

> **NOTE**
>
> As shown here, you must enclose both the variable name and the array or hash index (in square brackets) within curly braces to ensure they are interpolated together. Otherwise, the output will contain the entire array or hash, followed by the index as literal text in square brackets.

Next, we'll cover each of the functions that can iterate over values and pass them to a lambda.

Looping Through Iterations

In this section, we're going to introduce powerful new functions for iterating over sets of data. You can use iteration to evaluate many items within an array or hash of data using a single block of code (a lambda, described on the previous page).

> **WARNING**
>
> Iterations are a new feature of Puppet 4.

Here are some practical examples available to you from the basic facts provided by Facter (you can use iteration with any data point that can be presented as an array or a hash):

- Going through all IP addresses assigned to a node to see if any meet a specific condition.
- Going through all users on a node to find ones that match a certain criteria.
- Going through all mounted partitions to determine the total space available to the node.

There are five functions that iterate over a set of values and pass each one to a lambda for processing. The lambda will process each input and return a single response containing the processed values. Here are the five functions, what they do to provide input to the lambda, and what they expect the lambda to return as a response:

- `each()` acts on each entry in an array, or each key/value pair in a hash.
- `filter()` returns a subset of the array or hash that were matched by the lambda.
- `map()` returns a new array or hash from the results of the lambda.
- `reduce()` combines array or hash values using code in the lambda.
- `slice()` creates small chunks of an array or hash and passes it to the lambda.

The following examples show how these functions can be invoked. They can be invoked in traditional prefix style:

```
each( $facts['partitions'] ) |$name, $device| {
  notice( "${facts['hostname']} has device $name with size ${device['size']}" )
}
```

Or you can chain function calls to the values they operate on, which is a common usage within Ruby:

```
$facts['partitions'].each() |$name, $device| {
  notice( "${facts['hostname']} has device $name with size ${device['size']}" )
}
```

Finally, new to Puppet 4 is the ability to use a hash or array literal instead of a variable. The following example demonstrates iteration over a literal array of names:

```
['sally','joe','nancy','kevin'].each() | $name | {
  notice( "$name wants to learn more about Puppet." )
}
```

Now let's review each of the functions that can utilize a lambda.

each()

The `each()` function invokes a lambda once for each entry in an array, or each key/value pair in a hash. The lambda can do anything with the input value, as no response is expected. `each()` is most commonly used to process a list of items:

```
# Output a list of interfaces that have IPs
split( $facts['interfaces'], ',' ).each |$interface| {
  if( $facts["ipaddress_${interface}"] != '' ) {
    notice( sprintf( "Interface %s has IPv4 address %s",
      $interface, $facts["ipaddress_${interface}"] )
    )
  }
  if( $facts["ipaddress6_${interface}"] != '' ) {
    notice( sprintf( "Interface %s has IPv6 address %s",
      $interface, $facts["ipaddress6_${interface}"] )
    )
  }
}
```

When you apply this manifest, you'll get a list of interfaces and each IP they have:

```
[vagrant@client puppet]$ puppet apply /vagrant/manifests/interface_ips.pp
Notice: Scope(Class[main]): Interface enp0s3 has IPv4 address 10.0.2.15
Notice: Scope(Class[main]): Interface enp0s3 has IPv6 address fe80::a0:27:feb:2b2
Notice: Scope(Class[main]): Interface enp0s8 has IPv4 address 192.168.250.10
Notice: Scope(Class[main]): Interface enp0s8 has IPv6 address fe80::a0:27:fec:d78
Notice: Scope(Class[main]): Interface lo has IPv4 address 127.0.0.1
Notice: Scope(Class[main]): Interface lo has IPv6 address ::1
```

If you want a counter for the values, providing an array with two entries gives you an index on the first one. For example, creating a list of all the interfaces on a system thats hosts virtualization clients yields the following:

```
$ cat /vagrant/manifests/interfaces.pp
split( $facts['interfaces'], ',' ).each |$index, $interface| {
  notice( "Interface #${index} is ${interface}" )
}

$ puppet apply /vagrant/manifests/interfaces.pp
Notice: Scope(Class[main]): Interface #0 is enp0s3
Notice: Scope(Class[main]): Interface #1 is enp0s8
Notice: Scope(Class[main]): Interface #2 is lo
```

> **NOTE**
>
> Be aware that the index is placed in the first variable, and the array entry in the second. This is the opposite of the same concept used in Ruby and ERB templates provided by `each_with_index`, where the array entry comes in the first variable and the index in the second.

You can also use `each()` on hashes. If you provide a single variable you'll get an array with two entries. If you provide two variables you'll have the key in the first one, and the value in the second one:

```
$ cat /vagrant/manifests/uptime.pp
each( $facts['system_uptime'] ) |$type, $value| {
  notice( "System has been up ${value} ${type}" )
}

$ puppet apply /vagrant/manifests/uptime.pp
Notice: Scope(Class[main]): System has been up 23:04 hours uptime
Notice: Scope(Class[main]): System has been up 83044 seconds
Notice: Scope(Class[main]): System has been up 23 hours
Notice: Scope(Class[main]): System has been up 0 days
```

The following manifest would provide the exact same results:

```
each( $facts['system_uptime'] ) |$uptime| {
   notice( "System has been up $uptime[1] $uptime[0]" )
}
```

The `each()` function returns the result of the last operation performed. In most cases, you'll use it to process each entry and you won't care about the return; however, the result could be useful if the value of the last entry has some meaning for you. While you might consider calculating an aggregate value from the operations, that is exactly what the `reduce()` function is for.

REVERSE_EACH()

The `reverse_each()` function (new in Puppet 4.4) invokes a lambda once for each entry in an array or hash in reverse order. It is otherwise identical to `each()`.

STEP()

The `step(N)` function (new in Puppet 4.4) invokes a lambda once for each *N*th entry after the first one in an array or hash. It is otherwise identical to `each()`.

filter()

The `filter()` function returns a filtered subset of an array or hash containing only entries that were matched by the lambda. The lambda block evaluates each entry and returns a positive result if the item matches.

For an extended example, let's examine all interfaces and find all RFC1918 IPv4 and RFC4291 IPv6 internal-only addresses. We do this with multiple steps:

1. Filter all facts to find those containing IP addresses.
2. Filter the first results to return interfaces with RFC1918 or IPv6 link-local addresses.
3. Iterate over the second results to extract the interface name for each address.

   Here's a sample code block that does this:

```
$ips = $facts.filter |$key,$value| {
  $key =~ /^ipaddress6?_/
}
$private_ips = $ips.filter |$interface, $address| {
  $address =~ /^(10|172\.(?:1[6-9]|2[0-9]|3[0-1])|192\.168)\./
    or $address =~ /^fe80::/
}
$private_ips.each |$ip_interface,$address| {
  $interface = regsubst( $ip_interface, '^ipaddress6?_(\w+)', '\1' )
  notice( "interface $interface has private IP $address" )
}
```

If you apply this on a node, you'll see results like this:

```
$ puppet apply /vagrant/manifests/ipaddresses.pp
Notice: Scope(Class[main]): interface enp0s3 has private IP fe80::a0:27:feb:2b28
Notice: Scope(Class[main]): interface enp0s8 has private IP fe80::a0:27:fec:d78c
Notice: Scope(Class[main]): interface enp0s3 has private IP 10.0.2.15
Notice: Scope(Class[main]): interface enp0s8 has private IP 192.168.250.10
```

map()

The `map()` method returns an `Array` from the results of the lambda. You call `map()` on an array or hash, and it returns a new array containing the results. The lambda's final statement should result in a value that will be added to the array of results.

Here's an example where we create an array of IPv4 addresses. We pass in an array of interface names. We use the interface name to look for an IP address associated with that interface name in the `ipaddress_` facts.

As with `filter()`, when you pass in an array, the named variable contains the array value:

```
$ips = split( $facts['interfaces'], ',' ).map |$interface| {
  $facts["ipaddress_${interface}"]
}
```

If you pass in a hash, the named variable will contain an array with the key in the first position and the value in the second. The following example uses `filter()` to create a hash of interfaces that have IP addresses. Then it uses `map()` to create separate arrays of the interfaces, and the IPs:

```
$ints_with_ips = $facts.filter |$key,$value| {
  $key =~ /^ipaddress_/
}

# Create an array of ints with IPv4 addresses
$ints = $ints_with_ips.map |$intip| {
  $intip[0] # key
}

# Create an array of IPv4 addresses
$ips  = $ints_with_ips.map |$intip| {
  $intip[1] # value
}
```

reduce()

The `reduce()` function processes an array or hash and returns only a single value. It takes two arguments: an array or hash and an initial seed value. If the initial seed value is not supplied, it will use the first entry in the array or hash as the initial seed value. The lambda should be written to perform aggregation, addition, or some other function that will operate on many values and return a single value.

> **WARNING**
>
> The way `reduce()` utilizes the first entry in the array could have unintended consequences if the entry is not the appropriate data type for the output. As this is a common source of confusion, we'll show you an example of this problem.

In the following example, we pass the hash of partitions in to add together all of their sizes. As with all other functions, each hash entry is passed in as a small array of `[key,value]`:

```
$ cat /vagrant/manifests/partitions.pp
$total_disk_space = $facts['partitions'].reduce |$total, $partition| {
   notice( "partition $partition[0] is size $partition[1]['size']" )
   $total + $partition[1]['size']
}
notice( "Total disk space = ${total_disk_space}" )

$ puppet apply /vagrant/manifests/partitions.pp
Notice: Scope(Class[main]): partition sdb2 is size 3906502656
Notice: Scope(Class[main]): partition sda1 is size 524288
Notice: Scope(Class[main]): partition sda2 is size 3906502656
Notice: Scope(Class[main]): Total disk space = 7814053888
Total disk space = [sdb1, {filesystem => linux_raid_member, size => 524288},
  3906502656, 524288, 3906502656]
```

As we didn't supply an initial value, the first entry of the array contains the hash of values for the first partition (`sdb1`). It then added the remaining partition sizes to the array.

To resolve this situation, you should seed the initial value with the appropriate data type (integer `0`, in this case). The first hash entry is then processed by the block, adding the integer size to the seed value and creating the output we were looking for:

```
$ cat /vagrant/manifests/partitions.pp
$total_disk_space = $facts['partitions'].reduce(0) |$total, $partition| {
   notice( "partition $partition[0] is size $partition[1]['size']" )
   $total + $partition[1]['size']
}
notice( "Total disk space = ${total_disk_space}" )

$ puppet apply /vagrant/manifests/partitions.pp
Notice: Scope(Class[main]): partition sdb1 is size  524288
Notice: Scope(Class[main]): partition sdb2 is size 3906502656
Notice: Scope(Class[main]): partition sda1 is size 524288
Notice: Scope(Class[main]): partition sda2 is size 3906502656
Notice: Scope(Class[main]): Total disk space = 7814053888
```

slice()

The `slice()` function creates small chunks of a specified size from an array or hash. Of the available functions, this is perhaps one of the subtlest and trickiest to use, as the output changes depending on how you invoke it.

If you invoke `slice()` with a single parameter specified between the pipe operators, the value passed into the lambda will be an array containing the number of items specified by the slice size. The following example should make this clear:

```
[vagrant@client ~]$ cat /vagrant/manifests/slices.pp
[1,2,3,4,5,6].slice(2) |$item| {
   notice( "\$item[0] = ${item[0]}" )
   notice( "\$item[1] = ${item[1]}" )
}

[vagrant@client ~]$ puppet apply /vagrant/manifests/slices.pp
Notice: Scope(Class[main]): $item[0] = 1
Notice: Scope(Class[main]): $item[1] = 2
Notice: Scope(Class[main]): $item[0] = 3
Notice: Scope(Class[main]): $item[1] = 4
Notice: Scope(Class[main]): $item[0] = 5
Notice: Scope(Class[main]): $item[1] = 6
```

If you invoke `slice()` with the same number of parameters as the slice size, each variable will contain one entry from the slice. The following manifest would return exactly the same results as the previous example:

```
[vagrant@client ~]$ cat /vagrant/manifests/slices.pp
[1,2,3,4,5,6].slice(2) |$one, $two| {
   notice( "\$one == ${one}" )
   notice( "\$two == ${two}" )
}

[vagrant@client ~]$ puppet apply /vagrant/manifests/slices.pp
Notice: Scope(Class[main]): $one == 1
Notice: Scope(Class[main]): $two == 2
Notice: Scope(Class[main]): $one == 3
Notice: Scope(Class[main]): $two == 4
Notice: Scope(Class[main]): $one == 5
Notice: Scope(Class[main]): $two == 6
```

Unlike the other functions, hash entries are always passed in as a small array of `[key,value]`, no matter how many parameters you use. So if you have a slice of size `2` from a hash, the lambda will receive two arrays, each containing two values: the key and the value from the hash entry. Here's an example that demonstrates the idea.

```
$facts['partitions'].slice(2) |$part1, $part2| {
   notice( "partition names in this slice are $part1[0] and $part2[0]" )
}
```

Similar to `each()`, most invocations of slice do not return a value and thus the result can be ignored.

with()

The `with()` function invokes a lambda exactly one time, passing the variables provided as parameters. The lambda can do anything with the input values, as no response is expected.

You might point out that this function doesn't iterate and thus doesn't belong in this section of the book. You're quite right, but I've included it here because it behaves exactly like these other iterators, and can be very useful for testing:

```
with( 'austin', 'powers', 'secret agent' ) |$first,$last,$title| {
  notice( "A person named ${first} ${last}, ${title} is here to see you." )
}
```

The `with()` function is most commonly used to isolate variables to a private scope, unavailable in the main scope's namespace.

Capturing Extra Parameters

Most of the functions only produce one or two parameters for input to a lambda; however, `slice()` and `with()` can send an arbitrary number of parameters to a lambda. For ease of definition, you can prefix the final parameter with the splat operator (`*`) to indicate that it will accept all remaining arguments (called *captures-rest*). Even if only one value is supplied, the final item's data type will be `Array`.

> **TIP**
>
> The splat operator will also turn an array into a list of comma-separated values when necessary.

In the following example, we'll use the splat operator to transform the input array into comma-separated values for `with()`, and then use splat again to catch all remaining input values for the lambda. This example parses lines from */etc/hosts* and returns an single entry with an array of aliases:

```
# hosts line example:
#   192.168.250.6    puppetserver.example.com puppet.example.com puppetserver
$host = $hosts_line.split(' ')
with( $host* ) |$ipaddr, $hostname, *$aliases| {
```

```
  notice( "Host ${hostname} has IP ${ipaddr} and aliases ${aliases}" )
}
```

Test it out by running `puppet apply /vagrant/manifests/hostsfile_lines.pp`.

Iteration Wrap-Up

As you have seen in this section, the functions that iterate over arrays and hashes provide a tremendous amount of power not available in any previous version of Puppet.

You can invoke these functions like traditional functions or by chaining the functions to the data they are processing.

You can find more information about iteration and lambdas at "Language: Iteration and Loops" on the Puppet docs site.

Reviewing Puppet Configuration Language

This chapter introduced many of the components of the Puppet configuration language. Specifically, you learned the following:

- Variables provide named storage of data.
- All data has a type: `String`, `Numeric`, `Boolean`, `Array`, `Hash`, and others.
- Operators add, subtract, multiply, divide, concatenate, and merge values.
- Conditional operators compare data for equality and inclusion.
- Regular expressions match ranges of values or substrings within a value.
- Conditional expressions such as `if` and `unless` allow you to limit application.
- `case` and `select` evaluate a value to select an action or result.
- Lambdas are unnamed functions intended to process values passed in by iteration functions.
- Iterations pass each entry in an array or hash to a lambda block for evaluation.

These are the data types and functions available for evaluating and operating on data for use in Resource definitions within a Puppet manifest.