14

Testing

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Understanding basic tests
- Learning the `Test::More` module in depth
- Using different testing modules
- Understanding xUnit style testing

WROX.COM CODE DOWNLOAD FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `http://www.wrox.com/remtitle.cgi?isbn=1118013847` on the Download Code tab. The code for this chapter is divided into the following major examples:

- `lib/TestMe.pm`
- `t/testit.t.pm`
- `lib/TestQuery.pm`
- `t/query.t`
- `lib/Person.pm`
- `t/test_classes.t`
- `t/test_classes.t`
- `t/lib/TestsFor/Person.pm`
- `lib/Customer.pm`
- `t/lib/TestsFor/Customer.pm`
- `lib/TV/Episode.pm`

The author recently downloaded and built Perl version 5.15.9. All 521,047 tests passed. That's right: Perl ships with more than a half-million tests. Not all of them are perfect, and sometimes they verify that a feature works the way it was written and not the way that was intended, but it's still an incredible number. Few programming languages ship with more than a half-million tests. Testing your software is easy, and the Perl community is keen on testing. A few years ago, some people still argued against testing, but today professional Perl developers test more often than not.

BASIC TESTS

Consider the following line of code:

```
sub reciprocal { return 1 / shift }
```

Because the reciprocal of a number is 1 divided by that number, this looks like the canonical definition of a reciprocal. What could possibly go wrong?

When developers write tests, they often write a test to verify that the code does what it is supposed to do when the input is correct, but they don't think about incorrect input. So what happens if you pass `reciprocal()` a string? Or a hash reference? Or a zero? There are a variety of subtleties here that you may or may not care about, so let's delve into them.

Using Test::More

The standard testing module for Perl is `Test::More`, first released in Perl version 5.6.2. `Test::More` exports many helpful functions into your namespace, enables you to set the test plan (the number of tests), and makes writing tests simple. A basic test script might look like this:

```
use strict;
use warnings;
use Test::More tests => 3;
ok 7, '7 should evaluate to  a true value';
is 5 - 2, 3, '... and 5 - 2 should equal three';
ok 0, 'This test is designed to fail';
```

The `tests => 3` arguments to `Test::More` is called the plan. In this case, it states that you will run three tests. If you run fewer tests or more tests than the number of tests planned, even if all of them pass, the test script reports a failure.

The `ok()` and `is()` functions are each considered a single test. Thus, you have three tests in the preceding test script (two `ok()` and one `is()`), which are explained in a bit.

If you run this snippet of code, you should see the following output:

```
1..3
ok 1 - 7 should evaluate to  a true value
ok 2 - ... and 5 - 2 should equal three
not ok 3 - This test is designed to fail
#   Failed test 'This test is designed to fail'
#   at some_test_script.t line 7.
# Looks like you failed 1 test of 3.
```

> ***NOTE*** *The test output format is the* Test Anything Protocol*, also known as TAP. It's designed to be both readable by both humans and machines. If you're curious about this protocol, see* `http://www.testanything.org/`.

The `1..3` bit in the output is from the plan you asserted in the code. It means "we'll be running three tests."

Each line of test output is in this form:

```
ok $test_number - $test_message
```

Or:

```
not ok $test_number - $test_message;
```

Any line beginning with a `#` is called a diagnostic and should be a human readable message to help you figure out what your tests are doing. It is ignored for purposes of considering whether tests passed or failed.

> ***WARNING*** *The message for each test is optional. You could have written this:*
>
> ```
> ok 7;
> is 5 - 2, 3;
> ok 0;
> ```
>
> *However, if you do that, your test output looks like this:*
>
> ```
> 1..3
> ok 1
> ok 2
> not ok 3
> #   Failed test  at some_test_script.t line 7.
> # Looks like you failed 1 test of 3.
> ```
>
> *Needless to say, this makes it much harder to read the test output and understand what is going on. Always provide a clear test message for each test.*

As you write tests, it can be annoying to constantly update the test plan every time you add or delete tests, so you can state `no_plan` as the plan. (Note that we're adding parentheses to these test functions just to remind you that you can use them if you prefer them.)

```
use strict;
use warnings;
use Test::More 'no_plan';
```

```
ok( 7, '7 should evaluate to  a true value' );
is( 5 - 2, 3, '... and 5 - 2 should equal three' );
ok( 0, 'This test is designed to fail' );
```

When you run this, you see the following:

```
ok 1 - 7 should evaluate to  a true value
ok 2 - ... and 5 - 2 should equal three
not ok 3 - This test is designed to fail
#   Failed test 'This test is designed to fail'
#   at /var/tmp/eval_3qMq.pl line 7.
1..3
# Looks like you failed 1 test of 3.
```

The plan, `1..3`, now appears at the end of the test output. If you like numeric plans, then you should switch `no_plan` to `tests => $number_of_tests` when you finish writing your test code.

As a recommended alternative, if you use `Test::More` version 0.88 or better, you can do this:

```
use Test::More;
# write a bunch of tests here
done_testing;
```

With `done_testing()` at the end of your test script, if the program exits prior to calling `done_testing()`, you get an error, even if all tests pass. This lets you know whether your tests have completed successfully. If you run the following script with a new enough version of `Test::More`:

```
use strict;
use warnings;
use Test::More;
ok( 7, '7 should evaluate to  a true value' );
is( 5 - 2, 3, '... and 5 - 2 should equal three' );
exit;   # oops!
ok( 0, 'This test is designed to fail' );
done_testing;
```

You should get the following output:

```
ok 1 - 7 should evaluate to  a true value
ok 2 - ... and 5 - 2 should equal three
# Tests were run but no plan was declared and done_testing() was not seen.
```

You are recommended to install the latest version of `Test::More` from the CPAN. It's part of the `Test::Simple` distribution.

Writing Your Tests

As you work your way through this chapter, you start with a simple package named `TestMe`. You test it with a test script named `testit.t`. By convention, test scripts in Perl end with a `.t` extension.

The `TestMe` package will, of course, be in `lib/TestMe.pm`. Test scripts, however, usually live in the `t/` directory. Your directory structure should look like this:

```
lib/
|--TestMe.pm
t/
|--testit.t
```

Start by creating `lib/TestMe.pm` like this (code file `lib/Test.pm`):

```
package TestMe;

use strict;
use warnings;

use Exporter::NoWork;

sub reciprocal { return 1 / shift }

1;
```

The `Exporter::NoWork` module is similar to `Exporter`, but it automatically allows exporting of functions in your package that do not start with an underscore. (You need to install it from the CPAN.) You don't always want all functions exportable, but for these testing examples, it's perfect for your needs.

And code file `t/testit.t` should look like this:

```
use strict;
use warnings;
use Test::More;

use lib 'lib';
# Exporter::NoWork used the :ALL tag to import all functions
use TestMe ':ALL';

ok 1, 'this is a test!';

done_testing;
```

You add to both the `TestMe` package and the `testit.t` script as you work through the chapter.

Understanding the prove Utility

The `Test::More` distribution is used to test your modules. (Actually, it's a convenient wrapper around the `Test::Builder` module.) `Test::More` generates TAP output. However, something called the `Test::Harness` can run the test programs, read the TAP output, and determine if the tests pass or fail. Many large software systems can have tens of thousands of tests spread over hundreds of files. You don't want to run all those by hand, but you can use tools shipped with `Test::Harness` to do this.

> ***NOTE*** *Your author, in addition to writing the book, is also the author of the* `Test::Harness` *module that ships with the Perl language. (It's maintained by his good friend Andy Armstrong.) There might be truth in the rumor that your author is a testing bigot.*

One of the most useful utilities included with `Test::Harness` is the `prove` utility. From now on, you use this to run your test scripts:

```
$ prove -l -v t/testit.t
t/testit.t ..
ok 1 - this is a test!
1..1
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
Result: PASS
```

The `-l` switch to `prove` says "our code is in the `lib/` directory" (this is redundant in this case because you have `use lib 'lib'` in your test script) and the `-v` tells `prove` to use verbose output. Note that the bottom line says `Result: PASS`. If you have many tests, you can just glance at that output at the bottom of the test to see if it passed or failed.

Without the `-v` switch, you would see this:

```
t/testit.t .. ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
Result: PASS
```

For a single test script, you probably want to run it verbosely, but for multiple test scripts, leaving the `-v` switch off of the `prove` command can make the output easier to read:

```
$ prove -l t
t/01get_tag.t ...... ok
t/01get_token.t .... ok
t/02munge_html.t ... ok
t/03constructor.t .. ok
t/04internals.t .... ok
t/pod-coverage.t ... ok
t/pod.t ............ ok
All tests successful.
Files=7, Tests=188,  0 wallclock secs
Result: PASS
```

UNDERSTANDING TEST::MORE TEST FUNCTIONS

Now that you covered some of the basics of how testing works, it's time to write some tests. There are many testing modules out there, but the section goes through some of the more popular testing functions from the `Test::More` module.

Using ok

The `ok()` function is the most basic test function in Perl. The `Test::More` module automatically exports `ok()` and its syntax looks like this:

```
ok $true_or_false_expression, $optional_message;
```

The `ok()` function takes a single argument or expression and tests it for "truth." The test passes if the first argument is true:

```
ok 1;
ok "Hi there. I'm true!", "This is a silly test";
```

And it fails if the first argument is false:

```
my $balrog;
ok $balrog, 'You shall not pass!';
```

In your `t/testit.t` script, add the following line, replacing the useless `ok 1` test:

```
ok reciprocal(2), 'The reciprocal of 2 should be true';
```

If you run that script with `prove -lv t/testit.t`, you should see the following output:

```
t/testit.t ..
ok 1 - The reciprocal of 2 should be true
1..1
ok
All tests successful.
Files=1, Tests=1,  0 wallclock secs
Result: PASS
```

Congratulations! You've written your first successful test.

Using is

Obviously, checking that `reciprocal(2)` returns a true value is not useful. You want to ensure that it returns the correct value. You can do this with the `is()` function. Its syntax looks like this:

```
is $have, $want, $message;
```

The `$have` variable contains the value that you have, and `$want` is the value you expect to receive. Instead of replacing your `ok` test, just add another test after it:

```
is reciprocal(2), .5, 'The reciprocal of 2 should be correct';
```

Using the `prove` utility should now generate the following output:

```
t/testit.t ..
ok 1 - The reciprocal of 2 should be true
ok 2 - The reciprocal of 2 should be correct
```

```
1..2
ok
All tests successful.
Files=1, Tests=2,  0 wallclock secs
Result: PASS
```

Now write another test, but this time force it to fail:

```
is reciprocal(3), .5, 'The reciprocal of 3 should be correct';
```

If you add that after your first two tests, your test script should now look like this:

```
use strict;
use warnings;
use Test::More;
use lib 'lib';
use TestMe ':ALL';
ok reciprocal(2), 'The reciprocal of 2 should be true';
is reciprocal(2), .5, 'The reciprocal of 2 should be correct';
is reciprocal(3), .5, 'The reciprocal of 3 should be correct';
done_testing;
```

And running that produces:

```
t/testit.t ..
ok 1 - The reciprocal of 2 should be true
ok 2 - The reciprocal of 2 should be correct
not ok 3 - The reciprocal of 3 should be correct
1..3
#   Failed test 'The reciprocal of 3 should be correct'
#   at t/testit.t line 9.
#          got: '0.333333333333333'
#     expected: '0.5'
# Looks like you failed 1 test of 3.
Dubious, test returned 1 (wstat 256, 0x100)
Failed 1/3 subtests
Test Summary Report
-------------------
t/testit.t (Wstat: 256 Tests: 3 Failed: 1)
  Failed test:  3
  Non-zero exit status: 1
Files=1, Tests=3,  1 wallclock secs
Result: FAIL
shell returned 1
```

> ***NOTE*** *The diagnostics in the failure output may or may not appear immediately after the test failure. This is because the normal test output is sent to the computer's standard out (*`STDOUT`*), whereas your diagnostics are sent to standard error (*`STDERR`*). Because of how most operating systems work, those are not guaranteed to be in synch.*

If you want those in synch, you can pass the `--merge` *switch to* `prove`*:*

```
prove -v --merge t/testit.t
```

That attempts to "merge" the `STDOUT` *and* `STDERR` *output streams to make everything come out at the same time. It should be considered an experimental feature, and if something else in your software prints something to* `STDERR` *(or* `STDOUT`*, for that matter), the test harness may have trouble interpreting the output.*

So you focus on the test failure:

```
#   Failed test 'The reciprocal of 3 should be correct'
#   at t/testit.t line 9.
#          got: '0.333333333333333'
#     expected: '0.5'
# Looks like you failed 1 test of 3.
```

The failure output tells you what you `$have` (the `got:` line) and what you `$want` (the `expected:` line). This makes it easier to diagnose problems.

In this case, you might find it annoying (not to mention error-prone) to type `0.333333333333333` for the `$want` variable. That's when choosing an appropriate level of rounding and using the `sprintf` function can help. You can use this to easily make the test pass:

```
use strict;
use warnings;
use Test::More;
use lib 'lib';
use TestMe ':ALL';
ok reciprocal(2), 'The reciprocal of 2 should be true';
is reciprocal(2), .5, 'The reciprocal of 2 should be correct';
is sprintf( "%.4f", reciprocal(3) ), .3333,
  'The reciprocal of 3 should be correct';
done_testing;
```

NOTE *The* `sprintf()` *trick is also useful when you have floating point issues. For example, the following code, using the* `printf` *analogue to* `sprintf` *prints* `0.41999999999999998446` *on my computer. (The answer might be different on yours.)*

```
printf "%.20f", .42;
```

continues

continued

This is due to how computers handle numbers internally. Using `sprintf()` *to guarantee a certain level of precision can make it easier to test when one floating point number is equal to another.*

```
printf "%.5f", .42;
```

And that prints:

```
0.42000
```

Using like

Sometimes you know roughly what your data looks like, but you can't guarantee its exact output. That's where the `like()` test function comes into play. It looks like this:

```
like $data, qr/$regular_expression/, $message;
```

In your `TestMe` package, add `use DateTime;` to the top of the module, and add the following function:

```
sub order_date {
    my $today = DateTime->now->subtract( days => rand(30) );
    return join '/' => map { $today->$_ } qw/month day year/;
}
```

Use this to simulate a random order date.

The idea is that you want to output dates like `08/04/2012` for April 4th, 2012. Write a `like()` test for that, adding it to `testit.t` as usual.

```
# we use the /x modifier on the regex to allow whitespace to be
# ignored. This makes the regex easier to read
like today(), qr{^ \d\d/\d\d/\d\d\d\d $}x,
   'today() should return a DD/MM/YYYY format';
```

Run your `testit.t` script a few times. Because you're using a random value, sometimes it might pass, but other times it will fail with something like this:

```
not ok 4 - today() should return a DD/MM/YYYY format
1..4
#   Failed test 'today() should return a DD/MM/YYYY format'
#   at t/testit.t line 11.
#                   '3/4/2012'
#     doesn't match '(?x-ism:^ \d\d/\d\d/\d\d\d\d $)'
# Looks like you failed 1 test of 4.
```

The regular expression attempts to match two digits, followed by a slash, two more digits, a slash, and four digits. The `^` and `$` anchors force the regex to match from the start to the end of the string.

At the end it fails because `3/4/2012` clearly does not match the regular expression. You have two choices: If you decide your code is correct, then you fix the test. In this case, you decide that your test is correct and the code is in error. (This is usually the case for failing tests.) So fix the code:

```
sub order_date {
    my $today = DateTime->now->subtract( days => rand(30) );
    return sprintf "%02d/%02d/%04d" =>
        map { $today->$_ } qw/day month year/;
}
```

Now, no matter how many times you run your tests, the tests pass.

The other choice is to decide that your test is wrong and fix the test. The answer is not always this obvious, so pay careful attention to test failures.

Using is_deeply

Sometimes you want to compare data structures. Consider this:

```
is [ 3, 4 ], [ 3, 4 ], 'Identical array refs should match';
```

That may generate an error like this:

```
not ok 1 - Identical array refs should match
#   Failed test 'Identical array refs should match'
#   at some_script.t line 2.
#          got: 'ARRAY(0x7fc9eb0032a0)'
#     expected: 'ARRAY(0x7fc9eb026048)'
```

What happened? The `is()` functions tests whether two values are equal and, in this case, those values are references, not their contents. That's where the `is_deeply()` function comes in.

```
use Test::More;
is_deeply [ 3, 4 ], [ 3, 4 ], 'Identical array refs should match';
```

That test passes because the array reference contents are identical. However, if the contents are not identical:

```
is_deeply [ 3, 4 ], [ 3, 4, 1 ], 'Identical array refs should match';
```

You get an array like this:

```
not ok 1 - Identical array refs should match
#   Failed test 'Identical array refs should match'
#   at some_script.t line 2.
#     Structures begin differing at:
#          $got->[2] = Does not exist
#     $expected->[2] = '1'
```

The `is_deeply()` test function walks through each data structure, in parallel, keeping track of where it is and reports an error at the first item in each structure that is different. If there

are multiple differences, it can be annoying to track them all down, but later you look at the `Test::Differences` module that will make this easier.

Using SKIP

Sometimes you need to test code that should not always be run. For example, maybe some features work only if you have a particular module installed. You can use `SKIP` to skip them. `SKIP` looks like this:

```
SKIP: {
    skip $why, $how_many if $some_condition;
    # tests
}
```

And if `$some_condition` is true, `skip()` prints out several "successful" test lines (matching `$how_many`) and the rest of the block will be skipped. Here's a concrete example:

```
SKIP: {
    skip "Don't have an internet connection", 2
        unless have_internet_connection();
    my $website;
    ok $website = get_website($some_site),
      'We should be able to get the website';
    is $website->title, 'Some Title',
      '... and the title should be correct';
}
```

And assuming that `have_internet_connection()` returns false, you get output similar to the following:

```
ok 1 # skip Don't have an internet connection
ok 2 # skip Don't have an internet connection
```

Using TODO

Maybe you have a feature that you haven't completed writing, but you still want to write tests for it without necessarily having the test suite fail. You can do this with a `SKIP` test, but that's not quite what you want. Instead, you want the tests to be run, but not have their failures make the entire test suite fail. Use a `TODO` test for that. The structure looks like this:

```
TODO: {
    local $TODO = 'These tests do not work yet';
    # some tests
}
```

Let's add the following function to `TestMe`.

```
sub unique {
    my @array = @_;
    my %hash  = map { $_ => 1 } @array;
    return keys %hash;
}
```

And then add this to your `testit.t` program:

```
TODO: {
    local $TODO = 'Figure out how to avoid random order';
    my @have = unique( 2, 3, 5, 4, 3, 5, 7 );
    my @want = ( 2, 3, 5, 4, 7 );
    is_deeply \@have, \@want,
      'unique() should return unique() elements in order';
}
```

When you run your tests, you probably get an error similar to this. (The `not ok` and the `# TODO` lines are broken over two lines here only to fit the formatting of the book. Ordinarily they are on a single line.)

```
not ok 5 - unique() should return unique() elements in order
# TODO Figure out how to avoid random order
#   Failed (TODO) test 'unique() should return unique() elements in order'
#   at t/testit.t line 19.
#     Structures begin differing at:
#          $got->[0] = '0'
#     $expected->[0] = '2'
```

Perl can recognize that everything in the `TODO` block should not be considered a failure, even if the test fails. When the test does pass, you see output like this:

```
t/testit.t ..
ok 1 - The reciprocal of 2 should be true
ok 2 - The reciprocal of 2 should be correct
ok 3 - The reciprocal of 3 should be correct
ok 4 - today() should return a DD/MM/YYYY format
ok 5 - unique() should return unique() elements in order # TODO ...
1..5
ok
All tests successful.
Test Summary Report
-------------------
t/testit.t (Wstat: 0 Tests: 5 Failed: 0)
  TODO passed:   5
Files=1, Tests=5,  0 wallclock secs
Result: PASS
```

There is a `TODO passed: 5` note in the test output footer. That helps you track down a test that unexpectedly succeeds when it's in a `TODO` block.

Using eval {}

Sometimes you want to test failures that would ordinarily kill your test program, but you would like to keep your tests running. Consider `reciprocal()`:. What happens when you pass a zero? You can use `eval` to trap the error:

```
eval { reciprocal(0) };
my $error = $@;
like $error, qr{Illegal division by zero at t/testit.t},
  'reciprocal(0) should report an error from the caller';
```

> ***NOTE*** *If you don't remember the* `eval/$@` *syntax, see the* `eval` *section in Chapter 7.*

In this case, you don't want `reciprocal()` to report where the error occurred but to report the calling code that caused the error. The `eval` ensures that your tests can keep running, but your test fails:

```
not ok 6 - reciprocal(0) should report an error from the caller
#   Failed test 'reciprocal(0) should report an error from the caller'
#   at t/testit.t line 25.
#                   'Illegal division by zero at t/testit.t line 23.
# '
#     doesn't match '(?-xism:Illegal division by zero at testit.t)'
```

To make the test pass, make sure that you include `use Carp 'croak'` in `lib/TestMe.pm` and then fix the `reciprocal()` function:

```
sub reciprocal {
    my $number = shift;
    unless ($number) {
        croak("Illegal division by zero");
    }
    return 1 / $number;
}
```

Rerun your tests and now they should pass.

Using use_ok and require_ok

Sometimes you want to know if you can use a module you've written. You'll often see people do this:

```
use_ok     'My::Module', 'My::Module should load';
require_ok 'My::Module', 'My::Module should load';
```

You shouldn't use either of these; just use `use` *module* or `require` *module* as necessary. If the `use` or `require` fails, your script will die and you'll get a test failure reported. Without going into long, painful technical detail, suffice it to say that both these functions have several problems. People have tried to correct some of the problems with `use_ok` by doing this:

```
BEGIN {
    use_ok 'My::Module', 'My::Module should load'
      or die "Cannot load 'My::Module'";
}
```

But at this point, you may as well `use My::Module;` because the above construct doesn't gain you much.

Working with Miscellaneous Test Functions

The `can_ok` function tests whether the first argument (an object or class name) can execute the methods listed in `@list_of_method_names`.

```
can_ok $package_or_class, @list_of_method_names;
```

No description is required because `can_ok()` supplies one for you. It's the same as:

```
ok $package_or_class->can($method), "Package can execute $method";
```

But it works for multiple methods at once.

The `isa_ok` function is similar, but it takes only one class name to test against:

```
isa_ok $object, $class;
```

It's identical to:

```
ok $object->isa($class), "object isa $class";
```

However, you don't have to worry about wrapping it an `eval` and it provides the descriptive test name for you.

The `diag()` function lets you spit out diagnostic messages without worrying about them interfering with test output:

```
diag( "Testing with Perl $], $^X" );
```

If that's in your tests, it might output something like the following when you run your tests:

```
# Testing with Perl 5.012004 ~perl5/perlbrew/perls/perl-5.12.4/bin/perl
```

With well-thought-out `diag()` messages, if someone reports a failure in one of your modules, you can often have them send you the complete test failure output and better understand why your code failed.

> ***NOTE*** *In your sample* `diag()` *message, you included the* `$]` *and* `$^X` *special variables. These represent the Perl version and the path to the currently executing Perl. See* `perldoc perlvar` *for more information.*

TRY IT OUT Testing a Complex Function

So far you have seen a few trivial examples of testing, but let's look at something a little bit more real world. Generally when you write code, you want to reuse working code from the CPAN (or some other source) rather than write all of it yourself, but in this case, you write a simple query string parser (those extra bits on URLs that you see after a `?` sign) and test that it behaves correctly. Though query strings are relatively simple, there are plenty of examples of broken parsers on the web, and you don't want a broken example.

You're not actually going to do the work of decoding the characters from RFC 3986 encoding (`http://tools.ietf.org/html/rfc3986`). Let `URI::Escape` do that. Instead, ensure that you can translate a query string into a data structure you can easily test against.

All the code for this Try It Out is in the code file `lib/TestQuery.pm` and `t/query.t`.

1. Save the following as `lib/TestQuery.pm`:

```
package TestQuery;
use strict;
use warnings;
use URI::Escape 'uri_unescape';
use Encode 'decode_utf8';
use Exporter::NoWork;
sub parse_query_string {
    my $query_string = shift;
    my @pairs = split /[&;]/ => $query_string;
    my %values_for;
    foreach my $pair (@pairs) {
        my ( $key, $value ) = split( /=/, $pair );
        $_ = decode_utf8( uri_unescape($_) ) for $key, $value;
        $values_for{$key} ||= [];
        push @{ $values_for{$key} } => $value;
    }
    return \%values_for;
}
1;
```

2. Save the following test script as `t/query.t`:

```
use strict;
use warnings;
use Test::More;
use lib 'lib';
use TestQuery ':ALL';
my $query = parse_query_string('name=Ovid&color=black');
is_deeply $query, { name => ['Ovid'], color => ['black'] },
  'A basic query string parsing should be correct';
$query = parse_query_string('color=blue&color=white&color=red');
is_deeply $query, { color => [qw/blue white red/] },
  '... and multi-valued params should also parse correctly';
$query = parse_query_string('color=blue;color=white;color=red');
is_deeply $query, { color => [qw/blue white red/] },
  '... even if we are using the alternate ";" delimiter';
$query = parse_query_string('remark=%28parentheses%21%29');
is_deeply $query, { remark => ['(parentheses!)'] },
  '... or URI-encoded characters';
my $omega = "\N{U+2126}";
$query = parse_query_string('alpha=%E2%84%A6');
is_deeply $query, { alpha => [$omega] },
    '... and Unicode should decode correctly';
done_testing;
```

When you finish, your directory structure should look like this, assuming you also wrote the `lib/TestMe.pm` and `t/testit.t` programs:

```
lib/
|--TestMe.pm
|--TestQuery.pm
t/
|--query.t
|--testit.t
```

3. Run the test program with `prove -v t/query.t`. You should see the following output, verifying that you have correctly parsed the query strings:

```
t/query.t ..
ok 1 - A basic query string parsing should be correct
ok 2 - ... and multi-valued params should also parse correctly
ok 3 - ... even if we are using the alternate ";" delimiter
ok 4 - ... or URI-encoded characters
ok 5 - ... and Unicode should decode correctly
1..5
ok
All tests successful.
Files=1, Tests=5
Result: PASS
```

How It Works

When you have a query string like `name=value&color=blue`, you want that returned in a hash reference, such as `{ name => 'value', color => 'blue' }`. However, what many people get wrong is that this is a perfectly valid query string:

```
name=value;color=red;color=blue
```

The separator in that query string is a semicolon, `;`, and not an ampersand, `&`. You also have more than one value for `color`. Because you can have multiple values for each parameter, use array references and your resulting data structure should look like this:

```
{
    name  => [ 'value' ],
    color => [ 'red', 'blue' ].
}
```

And that's what your `parse_query_string()` function does:

```
sub parse_query_string {
    my $query_string = shift;
    my @pairs = split /[&;]/ => $query_string;
    my %values_for;
    foreach my $pair (@pairs) {
        my ( $key, $value ) = split( /=/, $pair );
        $_ = decode_utf8( uri_unescape($_) ) for $key, $value;
        $values_for{$key} ||= [];
        push @{ $values_for{$key} } => $value;
    }
    return \%values_for;
}
```

This line:

```
my @pairs = split /[&;]/ => $query_string;
```

Splits the query string on ampersands and semicolons into key value pairs.

And the body of the loop

```
my ( $key, $value ) = split( /=/, $pair );
$_ = decode_utf8( uri_unescape($_) ) for $key, $value;
$values_for{$key} ||= [];
push @{ $values_for{$key} } => $value;
```

That splits each key/value pair on the = sign and the following line calls `uri_unescape` and `decode_utf8` on each key and value. Then you create an array reference if you didn't have one, pushing the new value onto that array reference and storing it in the `%values_for hash`.

The reason you need to `decode_utf8` on the unescaped value is because the `uri_unescape` function can decode the characters into a byte string, but you need `decode_utf8` to turn that byte string into a proper UTF-8 string. See the Unicode section in Chapter 9 for more details. You may want to remove the `decode_utf8` and run the tests again. You can examine the test failure to see the difference in behavior.

Now look at the first test:

```
my $query = parse_query_string('name=Ovid&color=black');
is_deeply $query, { name => ['Ovid'], color => ['black'] },
  'A basic query string parsing should be correct';
```

In this, you parse a query string and expect two name/value pairs to be returned in a hash reference. The `is_deeply()` test function does a deep check to verify that the `$query` returned matches the expected data structure you supply.

Run the test with a variety of different query strings:

- `color=blue&color=white&color=red`
- `color=blue;color=white;color=red`
- `remark=%28parentheses%21%29`
- `alpha=%E2%84%A6`

The last one is rather interesting. The Greek letter Omega (Ω) is the Unicode code point `U+2126`. Its three-byte sequence is `\xe2\x84\xa6` and that gets URI encoded as `%E2%84%A6`. Without the `decode_utf8`, the `uri_unescape` function would return that three octets from the URI encoding. The `uri_unescape` function does not actually know about Unicode, which is why you need to decode it manually. Thus, you want a solid test here to verify that you are properly handling Unicode in query strings.

```
my $omega = "\N{U+2126}";
$query = parse_query_string('alpha=%E2%84%A6');
is_deeply $query, { alpha => [$omega] },
    '... and Unicode should decode correctly';
```

Now that you have a test program, can you think of other use cases you might want to handle in query strings? What do you do if you have no value for a parameter (for example: `name=&color=red`)? If you write another function that validates that your query strings have particular keys and values matching certain parameters, how would you test that? These are the sorts of questions you need to face when writing tests and the answers can vary depending on your needs.

USING OTHER TESTING MODULES

In reality, you could write an entire test suite using only the `ok()` test function. However, that would not provide you with good diagnostic information. Instead, there are a wide variety of other testing functions available from `Test::More` (not all of which are covered in this book). However, `Test::More`, while being great, is often not enough. Instead, you might want to use other testing modules that can make testing your code even easier. Some of the more popular ones are covered next.

Using Test::Differences

Consider the following test:

```
use Test::More;
my %have = (
    numbers => [ 3, 4 ],
    fields  => { this => 'that', big => 'bad' },
    extra   => [ 3, 4 ],
);
my %want = (
    numbers => [ 3, 4 ],
    fields  => { this => 'taht', big => 'bad' },
    extra   => [ 4, 4, ],
);
is_deeply \%have, \%want, 'have and want should be the same';
done_testing;
```

And that prints out:

```
not ok 1 - have and want should be the same
#   Failed test 'have and want should be the same'
#   at /var/tmp/eval_q5aW.pl line 12.
#     Structures begin differing at:
#          $got->{fields}{this} = 'that'
#     $expected->{fields}{this} = 'taht'
1..1
# Looks like you failed 1 test of 1.
```

That tells you the items don't match, but it tells you only the first item that it found that failed to match. And if you data structure is extremely complicated, you might dig through something like:

```
#          $got->{fields}[0][3]{some_value} = 'that'
#     $expected->{fields}[0][3]{some_value} = 'taht'
```

And that's not a lot of fun. Instead, you can use the `Test::Differences` modules. It exports several test functions, the most common of which is `eq_or_diff()`. It looks like this:

```
use Test::More;
use Test::Differences;
my %have = (
    numbers => [ 3, 4 ],
    fields  => { this => 'that', big => 'bad' },
    extra   => [ 3, 4 ],
);
my %want = (
    numbers => [ 3, 4 ],
    fields  => { this => 'taht', big => 'bad' },
    extra   => [ 4, 4, ],
);
eq_or_diff \%have, \%want, 'have and want should be the same';
done_testing;
```

When you run that, you get a full diff:

```
not ok 1 - have and want should be the same
#   Failed test 'have and want should be the same'
#   at /var/tmp/eval_1h41.pl line 13.
# +----+--------------------+--------------------+
# | Elt|Got                 |Expected            |
# +----+--------------------+--------------------+
# |   0|{                   |{                   |
# |   1|  extra => [        |  extra => [        |
# *   2|    3,              |    4,              *
# |   3|    4               |    4               |
# |   4|  ],                |  ],                |
# |   5|  fields => {       |  fields => {       |
# |   6|    big => 'bad',   |    big => 'bad',   |
# *   7|    this => 'that'  |    this => 'taht'  *
# |   8|  },                |  },                |
# |   9|  numbers => [      |  numbers => [      |
# |  10|    3,              |    3,              |
# |  11|    4               |    4               |
# |  12|  ]                 |  ]                 |
# |  13|}                   |}                   |
# +----+--------------------+--------------------+
1..1
# Looks like you failed 1 test of 1.
```

The `Elt` column is the line number of the diff. The `Got` column is what you have, and the `Expected` column is what you want. Prior to each `Elt` number matching an incorrect line is an asterisk showing which lines are different.

For extremely large data structures, it show only a range of lines and also has functions to customize the output.

Using Test::Exception

You may remember how you tested a function that might die:

```
eval { reciprocal(0) };
my $error = $@;
like $error, qr{Illegal division by zero at t/testit.t},
  'reciprocal(0) should report an error from the caller';
```

The `Test::Exception` module makes this a little bit cleaner:

```
use Test::Exception;
throws_ok { reciprocal(0) }
   qr{Illegal division by zero at t/testit.t},
  'reciprocal(0) should report an error from the caller';
```

The first argument to `throws_ok` is a subroutine reference. Because the prototype to `throws_ok` is `(&$;$)` (see the prototypes section in Chapter 7, "Subroutines"), the first argument can be a block, `{ ... }`, without the `sub` keyword in front of it.

The second argument is a regular expression compiled with the `qr//` builtin. This is a common mistake:

```
use Test::Exception;
throws_ok { some_code() }
  /some regular expression/,
  'I should get the correct error';
```

When you see a bare regular expression in the form of `/.../`, that's executed immediately against `$_`. You must use the `qr//` compiled form of the regular expression.

> ***NOTE*** *When using* `Test::Exception::throws_ok` *function (or any function that has a* `&` *prototype), you do not use a trailing comma if you provide a block:*
>
> ```
> throws_ok { some_code() } qr/.../, $message;
> ```
>
> *You do need the trailing comma if you provide the* `sub` *keyword:*
>
> ```
> throws_ok sub { some_code() }, qr/.../, $message;
> ```
>
> `Test::Exception` *exports other test functions, such as* `dies_ok` *and* `lives_ok`*, so read the documentation for a full understanding of its capabilities.*

Using Test::Warn

Sometimes your code throws warnings. There are two types:

- Those generated by `perl` (such as the common `Use of uninitialized value`).
- Warnings the programmer creates. Warnings issued by `perl` should be dealt with and eliminated because those are warnings to the developer writing some code, but warnings created by the programmer should be tested because those are warnings to the *user* of the code. This is where `Test::Warn` comes in.

Consider the following code snippet:

```
sub read_config {
    my $config = shift;
    unless ( -f $config && -r _ ) {
        carp "Cannot read '$config'. using default config";
        $config = $DEFAULT_CONFIG;
    }
    # read the config
}
```

You can test that with `Test::Warn` with code like the following:

```
use Test::More;
use Test::Warn;
warning_is { read_config($config) }
   "Cannot read '$config'. using default config",
   'Reading a bad config should generate a warning';
```

Code might throw no warnings or multiple warnings, or you may need to test the warnings against a regular expression. `Test::Warn` handles all these cases.

Using Test::Most

Finally, here's your author's favorite `Test::` module (disclaimer: he wrote it), `Test::Most`, a name chosen to gently make fun of the `Test::More` module's name. (The authors are actually good friends, so this isn't mean-spirited.)

When dealing with large test suites, it's common to see something like this at the beginning of each test module:

```
use strict;
use warnings;
use Test::Exception;
use Test::Differences;
use Test::Deep;
use Test::Warn;
use Test::More tests => 42;
```

That's a lot of typing and frankly, it gets annoying re-creating this boilerplate every time. As a result, the `Test::Most` module was created. The above boilerplate (even the `strict` and `warnings`) can be replaced with this:

```
use Test::Most tests => 42;
```

You've now eliminated six lines of code and can just start writing tests without retyping those six lines every time (or forgetting one and going back and adding it). The test modules in question were chosen by running a heuristic analysis over the entire CPAN and seeing which test modules were most commonly used and including the appropriate ones in `Test::Most`.

> ***NOTE*** *If you like this idea but want a different list of modules, you author has also released* `Test::Kit`*. This allows you to assemble your own list of modules to bundle together.*

UNDERSTANDING XUNIT STYLE USING TESTING

If you're familiar with testing in other languages, you may be familiar with *xUnit* style testing. This is a type of testing that is particularly well suited to object-oriented code. Just as you have classes that you want to test, with xUnit style testing, you can create corresponding test classes. These test classes can inherit tests from one another, just as your classes can inherit from other classes.

The most popular xUnit style testing package in Perl is called `Test::Class` and that's what you'll use.

> ***WARNING*** *Sometimes you'll find people using* `Test::Unit` *for xUnit style testing instead of* `Test::Class`*. Unfortunately,* `Test::Unit` *is not compatible with* `Test::Builder`*, the module that most modern Perl testing tools are built with, so you should not use it. It has not been updated since 2005 and appears to be abandoned.*

Using Test::Class

Start by creating a `Person` class. The `Person` has a given name, a family name (analogous to first name and last name), an optional read/write title, such as Dr., and a birth date. The class will look like the following code (code file: `lib/Person.pm`):

```
package Person;
use Moose;
use Moose::Util::TypeConstraints;
use DateTime::Format::Strptime;
use namespace::autoclean;
# Moose doesn't know about non-Moose-based classes.
```

```
class_type 'DateTime';
my $datetime_formatter = DateTime::Format::Strptime->new(
    pattern   => '%Y-%m-%d',
    time_zone => 'GMT',
);
coerce 'DateTime'
  => from 'Str'
  => via { $datetime_formatter->parse_datetime($_) };
use DateTime;
has 'given_name'  => ( is => 'ro', isa => 'Str', required => 1 );
has 'family_name' => ( is => 'ro', isa => 'Str', required => 1 );
has 'title'       => ( is => 'rw', isa => 'Str', required => 0 );
has 'birthdate' =>
  ( is => 'ro', isa => 'DateTime', required => 1, coerce => 1 );
sub name {
    my $self = shift;
    my $name = '';
    if ( my $title = $self->title ) {
        $name = "$title ";
    }
    $name .= join ' ', $self->given_name, $self->family_name;
    return $name;
}
sub age {
    my $self     = shift;
    my $duration = DateTime->now - $self->birthdate;
    return $duration->years;
}
__PACKAGE__->meta->make_immutable;
1;
```

NOTE *The* `DateTime::Format::Strptime` *module can help you convert date strings like* `1967-06-20` *to a proper* `DateTime` *object. Then use special* `Moose` *code to coerce strings to* `DateTime` *objects for the* `birthdate`*. That lets you do this:*

```
my $person = Person->new(
    title       => 'President',
    given_name  => 'Dwight',
    family_name => 'Eisenhower',
    birthdate   => '1890-10-14',
);
```

That's much easier than creating the entire `DateTime` *object every time.*

Chapter 13 doesn't cover coercions for `Moose` *because they're a bit more advanced, so you won't see them here, but you can read* `Moose::Cookbook::Basics::Recipe5` *to better understand how to use coercions in your own* `Moose` *classes.*

You could write normal `.t` scripts for this, but I'll show you how to take advantage of the power of `Test::Class` to extend these classes.

To start writing your tests, you need a little bit of code to make it easy to run the tests. There are a variety of ways to configure how to run `Test::Class` tests, but the following method is your author's preferred setup. It is extremely flexible and makes running your classes a breeze.

First, make a "driver" test that can run all your test classes at once. Save the following (code file `t/test_classes.t`).

```
use strict;
use warnings;
use Test::Class::Load 't/lib';
```

That script can find all `Test::Class` tests in the `t/lib` directory and run them. It can also add `t/lib` to `@INC` to ensure that Perl can find them, too. (`@INC` is the special Perl variable telling Perl which paths to look for code in. For more information, see Chapter 11).

Next, create a `t/lib` directory and add the following code (code file `t/lib/TestsFor.pm`):

```
package TestsFor;
use Test::Most;
use base 'Test::Class';
INIT { Test::Class->runtests }
sub startup  : Tests(startup)  {}
sub setup    : Tests(setup)    {}
sub teardown : Tests(teardown) {}
sub shutdown : Tests(shutdown) {}
1;
```

Now use the `TestsFor::` namespace to ensure that your test classes do not try to use a namespace in use by another package. The `INIT` block tells `Test::Class` to run all the tests after they have been compiled. I'll explain the `startup`, `setup`, `teardown`, and `shutdown` methods in the "Using Test Control Methods" section later in this chapter. The `use base 'Test::Class'`, tells Perl that `TestsFor.pm` will inherit from the `Test::Class` module. That's what makes all this work.

A Basic Test Class

The code files `t/test_classes.t` script and `t/lib/TestsFor.pm` class are all the code you need for your setup. Now you can begin to write tests for your `Person` class. You need to create a `t/lib/TestsFor/` directory and save the following code as `t/lib/TestsFor/Person.pm`:

```
package TestsFor::Person;
use Test::Most;
use base 'TestsFor';
use Person;
use DateTime;

sub class_to_test { 'Person' }

sub constructor : Tests(3) {
```

```
    my $test = shift;
    my $class = $test->class_to_test;
    can_ok $class, 'new';
    throws_ok { $class->new }
    qr/Attribute.*required/,
      "Creating a $class without proper attributes should fail";
    my $person = $class->new(
        given_name  => 'Charles',
        family_name => 'Drew',
        birthdate   => '1904-06-03',
    );
    isa_ok $person, $class;
}
1;
```

You can now run `prove -lv t/test_classes.t` to should see the following output:

```
$ prove -lv t/test_classes.t
t/test_classes.t .. #
# TestsFor::Person->constructor
1..3
ok 1 - Person->can('new')
ok 2 - Creating a Person without proper attributes should fail
ok 3 - The object isa Person
ok
All tests successful.
Files=1, Tests=3,  2 wallclock secs
Result: PASS
```

Now break down what's happening here. Here are the opening lines from the `TestsFor::Person` class:

```
package TestsFor::Person;
use Test::Most;
use base 'TestsFor';
use Person;
use DateTime;
```

Use `Test::Most` to avoid repeatedly using many test modules, plus, it turns on `strict` and `warnings`. You inherit from your `TestsFor` module because when you create test classes, it's a good idea to have a common test class that you might have shared methods in, such as methods to connect to a database.

You also have this curious bit:

```
sub class_to_test { 'Person' }
```

Why do you do that? Because you want to ensure that your subclasses can override this method to assert which class they are testing. This becomes more clear when you learn how to inherit from subclasses.

Now take a close look at your test method.

The test method is defined as:

```
sub constructor : Tests(3) {
    ...
}
```

The `: Tests(3)` bit is called a *subroutine attribute*. These are not the same thing as class attributes (data attached to a class). These are special extra bits of information you can attach to a subroutine to provide a bit more information about them. The syntax looks like this:

```
sub NAME : ATTRIBUTES { ... }
```

By themselves they don't do anything, but some authors use the `Attribute::Handlers` module to define custom attributes and describe their meaning. I won't cover them more except to say that `Test::Class` has attributes for test methods to mark them as something that `Test::Class` knows how to deal with. So this:

```
sub customer : Tests(3) { ... }
```

Tells `Test::Class` that you will run three tests in this test method. If you run more, you have a test failure. If you run fewer, `Test::Class` assumes you meant to skip the tests and issues "skipped test" lines in your test output.

> ***WARNING*** *You used the name* `constructor()` *for the* `new()` *method because* `Test::Class` *has its own* `new()` *method, and overriding that method can cause strange bugs in your test classes. Never override* `new()` *unless you know exactly what you're doing.*

If you're not sure how many tests you will run in a test method, you can omit the number of tests:

```
sub some_test_method : Tests { ... }
```

The next line is this:

```
my $test = shift;
```

Ordinarily the invocant to a method is called `$self` and most developers who use `Test::Class` write my `$self = shift`. Your author prefers to use the `$test` variable to help keep you in the frame of mind of "writing tests." It's silly, but there you go.

The rest of the test method is composed of standard testing code that you already understand.

When you run a test class, the `Test::Class` module can find all methods in all test class with a `:Tests` attribute and execute them sequentially as tests, grouping test results per test class, per test method.

Now add test methods for the `name` and `age` methods. First, because you'll be instantiating the `Person` class several times, pull that out into its own method `default_person()` method:

```
sub constructor : Tests(3) {
    my $test = shift;
    my $class = $test->class_to_test;
    can_ok $class, 'new';
    throws_ok { $class->new }
    qr/Attribute.*required/,
      "Creating a $class without proper attributes should fail";
    isa_ok $test->default_person, 'Person';
}
sub default_person {
    my $test = shift;
    return $test->class_to_test->new(
        given_name  => 'Charles',
        family_name => 'Drew',
        birthdate   => '1904-06-03',
    );
}
```

As you can see in the `constructor()` test method, you can write `$test->default_person` to get the `Person` object. If you rerun the tests, you see no change of behavior because `default_person()` does not have a `:Tests` attribute and `Test::Class` will not attempt to run it directly.

Now add the following test methods to the `TestsFor::Person` class:

```
sub name : Tests(2) {
    my $test   = shift;
    my $person = $test->default_person;
    is $person->name, 'Charles Drew', 'name() should return the full name';
    $person->title('Dr.');
    is $person->name, 'Dr. Charles Drew',
      '... and it should be correct if we have a title';
}
sub age : Tests(2) {
    my $test = shift;
    my $person = $test->default_person;
    can_ok $person, 'age';
    cmp_ok $person->age, '>', 100,
        'Our default person is more than one hundred years old';
}
```

Notice that in `name()` you take advantage that you made `title()` a read-write object attribute. In both of these, you used your `default_person()` method to avoid rewriting code.

After you've added these methods, you can rerun your tests and get this output:

```
t/test_classes.t .. #
1..8
# TestsFor::Person->age
ok 1 - Person->can('age')
ok 2 - Our default person is more than one hundred years old
```

```
#
# TestsFor::Person->constructor
ok 3 - Person->can('new')
ok 4 - Creating a Person without proper attributes should fail
ok 5 - The object isa Person
#
# TestsFor::Person->name
ok 6 - Person->can('name')
ok 7 - name() should return the full name
ok 8 - ... and it should be correct if we have a title
ok
All tests successful.
Files=1, Tests=8,  0 wallclock secs
Result: PASS
```

You can see diagnostic comments indicating which test methods have run, along with their corresponding tests.

Extending a Test Class

If your classes have subclasses, it's natural that your test classes have subclasses, too. Look at a `Customer` subclass of `Person`. It's identical to `Person`, but it's required to have a minimum age of `18`. First, write the `Customer` class (code file `lib/Customer.pm`):

```
package Customer;
use Moose;
extends 'Person';
use Carp 'croak';
use namespace::autoclean;
sub BUILD {
    my $self = shift;
    if ( $self->age < 18 ) {
        my $age = $self->age;
        croak("Customers must be 18 years old or older, not $age");
    }
}
__PACKAGE__->meta->make_immutable;
1;
```

As you can see, this is a simple, straightforward subclass with no complications. Now look at its test class (code file `t/lib/TestsFor/Customer.pm`):

```
package TestsFor::Customer;
use Test::Most;
use base 'TestsFor::Person';
use Customer;
use DateTime;

sub class_to_test { 'Customer' }

sub mininum_age : Tests(2) {
    my $test = shift;
```

```
    my $year = DateTime->now->year;
    $year -= 16;
    throws_ok {
        $test->class_to_test->new(
            given_name  => 'Sally',
            family_name => 'Forth',
            birthdate   => "$year-06-05",
        );
    }
    qr/^Customers must be 18 years old or older, not \d+/,
      'Trying to create a customer younger than 18 should fail';
    $year -= 10;    # take another ten years off
    lives_ok {
        $test->class_to_test->new(
            given_name  => 'Sally',
            family_name => 'Forth',
            birthdate   => "$year-06-05",
        );
    }
    'Trying to create a customer older than 18 should succeed';
}
1;
```

Just as the `Customer` class inherited from the `Person` class, the `TestsFor::Customer` class inherited from `TestsFor::Person` (using slightly different syntax because the classes are written with `Moose` and the test classes are not).

There are two interesting things here:

- You compute the year rather than hard-code it with something like `my $year = 1999;`. You do that to ensure that these tests don't start to break in the future.
- You again fetch the name of the class to test from your `class_to_test()` method. You'll see why right now.

So run this test class, but only this test class. Remember this line from in the `t/lib/TestsFor.pm` package?

```
INIT { Test::Class->runtests }
```

Because you have that, loading your tests automatically runs them, but you need to tell the `prove` utility where to find the test classes. You do this by passing `-It/lib` to `prove`.

```
$ prove -lv -It/lib t/lib/TestsFor/Customer.pm
t/lib/TestsFor/Customer.pm .. #
# TestsFor::Person->age
1..18
ok 1 - Person->can('age')
ok 2 - Our default person is more than one hundred years old
#
# TestsFor::Person->constructor
ok 3 - Person->can('new')
ok 4 - Creating a Person without proper attributes should fail
```

```
ok 5 - The object isa Person
#
# TestsFor::Person->name
ok 6 - Person->can('name')
ok 7 - name() should return the full name
ok 8 - ... and it should be correct if we have a title
#
# TestsFor::Customer->age
ok 9 - Customer->can('age')
ok 10 - Our default person is more than one hundred years old
#
# TestsFor::Customer->constructor
ok 11 - Customer->can('new')
ok 12 - Creating a Customer without proper attributes should fail
ok 13 - The object isa Customer
#
# TestsFor::Customer->mininum_age
ok 14 - Trying to create a customer younger than 18 should fail
ok 15 - Trying to create a customer older than 18 should succeed
#
# TestsFor::Customer->name
ok 16 - Customer->can('name')
ok 17 - name() should return the full name
ok 18 - ... and it should be correct if we have a title
ok
All tests successful.
Files=1, Tests=18,  1 wallclock secs
Result: PASS
```

> ***WARNING*** *Sometimes when you run* `Test::Class` *tests, you get a warning like the following:*
>
> ```
> t/lib/TestsFor/Customer.pm .. Invalid CODE attribute:
>   Tests(startup) at
> t/lib/TestsFor/Customer.pm line 12.
> BEGIN failed--compilation aborted at t/lib/TestsFor/Customer.pm
>   line 12.
> t/lib/TestsFor/Customer.pm .. Dubious, test returned 255
> No subtests run
> ```
>
> *This usually means that you have forgotten to tell Perl (or* `prove`*) where to find the test classes. Make sure that you have supplied the appropriate* `-It/lib` *switch to* `prove`*. If you run* `prove t/test_class.t` *and it uses* `Test::Class::Load`*, this issue will probably not happen.*

Whoa! What just happened here? Your `Person` class had only eight tests, and you added only two tests! That means ten tests, right?

Nope.

`Test::Class` knows that `TestsFor::Customer` inherited `TestsFor::Person`. Because of the Liskov Substitution Principle (mentioned in Chapter 12), you know that you should use a subclass in any place you can use a parent class. Thus, when you run the tests for `TestsFor::Customer`, `Test::Class` also runs all the tests you inherit from `TestsFor::Person`, but because of your `INIT` block in the `TestsFor` base class, it also ran the tests for `TestsFor::Person`.

This means that `TestsFor::Person` ran its eight tests, and `TestsFor::Customer` ran its two tests, plus the eight tests it inherited from `TestsFor::Person`. That makes 18 tests in total, even though you've only written 10 tests.

That is why I provided this method:

```
sub class_to_test { 'Person' }
```

Look at the `constructor()` test method again to understand why `TestsFor::Customer` overrode the `TestsFor::Person::class_to_test` method:

```
sub constructor : Tests(3) {
    my $test = shift;
    my $class = $test->class_to_test;
    can_ok $class, 'new';
    throws_ok { $class->new }
    qr/Attribute.*required/,
      "Creating a $class without proper attributes should fail";
    my $person = $class->new(
        given_name  => 'Charles',
        family_name => 'Drew',
        birthdate   => '1904-06-03',
    );
    isa_ok $person, $class;
}
```

Because you now fetch the class to test from a method you can override, that test method outputs the following:

```
# TestsFor::Customer->constructor
ok 11 - Customer->can('new')
ok 12 - Creating a Customer without proper attributes should fail
ok 13 - The object isa Customer
```

If you did not override the class name, you would have had this test out:

```
# TestsFor::Customer->constructor
ok 11 - Person->can('new')
ok 12 - Creating a Person without proper attributes should fail
ok 13 - The object isa Person
```

Clearly that would not be the test you would want. By allowing the code to override the name of the class, you allow your subclasses to use the parent class tests and ensure that your behavior did not change in the subclass.

At this point, you might wonder why your author showed you this:

```
$ prove -lv -It/lib t/lib/TestsFor/Customer.pm
```

Because of the `INIT` block in your subclass, that causes all tests to run for all loaded classes, not just `TestsFor::Customer`. However, you can do this with the `Person` class. (Leave off the `-v` so that the test output is not so verbose.)

```
$ prove -lv -It/lib t/lib/TestsFor/Person.pm
t/lib/TestsFor/Person.pm .. ok
All tests successful.
Files=1, Tests=8,  1 wallclock secs
Result: PASS
```

This shows that you have run only eight tests, and not the full 18. That's because you loaded `TestsFor::Person` and not `TestsFor::Customer`. If you have a large set of test classes, running different test classes like this can make it easier to run subsets of your tests and verify they work. Later, when you finish coding, you can rerun the full test suite with `prove -l t/`.

Using Test Control Methods

So far you might be getting an inkling of the power of `Test::Class`, but you're going to start feeding it steroids now and not only see you powerful it can be, but also how powerful object-oriented programming can be (two chapters for the price of one).

Remember the original `TestsFor` base class:

```
package TestsFor;
use Test::Most;
use base 'Test::Class';
INIT { Test::Class->runtests }
sub startup  : Tests(startup)  {}
sub setup    : Tests(setup)    {}
sub teardown : Tests(teardown) {}
sub shutdown : Tests(shutdown) {}
1;
```

You have `startup`, `setup`, `teardown`, and `shutdown` methods. Each of them has a corresponding `:Tests(methodname)` attribute. These are test control methods. These are run at the beginning and end of every class, or at the beginning and end of every test method, as shown in Table 14-1.

TABLE 14-1: Test Control Methods

METHOD	WHEN IT'S RUN
`startup`	Before every test class starts
`setup`	Before every test method start
`teardown`	After every test method ends
`shutdown`	After every test class ends

Now rewrite your `TestsFor` class. Use the `startup` method to load the class you want to test before each test class runs. Also set the class name in `class_to_test()`. Use multiple inheritance to provide class data here, but see `Test::Class::Most` to see a better way to do this:

```
package TestsFor;
use Test::Most;
use base qw(Test::Class Class::Data::Inheritable);
INIT {
    __PACKAGE__->mk_classdata('class_to_test');
    Test::Class->runtests;
}
sub startup : Tests(startup) {
    my $test  = shift;
    my $class = ref $test;
    $class =~ s/^TestsFor::/;
    eval "use $class";
    die $@ if $@;
    $test->class_to_test($class);
}
sub setup    : Tests(setup)    {}
sub teardown : Tests(teardown) {}
sub shutdown : Tests(shutdown) {}
1;
```

You inherited from both `Test::Class` and `Class::Data::Inheritable`. The `INIT` block now creates a class data method before running the tests. Use the `startup` method to do some magic! First, the `$test` invocant passed to the `startup` method is a reference, so use the `ref` function to determine the class name. Then use a substitution to strip off the `TestsFor::` prefix:

```
$class =~ s/^TestsFor::/;
```

That's when it gets interesting. Use `eval` to load the class and, if it succeeds, use `$test->class_to_test($class)` to set the class name on a per class basis!

```
eval "use $class";
die $@ if $@;
$test->class_to_test($class);
```

You can now edit `TestsFor::Person` and `TestsFor::Customer` and delete the `use Person` or `use Customer` lines, along with the `class_to_test()` methods. This is now automatically done for you!

Later, when you write `TestsFor::Order::Item`, the `Order::Item` class automatically loads without asking, and `class_to_test()`returns `Order::Item`, as you would expect. This little trick makes it much easier to build test classes on-the-fly.

You might also remember this method:

```
sub default_person {
    my $test = shift;
    return $test->class_to_test->new(
        given_name  => 'Charles',
```

```
            family_name => 'Drew',
            birthdate   => '1904-06-03',
        );
    }
```

Every time you call that method, the default person is re-created. If you want, you could set this in the `setup` method prior to every test. Make sure to do this in your `TestsFor::Person` class and not your `TestsFor` class because other test classes are likely to inherit from `TestsFor` and won't necessarily need a default `Person` object.

By doing this, your `TestsFor::Person` class now looks like this:

```
package TestsFor::Person;
use Test::Most;
use base 'TestsFor';
sub startup : Tests(startup) {
    my $test = shift;
    $test->SUPER::startup;
    my $class = ref $test;
    $class->mk_classdata('default_person');
}
sub setup : Tests(setup) {
    my $test = shift;
    $test->SUPER::setup;
    $test->default_person(
        $test->class_to_test->new(
            given_name  => 'Charles',
            family_name => 'Drew',
            birthdate   => '1904-06-03',
        )
    );
}
sub constructor : Tests(3) {
    # constructor tests
}
sub name : Tests(3) {
    # name tests
}
sub age : Tests(2) {
    # age tests
}
1;
```

You didn't use `Person` (because you no longer need to) and the `class_to_test()` method returns the correct class.

Calling Parent Test Control Methods

Now look at that `startup` method you created for `TestsFor::Person`:

```
sub startup : Tests(startup) {
    my $test = shift;
    $test->SUPER::startup;
```

```
    my $class = ref $test;
    $class->mk_classdata('default_person');
}
```

After you shift off the invocant (`$test`) but before you create the `default_person()` class data method, call `$test->SUPER::startup`. Why? Because you want to ensure that your `TestsFor::startup` method has been called and loaded your test class and set the `class_to_test()` method. Generally, when you call `setup` and `startup` methods, there may be a parent method available, and you should call them first to ensure that your test class has everything it needs to run.

In fact, The `setup()` method calls `$self->SUPER::setup` even though you have an empty `setup` method in your base class. Later, you may find that you need your parent classes to have some code in the setup method and you want to ensure that your subclasses do not mysteriously break by forgetting to call this important code.

The `teardown` and `shutdown` test control methods are not called as often, but they're used for things like cleaning up temp files or perhaps closing a database connection. When you use them, make sure to call the `SUPER::` method after you have done your cleanup (unless you have a good reason not to). The reason for this is because if you call the parent `teardown` or `shutdown` method before you're cleaning up in the subclass method, it's entirely possible that the database handle you needed has been destroyed, or some other critical bit of your test class state is gone. By running your `teardown` or `shutdown` code first and then calling the parent method, you can make sure you haven't destroyed anything you need until nothing else needs it.

Given what I've explained, for each test control method you think you need, the methods should look like this:

```
# startup and setup call parent methods before their code
sub startup : Tests(startup) {
    my $test = shift;
    $test->SUPER::startup;
    # startup up code here
}
sub setup : Tests(setup) {
    my $test = shift;
    $test->SUPER::setup;
    # setup up code here
}
# teardown and shutdown call parent methods after their code
sub teardown : Tests(teardown) {
    my $test = shift;
    # teardown up code here
    $test->SUPER::teardown;
}
sub shutdown : Tests(shutdown) {
    # shutdown up code here
    $test->SUPER::shutdown;
}
```

This introduction to `Test::Class` only skims the surface of what you can do. The author recommends his five-part online tutorial on `Test::Class` at `http://www.modernperlbooks.com/mt/2009/03/organizing-test-suites-with-testclass.html`.

TRY IT OUT Write Tests for the TV::Episode Class

You may recall the `TV::Episode` class from Chapter 13. This Try It Out shows you how to write some tests for it using `Test::Class`and assumes that you are already using your `TestsFor` class. All the code for this Try It Out is in code file `lib/TV/Episode.pm`.

1. Type in the following program and save it as `lib/TV/Episode.pm` (assuming you didn't do that for Chapter 13):

```
package TV::Episode;
use Moose;
use MooseX::StrictConstructor;
use Moose::Util::TypeConstraints;
use namespace::autoclean;
use Carp 'croak';
our $VERSION = '0.01';
subtype 'IntPositive',
         as    'Int',
         where { $_ > 0 };
has 'series'         => ( is => 'ro', isa => 'Str',
  required => 1 );
has 'director'       => ( is => 'ro', isa => 'Str',
  required => 1 );
has 'title'          => ( is => 'ro', isa => 'Str',
  required => 1 );
has 'season'         => ( is => 'ro', isa => 'IntPositive',
  required => 1 );
has 'episode_number' => ( is => 'ro', isa => 'IntPositive',
  required => 1 );
has 'genre'          => (
    is       => 'ro',
    isa      => enum(qw(comedy drama documentary awesome)),
    required => 1
);
sub as_string {
    my $self = shift;
    my @attributes = map { $_->name }
      $self->meta->get_all_attributes;
    my $as_string = '';
    foreach my $attribute (@attributes) {
        $as_string .= sprintf "%-14s - %s\n", ucfirst($attribute),
          $self->$attribute;
    }
    return $as_string;
}
__PACKAGE__->meta->make_immutable;
1;
```

2. Type in the following program, and save it as `t/lib/TestsFor/TV/Episode.pm`. (Create the `t/lib/TestsFor/TV` directory first.)

```
package TestsFor::TV::Episode;
use Test::Most;
use base 'TestsFor';
sub attributes : Tests(14) {
```

```
    my $test               = shift;
    my %default_attributes = (
        series         => 'Firefly',
        director       => 'Marita Grabiak',
        title          => 'Jaynestown',
        genre          => 'awesome',
        season         => 1,
        episode_number => 7,
    );
    my $class   = $test->class_to_test;
    my $episode = $class->new(%default_attributes);
    while (my ($attribute, $value) = each %default_attributes) {
        can_ok $episode, $attribute;
        is $episode->$attribute, $value,
          "The value for '$attribute' should be correct";
    }
    my %attributes = %default_attributes;    # copy 'em
    foreach my $attribute (qw/season episode_number/) {
        $attributes{$attribute} = 0;
        throws_ok { $class->new(%attributes) }
        qr/\Q($attribute) does not pass the type constraint/,
          "Setting $attribute to less than zero should fail";
    }
}
1;
```

3. Run the program with `prove -lv t/lib t/lib/TestsFor/TV/Episode.pm`. You should see the following output:

```
$ prove -lv -It/lib t/lib/TestsFor/TV/Episode.pm
t/lib/TestsFor/TV/Episode.pm .. #
# TestsFor::TV::Episode->attributes
1..14
ok 1 - TV::Episode->can('episode_number')
ok 2 - The value for 'episode_number' should be correct
ok 3 - TV::Episode->can('title')
ok 4 - The value for 'title' should be correct
ok 5 - TV::Episode->can('season')
ok 6 - The value for 'season' should be correct
ok 7 - TV::Episode->can('genre')
ok 8 - The value for 'genre' should be correct
ok 9 - TV::Episode->can('director')
ok 10 - The value for 'director' should be correct
ok 11 - TV::Episode->can('series')
ok 12 - The value for 'series' should be correct
ok 13 - Setting the season to a value less than zero should fail
ok 14 - Setting episode_number to less than zero should fail
ok
All tests successful.
Files=1, Tests=14,  1 wallclock secs
Result: PASS
```

4. Run the full test suite in nonverbose mode (leaving off the `-v` switch to prove) with `prove t/`. If you've worked through all the examples in this chapter, it should run tests for `t/testit.t`, `t/query.t`, and `t/test_classes.t`. The output should resemble the following:

```
$ prove t
t/query.t ......... ok
t/test_classes.t .. ok
t/testit.t ........ ok
All tests successful.
Files=3, Tests=43,  1 wallclock secs
Result: PASS
```

How It Works

At this point you've gone from the point of writing a few individual tests to having a full test suite (admittedly for a grab bag of unrelated modules).

You already know what `TV::Episode` does from Chapter 13, and `t/lib/TestsFor/TV/Episode.pm` are straightforward — there's nothing new there. Because you used the `TestsFor` base class you created earlier in this chapter, you didn't even need to have an explicit use `TV::Episode` line. The base class does that for you and sets the `$test->class_to_test value`.

This command is interesting:

```
$ prove -lv t/lib t/lib/TestsFor/TV/Episode.pm
```

Use the `-It/lib` switch to tell `prove` where the test classes are loaded; the `-l` tells it where the ordinary classes are loaded (`-l` is the same thing is `-Ilib/`); and then you pass `t/lib/TestsFor/TV/Episode.pm` as the argument to `prove`.

When you do that, you run only the 14 tests in that class. You could have run this:

```
$ prove -l t/test_classes.t
t/test_classes.t .. ok
All tests successful.
Files=1, Tests=32,  3 wallclock secs
Result: PASS
```

And that would have run all your test classes without the other `t/*.t` tests. Instead, you ran `prove -l t/` to run the full test suite. Each test program is run, in alphabetical order, until a total of all 43 tests are run. The `prove` utility makes it easy to run and manage tests.

SUMMARY

In this chapter, you learned the basics of writing tests in Perl. You learned about the standard `Test::More` module and how to use other testing modules such as `Test::Differences`, `Test::Exception`, and `Test::Most`. You learned how to catch exceptions in testing with `eval` and `throws_ok` and how to test for warnings with `Test::Warn`.

You also learned quite a bit about advanced usage of `Test::Class`, an excellent module that is sadly underused in Perl testing. Many of the techniques demonstrated in this chapter are fairly new to many Perl developers (particularly the tricks about auto-loading classes in your `startup` method) and after you master them, you'll have fairly advanced testing skills.

EXERCISES

1. Earlier you had a `unique()` function:

```
sub unique {
    my @array = @_;
    my %hash  = map { $_ => 1 } @array;
    return keys %hash;
}
```

Unfortunately, you wrapped the test in a `TODO:` block because it doesn't do exactly what you want it to do:

```
TODO: {
    local $TODO = 'Figure out how to avoid random order';
    my @have = unique( 2, 3, 5, 4, 3, 5, 7 );
    my @want = ( 2, 3, 5, 4, 7 );
    is_deeply \@have, \@want,
      'unique() should return unique() elements in order';
}
```

Study the test and rewrite the `unique()` function to make the test pass. Remove the `TODO:` block around the test.

2. Usually when you have a failing test, the code is wrong and not the test. Take the example from the first exercise and assume that the code is correct and the test is wrong. How might you fix it?

3. The following test fails. Make it pass.

```
use Test::Most;
use Carp 'croak';
sub reciprocal {
    my $number = shift;
    unless ($number) {
        croak("Illegal division by zero");
    }
    return 1 / $number;
}
throws_ok { reciprocal([]) }
  qr/Argument to reciprocal\(\) must be a number/,
  'Passing non-numbers to reciprocal() should fail';
diag reciprocal([]);
done_testing;
```

4. Extra credit: In the second Try It Out for this chapter, you created a test class for the `TV::Episode` class from Chapter 13. Create a test class for `TV::Episode::Broadcast`. Make sure it inherits from `TestsFor::TV::Episode`.

```
package TV::Episode::Broadcast;
      use Moose;
      use namespace::autoclean;
      extends 'TV::Episode';
      has broadcast_date => ( is => 'ro', isa => 'DateTime', required => 1 );
      __PACKAGE__->meta->make_immutable;
      1;
```

You won't actually need to write any tests for this class if you inherit properly. Because your attributes test method will have two extra tests (one for whether `$broadcast->can ('broadcast_date')` and another for whether it returns the correct value), you can use the following special syntax to make that work:

```
      sub attributes : Tests(+2) {
          my $test = shift;
          $test->SUPER::attributes;
      }
```

That allows you to override the original test method and declare that there are two extra tests (`Tests(+2)`).

Hint: To do this properly, you need to make a small change to `TestsFor::TV::Episode`.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	DESCRIPTION
Testing	The way you ensure your code does what you want it to do.
`Test::More`	The most popular test module for Perl.
`ok()`	Tests if a value is true.
`is()`	Tests if one scalar matches another scalar.
`like()`	Tests if a scalar matches a regular expression.
`is_deeply()`	Tests if one reference contains the same value as another reference.
`SKIP`	Skips tests that cannot be run.
`TODO`	Marks tests as expected failures.
`Test::Differences`	Produces a diff of data structures that don't match.
`Test::Exception`	Easily tests exceptions.
`Test::Warn`	Tests for warnings in your code.
`Test::Most`	Bundles the most common testing functions into one module.
`Test::Class`	Used to write object-oriented tests.