Chapter 11. Perl Modules

If there is a problem to solve, somebody has probably already solved it and made their solution available on the Comprehensive Perl Archive Network (CPAN), which is a worldwide collection of servers and mirrors containing thousands of modules of reusable Perl code. Indeed, most of Perl 5 is in the modules, since Larry designed it as an extensible language.

We're not going to teach you how to write modules here: you'll have to get that from *Intermediate Perl*. In this chapter, we will show you how to use modules that already exist. The idea is to get you started with CPAN rather than give you a survey on modules.

Finding Modules

Modules come in two types: those that come with Perl and that you should already have available to you, and those that you can get from CPAN to install yourself. Unless we say otherwise, the modules that we show come with Perl.

NOTE

Some vendors provide even more modules with their stock versions of Perl. There's actually a third type: vendor modules, but those are a bonus. Check your operating system to see what else you might have.

To find modules that don't come with Perl, start at MetaCPAN. You can also browse the distribution and have a peek at the files without the bother of installing the modules. You can read the module documentation before you download the entire package. There are many other tools for inspecting a distribution too.

But before you go looking for a module, you should check if it is already installed. One way is to just try to read the documentation with *perldoc*.

The `Digest::SHA` module comes with Perl (and we'll use it later), so you should be able to read its documentation:

```
$ perldoc Digest::SHA
```

Try it with a module that does not exist and you'll get an error message:

```
$ perldoc Llamas
No documentation found for "Llamas".
```

The documentation may be available in other formats (such as HTML) on your system too. If the documentation is there, you have the module installed.

The `cpan` command that comes with Perl can give you details on a module:

```
$ cpan -D Digest::SHA
```

Installing Modules

When you want to install a module that you don't already have, sometimes you can simply download the distribution, unpack it, and run a series of commands from the shell. There are two major build systems for Perl distributions, and you use them similarly. Check for a *README* or *INSTALL* file that gives you more information.

If the module uses `ExtUtils::MakeMaker` , which comes with Perl, the sequence will be something like this:

```
$ perl Makefile.PL
$ make install
```

If you can't install modules in the system-wide directories, you can specify another directory with an `INSTALL_BASE` argument to *Makefile.PL*:

```
$ perl Makefile.PL INSTALL_BASE=/Users/fred/lib
```

Some Perl module authors use another module, `Module::Build`, to build and install their creations. That sequence will be something like this:

```
$ perl Build.PL
$ ./Build install
```

As before, you can specify an alternate installation directory:

```
$ perl Build.PL --install_base=/Users/fred/lib
```

Some modules depend on other modules, though, and they won't work unless you install yet more modules. Instead of doing all that work yourself, you can use one of the modules that come with Perl, `CPAN.pm`. From the command line, you can start up the `CPAN.pm` shell from which you can issue commands:

```
$ perl -MCPAN -e shell
```

NOTE

The ".pm" file extension stands for "Perl Module," and some popular modules are pronounced with the ".pm" to distinguish them from something else. In this case, CPAN the archive is different than CPAN the module, so we say "CPAN.pm".

Even this can be a little complicated, so a while ago one of our authors wrote a little script called *cpan*, which also comes with Perl and is usually installed with *perl* and its tools. Just call the script with a list of the modules you want to install:

```
$ cpan Module::CoreList Mojolicious Business::ISBN
```

There's another handy tool, *cpanm* (for *cpanminus*), although it doesn't come with *perl* (yet). It's designed as a zero-conf, lightweight CPAN client that handles most of what people want to do. You can download the single file from *https://cpanmin.us* to get started.

Once you have *cpanm*, you simply tell it which modules you want to install:

```
$ cpanm DBI WWW::Mechanize
```

Using Your Own Directories

One of the common problems with Perl module installation is that by default, the CPAN tools want to install new modules into the same directories where *perl* is. You might not have the proper permissions to create new files in those directories.

The easiest way for beginners to keep additional Perl modules in their own directories is to use `local::lib`, which you'll have to get from CPAN since it doesn't come with *perl* (yet). This module sets the various environment variables that affect where CPAN clients install modules. You can see what they set by loading the module on the command line without anything else:

```
$ perl -Mlocal::lib
export PERL_LOCAL_LIB_ROOT="/Users/fred/perl5";
export PERL_MB_OPT="--install_base /Users/fred/perl5";
export PERL_MM_OPT="INSTALL_BASE=/Users/fred/perl5";
export PERL5LIB="...";
export PATH="/Users/fred/perl5/bin:$PATH";
```

NOTE

We haven't told you about command-line switches yet, but they are all in the perlrun documentation.

The *cpan* client supports this if you use the `-I` switch to install modules:

```
$ cpan -I Set::CrossProduct
```

The *cpanm* tool is a bit smarter. If you've already set the same environment variables `local::lib` would set for you, it uses them. If not, it checks the default module directories for write permissions. If you don't have write permissions, it automatically uses `local::lib` for you.

Advanced users can configure their CPAN clients to install into whatever directories that they like too. You can set this in your `CPAN.pm` configuration so that modules automatically install in your private library directory when you use the `CPAN.pm` shell. You need to configure two settings, one each for the `ExtUtils::MakeMaker` and `Module::Build` systems:

```
$ cpan
cpan> o conf makepl_arg INSTALL_BASE=/Users/fred/perl5
cpan> o conf mbuild_arg "--install_base /Users/fred/perl5"
cpan> o conf commit
```

Notice these are the same settings that `local::lib` created for you in the environment. By setting them in the `CPAN.pm` configuration, it adds them every time it tries to install a module.

Once you've chosen where you want to put your Perl modules, you have to tell your programs where to find them. If you are using `local::lib` , you simply load that module in your program:

```
# inside your Perl program
use local::lib;
```

If you installed them in some other location, you can use the `lib` pragma with a list of additional module directories:

```
# also inside your Perl program
use lib qw( /Users/fred/perl5 );
```

As of v5.26, the current directory is no longer part of the module search path. Prior to that, Perl would look for modules in the current working directory (which might not be where your program is!). If your program changed its working directory, loading more modules would look in that directory instead of where you started. This was a problem with security, so it's no longer there.

Most people probably want to look for modules in the same directory as their program—usually when they have written those modules instead of downloading them. In that case, the `FindBin` module, which comes with Perl, can help. It knows how to find the directory of your program, which you can then use to add your module directory to the search path:

```
use FindBin qw($Bin);
use lib "$Bin/../lib";
```

This is just enough to get you started. We talk much more about this in *Intermediate Perl*, where you also learn to make your own modules. You can also read the entries in the perlfaq8 documentation.

Using Simple Modules

Suppose that you've got a long filename like */usr/local/bin/perl* in your program, and you need to find out the basename without the directory portion. That's easy enough, since the basename is everything after the last slash (it's just *perl* in this case):

```
my $name = "/usr/local/bin/perl";
(my $basename = $name) =~ s#.*/##;
```

As you saw earlier, first Perl will do the assignment inside the parentheses, then it will do the substitution. The substitution is supposed to replace any string ending with a slash (that is, the directory name portion) with an empty string, leaving just the basename. You can even do this with the `/r` switch for the substitution operator:

```
use v5.14;
my $name = "/usr/local/bin/perl";
my $basename = $name =~ s#.*/##r;
```

And if you try these, it seems to work. Well, it *seems* to, but actually, there are three problems.

First, a Unix file or directory name could contain a newline character. (It's not something that's likely to happen by accident, but it's permitted.) So, since the regular expression dot (`.`) can't match a newline, a filename like the string `"/home/fred/ flintstone\n/brontosaurus"` won't work right—that code would think the basename is `"flintstone\n/brontosaurus"`. You could fix that with the `/s` option to the pattern (if you remembered about this subtle and infrequent case), making the substitution look like this: `s#.*/##s`.

The second problem is that this is Unix-specific. It assumes that the forward slash will always be the directory separator, as it is on Unix, and not the backslash or colon that some systems use. Although you might think that your work will never leak out from your Unix-only environment, most useful scripts (and some not so useful) tend to breed in the wild.

And the third (and biggest) problem with this is that we're trying to solve a problem someone else has already solved. Perl comes with a number of modules, which are smart extensions to Perl that add to its functionality. And if those aren't enough, there are many other useful modules available on CPAN, with new ones being added every week. You (or, better yet, your system administrator) can install them if you need their functionality.

In the rest of this section, we'll show you how to use some features of a couple simple modules that come with Perl. (There's more that these modules can do; this is just an overview to illustrate the general principles of how to use a simple module.)

Alas, we can't show you everything you'd need to know about using modules in general, since you'd have to understand advanced topics like ref-

erences and objects in order to use some modules. As we'll see in the next few pages, though, you may be able to use a module that uses objects and references without having to understand those advanced topics. Those topics, including how to create a module, will be covered in great detail in *Intermediate Perl*. But this section should prepare you for using many simple modules. Further information on some interesting and useful modules is included in Appendix B.

The File::Basename Module

In the previous example, you found the basename of a filename in a way that's not portable. Something that seemed straightforward was susceptible to subtle mistaken assumptions (here, the assumption was that newlines would never appear in file or directory names). And you were reinventing the wheel, solving a problem that others have solved (and debugged) many times before you. Not to worry; it happens to all of us.

Here's a better way to extract the basename of a filename. Perl comes with a module called `File::Basename`. With the command *perldoc File::Basename*, or with your system's documentation, you can read about what it does. That's always the first step when using a new module. (It's often the third and fifth steps, as well.)

Soon you're ready to use it, so you declare it with a `use` directive near the top of your program:

```
use File::Basename;
```

NOTE

It's traditional to declare modules near the top of the file since that makes it easy for the maintenance programmer to see which modules you'll be using. That greatly simplifies matters when it's time to install your program on a new machine, for example.

During compilation, Perl sees that line and loads the module. Now it's as if Perl has some new functions that you may use in the remainder of your program. The one we wanted in the earlier example is the `basename` function itself:

```
use File::Basename;

my $name = "/usr/local/bin/perl";
my $basename = basename $name;  # gives 'perl'
```

Well, that worked for Unix. What if your program runs on MacPerl or Windows or VMS, to name a few? There's no problem—this module can tell which kind of machine you're using, and it uses that machine's file-name rules by default. (Of course, you'd have that machine's kind of file-name string in `$name`, in that case.)

There are some related functions also provided by this module. One is the `dirname` function, which pulls the directory name from a full filename. The module also lets you separate a filename from its extension, or change the default set of filename rules.

Using Only Some Functions from a Module

Suppose you discovered that when you went to add the `File::Basename` module to your existing program, you already have a subroutine called `&dirname`—that is, you already have a subroutine with the same name as one of the module's functions. Now there's trouble because the new `dirname` is *also* implemented as a Perl subroutine (inside the module). What do you do?

In your `use` declaration, simply give `File::Basename` an *import list* showing exactly which function names it should give you, and it'll supply those and no others. Here, you'll get nothing but `basename`:

```
use File::Basename qw/ basename /;
```

And here, you ask for no new functions at all:

```
use File::Basename qw/ /;
```

This is also frequently written as an empty set of parentheses:

```
use File::Basename ();
```

Why would you want to do that? Well, this directive tells Perl to load `File::Basename`, just as before, but not to *import* any function names. Importing lets you use the short, simple function names like `basename` and `dirname`. But even if you don't import those names, you can still use the functions. When they're not imported, though, you have to call them by their full names:

```
use File::Basename qw/ /;      # import no function names

my $betty = &dirname($wilma);  # uses your own subroutine &dirname
                               # (not shown)

my $name = "/usr/local/bin/perl";
my $dirname = File::Basename::dirname $name;  # dirname from the module
```

As you see, the full name of the `dirname` function from the module is `File::Basename::dirname`. You can always use the function's full name (once you've loaded the module), whether you've imported the short name `dirname` or not.

Most of the time, you'll want to use a module's default import list. But you can always override that with a list of your own, if you want to leave out some of the default items. Another reason to supply your own list would be if you wanted to import some function not on the default list, since most modules include some (infrequently needed) functions that are not on the default import list.

As you'd guess, some modules will, by default, import more symbols than others. Each module's documentation should make it clear which symbols it exports, if any, but you are always free to override the default import

list by specifying one of your own, just as we did with `File::Basename`. Supplying an empty list imports no symbols.

The File::Spec Module

Now you can find out a file's basename. That's useful, but you'll often want to put that together with a directory name to get a full filename. For example, here you want to take a filename like */home/fred/ice-2.1.txt* and add a prefix to the basename:

```
use File::Basename;

print "Please enter a filename: ";
chomp(my $old_name = <STDIN>);

my $dirname  = dirname $old_name;
my $basename = basename $old_name;

$basename =~ s/^/not/;  # Add a prefix to the basename
my $new_name = "$dirname/$basename";

rename($old_name, $new_name)
    or warn "Can't rename '$old_name' to '$new_name': $!";
```

Do you see the problem here? Once again, you're making the assumption that filenames will follow the Unix conventions and use a forward slash between the directory name and the basename. Fortunately, Perl comes with a module to help with this problem too.

The `File::Spec` module is used for manipulating *file specifications*, which are the names of files, directories, and the other things that are stored on filesystems. Like `File::Basename`, it understands what kind of system it's running on, and it chooses the right set of rules every time. But unlike `File::Basename`, `File::Spec` is an object-oriented (often abbreviated "OO") module.

If you've never caught the fever of OO, don't let that bother you. If you understand objects, that's great; you can use this OO module. If you don't

understand objects, that's OK, too. You just type the symbols as we show you, and it works just as if you knew what you were doing.

In this case, you learn from reading the documentation for `File::Spec` that you want to use a *method* called `catfile`. What's a method? It's just a different kind of function, as far as you're concerned here. The difference is that you'll always call the methods from `File::Spec` with their full names, like this:

```
use File::Spec;

.
.  # Get the values for $dirname and $basename as earlier
.

my $new_name = File::Spec->catfile($dirname, $basename);

rename($old_name, $new_name)
    or warn "Can't rename '$old_name' to '$new_name': $!";
```

As you can see, the full name of a method is the name of the module (called a *class* here), a small arrow (`->`), and the short name of the method. It is important to use the small arrow rather than the double-colon that we used with `File::Basename`.

Since you're calling the method by its full name, though, what symbols does the module import? None of them. That's normal for OO modules. So you don't have to worry about having a subroutine with the same name as one of the many methods of `File::Spec`.

Should you bother using modules like these? It's up to you, as always. If you're sure your program will never be run anywhere but on a Unix machine, say, and you're sure you completely understand the rules for file-names in Unix, then you may prefer to hardcode your assumptions into your programs. But these modules give you an easy way to make your programs more robust in less time—and more portable at no extra charge.

Path::Class

The `File::Spec` module does work with file paths from just about any platform, but the interface is a bit clunky. The `Path::Class` module, which doesn't come with Perl, gives you a more pleasant interface:

```
my $dir     = dir( qw(Users fred lib) );
my $subdir  = $dir->subdir( 'perl5' );     # Users/fred/lib/perl5
my $parent  = $dir->parent;                # Users/fred

my $windir  = $dir->as_foreign( 'Win32' ); # Users\fred\lib
```

Databases and DBI

The `DBI` (Database Interface) module doesn't come with Perl, but it's one of the most popular modules since most people have to connect to a database of some sort. The beauty of `DBI` is that it allows you to use the same interface for just about any database server (or fake server, even), from simple comma-separated value files to enterprise servers such as Oracle. It has ODBC drivers, and some of its drivers are even vendor-supported. To get the full details, check out *Programming the Perl DBI* by Alligator Descartes and Tim Bunce (O'Reilly). You can also take a look at the DBI website.

Once you install `DBI` , you also have to install a DBD (Database Driver). You can get a long list of DBDs from MetaCPAN. Install the right one for your database server, and ensure that you get the version that goes with the version of your server.

The `DBI` is an object-oriented module, but you don't have to know everything about OO programming to use it. You just have to follow the examples in the documentation. To connect to a database, you `use` the `DBI` module, then call its `connect` method:

```
use DBI;

$dbh = DBI->connect($data_source, $username, $password);
```

The `$data_source` contains information particular to the DBD that you want to use, so you'll get that from the DBD. For PostgreSQL, the driver is `DBD::Pg`, and the `$data_source` is something like:

```
my $data_source = "dbi:Pg:dbname=name_of_database";
```

Once you connect to the database, you go through a cycle of preparing, executing, and reading queries:

```
my $sth = $dbh->prepare("SELECT * FROM foo WHERE bla");
$sth->execute();
my @row_ary  = $sth->fetchrow_array;
$sth->finish;
```

When you are finished, you disconnect from the database:

```
$dbh->disconnect();
```

The `DBI` can do all sorts of other things too. See its documentation for more details. Although it's a bit old, *Programming the Perl DBI* is still mostly a good introduction to the module.

Dates and Times

There are many modules that can handle dates and times for you, but the most popular is the `Time::Moment` module from Christian Hansen. It's a nearly complete solution for dates and times. You need to get this module from CPAN.

NOTE

If `Time::Moment` isn't enough for you, check out the `DateTime` module, which is a *complete* solution. It's a bit more heavyweight, but that's the price you pay.

Often, you will have the time as the system (or epoch) time, and you can easily convert that to a `Time::Moment` object:

```
use Time::Moment;
my $dt = Time::Moment->from_epoch( time );
```

Or skip the argument if you want the current time:

```
my $dt = Time::Moment->now;
```

From there, you can access various parts of the date to get what you need:

```
printf '%4d%02d%02d', $dt->year, $dt->month, $dt->day_of_month;
```

If you have two `Time::Moment` objects, you can do date math with them:

```
my $dt1 = Time::Moment->new(
  year       => 1987,
  month      => 12,
  day        => 18,
  );

my $dt2 = Time::Moment->now;

my $years  = $dt1->delta_years( $dt2 );
my $months = $dt1->delta_months( $dt2 ) % 12;

printf "%d years and %d months\n", $years, $months;
```

For those dates, this gives you the output:

```
32 years and 8 months
```

Exercises

See "Answers to Chapter 11 Exercises" for answers to these exercises. Remember, you have to install some modules from CPAN, and part of these exercises require you to research the module by reading its documentation:

1. [15] Install the `Module::CoreList` module from CPAN (if you don't already have it). Print a list of all the modules that came with v5.34. To build a hash whose keys are the names of the modules that came with a given version of *perl*, use this line:

```
my %modules = %{ $Module::CoreList::version{5.034} };
```

2. [20] Write a program using `Time::Moment` to compute the interval between now and a date that you enter as the year and month on the command line:

```
$ perl duration.pl 1960 9
60 years, 2 months
```