10

Modules

In Chapter 8, we divided our programs up into subroutines: functional units that help us organize our program. In this chapter, we'll look at modules, which are the next stage of division.

Very simply, a module is a package within a file. It's a collection of subroutines and variables, all belonging to the same package and stored away in its own file. While subroutines allow us to bundle up individual tasks, modules are more about bundling up an entire area of activity.

Perl comes with quite a large library of modules, which means there are quite a few tasks already coded. That's great for us, since it means we don't have to program them again – we'll be taking a look at some of the more useful ones here. Modules were invented to help code reuse, and recycling your code is an excellent principle, and a good habit to get into.

In fact, reusing ready-made code is such a good idea that there's a whole archive of modules out there, each providing us with a set of subroutines – a set of tools – to work in a different area. Most of the publicly available Perl modules can to be found on CPAN, the 'Comprehensive Perl Archive Network'. Later in the chapter we'll see how to find what we want on CPAN, and how to install modules from there. We'll also have a look at how to use some of the major modules that reside there.

Types of Module

Despite the fact that the modules you'll find out there can be on any subject under the sun, there are just a few standard ways in which they tend to be categorized. In fact, the CPAN Modules List classifies modules by subject area, development stage, where the support comes from, language used, and interface style. We'll use a slightly simpler classification here:

- Pragmatic modules – as we saw in Chapter 9, these alter the way Perl does certain things. Usually, they pass special information to the perl interpreter, which perl then uses internally.
- Standard modules – these make up the majority of modules out there and are purely Perl code. On the whole, they do things in a pretty standard way, which we'll examine later.
- Extension modules combine Perl with C (or other languages) to interface with either the operating system or third-party software.

There's a full list of both the pragmatic and standard modules installed with Perl in Appendix D.

Why Do I Need Them?

Why should you use modules? The simple answer is that it saves time. If you need that program written yesterday, it's exceptionally handy to be able to pull down a bunch of modules that you know will do the job and then simply glue them together. There's not much creativity involved, and you don't learn a great deal doing it that way – but in some cases, there's just not the time for creativity or learning.

The second answer is because programmers are lazy and don't like reinventing the wheel. Now, don't get me wrong – there's good laziness and there's bad laziness. Bad laziness says 'I should get someone else to do this for me', whereas good laziness says 'Maybe someone's already done this.' The good kind pays off. Most of the programming you'll be doing, at some level, has been done before.

There's also the fact that, as we've seen in several cases already, some of the things we want to do are far from straightforward. Unless we really know what we're doing, we run the risk of making incorrect assumptions or overlooking details.

Modules that have been kicking around on CPAN for a while will have been used by thousands of individuals, many of whom will have spent time fixing bugs and returning the results to the maintainer. Most of the borderline cases will have been worked out by now, and you can be pretty confident that the modules will do things correctly. When it comes to parsing HTML or reading CGI forms, I'm perfectly willing to admit that the people who wrote `HTML::Parser` and the `CGI` modules have done more work on the subject that I have – so I use their code, instead of trying to work out my own.

In short: don't reinvent the wheel – use modules.

Including Other Files

A module, as we've mentioned, is just a package stored in a file. We want to get perl to read that file and use it as part of our own program. We have three ways of doing this: `do`, `require` and `use`.

do

This is the most difficult of the three to understand; the others are just slightly varied forms of `do`.

`do` will look for a file by searching the `@INC` path (more on that later). If the file can't be found, it'll silently move on. If it is found, it will run the file just as if it was placed in a block within our main program – but with one slight difference: we won't be able to see lexical variables from the main program once we're inside the additional code. So if we have a file `dothis.plx`:

```
#!/usr/bin/perl
# dothis.plx
use warnings;
use strict;

my $a = "Been there, done that, got the T-shirt";
do "printit.plx";
```

and a file `printit.plx`:

```
print $a;
```

we'll get no output, not even a warning that `$a` is uninitialized within `printit.plx`, because we didn't turn on `warnings` in our included file. On the other hand, we can have subroutines in our included file and call them from the main file.

require

`require` is like `do`, but it'll only `do` once. It'll record the fact that a file has been loaded and will ignore further requests to `require` it again. It also fails with an error if it can't find the file you're loading:

```
#!/usr/bin/perl
# cantload.plx
use warnings;
use strict;

require "nothere.plx";
```

will die with an error like this:

```
>perl cantload.plx
Can't locate nothere.plx in @INC (@INC contains: /usr/local/lib/perl5/5.6.0/cygwin
/usr/local/lib/perl5/5.6.0 /usr/local/lib/perl5/site_perl/5.6.0/cygwin /usr/local/lib/perl5/site_perl/5.6.0
/usr/local/lib/perl5/site_perl .) at cantload.plx line 6.
>
```

This is the `@INC` array, which contains a list of paths where Perl looks for modules and other additional files. The first two paths are where Perl keeps the standard library. The first includes the word `cygwin`, which is the operating system I'm running on and contains the parts of the library specific to this operating system. The second is the part of the standard library, which does not depend on the operating system. In Windows, these two libraries are `C:/Perl/lib` and `C:/Perl/site/lib` by default.

The next two paths are the local 'site' modules, which are third-party modules that we'll install from CPAN or create ourselves. The version number (5.6.0) reminds us that these are modules specific to that version. The next path doesn't have a Perl version number in it, and that's for site modules that do not need a particular version of Perl. Finally, the `.` represents the current directory.

You can also use `require` like this:

```
require Wibble;
```

Using a bareword tells perl to look for a file called `Wibble.pm` in the `@INC` path. It also converts any instance of `::` into a directory separator. For instance, then:

```
require Monty::Python;
```

will send perl looking for `Python.pm` in a directory called `Monty` which is itself in one of the directories given in `@INC`.

use

The way we normally use modules is, logically enough, with the `use` statement. This is like `require`, except that perl applies it *before* anything else in the program starts. If Perl sees a `use` statement *anywhere* in your program, it'll include that module. So, for instance, you can't say this:

```
if ($graphical) {
   use MyProgram::Graphical;
} else {
   use MyProgram::Text;
}
```

because when perl's reading your program, it will include *both* modules – the `use` takes place way before the value of `$graphical` is decided. We say that `use` takes place at compile time and not at run time.

Changing @INC

The default contents of the search path `@INC` are decided when perl is compiled – if we move those directories elsewhere, we'll have to recompile perl to get it working again. However, we can tell it to search in directories other than these. `@INC` is an ordinary array, so you might expect us to be able to say:

```
push @INC, "my/module/directory";
use Wibble;
```

However, this isn't going to work. Why not? Well, remember that the statement above will execute at run time. Unfortunately the `use` statement takes place at compile time, well before that. No problem! There's a special subroutine called `BEGIN,` which is guaranteed execution at compile time, so we can put it there:

```
sub BEGIN {
   push @INC, "my/module/directory";
}
use Wibble;
```

Now that'll work just fine. However, it's a little messy, and what's more, there's an easier way to do it. We can use the `lib` pragma to add our directory to `@INC` before anything else gets a chance to look at it:

```
use lib "my/module/directory";
use Wibble;
```

Package Hierarchies

We've already seen how packages can help us break up a namespace: `$Fred::name` isn't the same variable as `$Barney::name`. When modules come into play, packages are used to identify the module. Now our variables have a nice namespace, but our modules have to identify themselves by a single word. With several thousand modules out there, it gets hard to find the one we want. So the librarians at CPAN have come up with a solution: we split up the module package names into hierarchies. Instead of having tens of modules about sorting, we now have `Sort::Fields`, `Sort::Versions`, and so on.

This hierarchy is only a naming scheme. It doesn't mean that `Sort::Fields` and `Sort::Versions` are somehow related to a bigger package called `Sort` – it's simply a way of making it easier to categorize modules.

So how do we store these in files? Some operating systems won't let us have colons inside file names, so `Sort::Versions.pm` won't be legal. However, since these names represent a consistent hierarchy, there's a natural way we can organize them on the disk: as mentioned above, `require` and `use` translate colons into directory separators, so `Sort::Versions` will actually be stored in a file called `Versions.pm` in a directory called `Sort` somewhere off one of the site paths.

Exporters

Since modules are usually packages stored in a file, a subroutine in the `Text::Wrap` module, for example, would normally be tucked away in the `Text::Wrap` package. However, let's say it would be more convenient for us to have this as a subroutine in the package we're currently in – usually the `main` package. To do this, perl uses a module called `Exporter`, which provides it with a way of **importing** subroutines from the module into the caller's package. Here's how it works:

When you `use` a module, as well as reading and executing the code, perl will try and run a subroutine called `import` inside the module's package. If that's not found, nothing happens, and there's no error. If it is found, though, it's called with all the parameters given on the `use` line. So, for instance:

```
use Wibble ("wobble", "bounce", "boing");
```

loads the `Wibble` module and then runs:

```
Wibble::import("wobble", "bounce", "boing");
```

Theoretically, this `import` subroutine could do anything. In fact, a few modules use it to let you pass parameters to setup the module. However, you'll usually want to use it to import subroutines and variables.

`Exporter` lets the modules that use it borrow a standard `import` subroutine. This subroutine checks a number of variables inside the module as well as the parameters that we give it. If we give an empty list, like this:

```
use Wibble ();
```

then nothing will be imported. If there's a particular subroutine we want to use – `wobble()` for example – then we could call it as `Wibble::wobble()`, and we'll get it imported into our current package. We can only import subroutines that the module is prepared to export, and it'll detail those in a package variable called `@EXPORT_OK`. So if, for instance, I wanted a to make a `Wibble` module from which we could import `wobble()`, `bounce()` and `boing()`, I'd say this:

```
package Wibble;
use warnings;
use strict;
use Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(wobble bounce boing);

sub wobble { print "wobble\n" }
sub bounce { warn  "bounce\n" }
sub boing  { die   "boing!\n" }
```

If we don't pass any parameters at all, we get the default subroutines, which are defined in `@EXPORT`. So if our module looked like this:

```
package Wibble;
use warnings;
use strict;

use Exporter;
our @ISA = qw(Exporter);
our @EXPORT_OK = qw(wobble bounce boing);
our @EXPORT    = qw(bounce);

sub wobble { print "wobble\n" }
sub bounce { warn  "bounce\n" }
sub boing  { die   "boing!\n" }
```

and we ran `use Wibble`; in our main program, we'd be able to call `bounce()` from the main program, but not `wobble()` or `boing()` – we would have to call these as `Wibble::wobble()` and `Wibble::boing()`.

We can also define `tags` with the `%EXPORT_TAGS` hash. This allows us to group together a bunch of subroutines or variables under a group name. For instance, the `CGI` module (which we'll be using in Chapter 12) allows us to say:

```
use CGI qw(:standard);
```

which will import all its most useful subroutines.

The Perl Standard Modules

As we've mentioned, Perl comes with a number of modules included. Some of these (such as `Socket`) are system specific and generally used by higher-level modules – some however, are useful on their own. You can find a list of all the standard modules in Appendix D. We'll take a quick look at some of the more useful and interesting ones here.

File::Find

We looked briefly at `File::Find` when we examined callbacks – we'll see more of these in the final chapter. This is a module for traversing directory trees, visiting each file in turn and running a subroutine (the callback) on them. We have two subroutines, `find` and `finddepth`. The former does a depth-first search (see Chapter 6), visiting directories only after their files have been processed. This is useful if, for example, you want to delete entire directory trees, since you're not usually permitted to delete a directory until you've deleted all the files in it.

Why shouldn't you do this yourself? One of the problems is symbolic links: some operating systems have the ability to point one directory into another, which can create loops in the file system, in which you'll get stuck. The main reason, though, is that it involves a lot of work – work that someone else has done already.

We call the subroutines with two parameters: the callback subroutine reference, and the directory (or a list of directories) to start from:

```
find(\&wanted, "/home/simon/");
```

The subroutine works under the following conditions:

- You are moved into the same directory as the file under consideration.
- The current directory, relative to the top of the tree is held in `$File::Find::dir`.
- `$_` contains the name of the current file.
- `$File::Find::name` is the name including the directory.

With that, we can do anything. Do you remember, way back in Chapter 1, we wanted a program that would delete useless files? Here it is:

```
#!/usr/bin/perl
# hoover.plx
use strict;
use warnings;

use File::Find;
find(\&cleanup, "/");

sub cleanup {
   # Not been accessed in six months?
   if (-A > 180) {
      print "Deleting old file $_\n";
      unlink $_ or print "oops, couldn't delete $_: $!\n";
      return;
   }
   open (FH, $_) or die "Couldn't open $_: $!\n";
   for (1..5) { # You've got five chances.
      my $line = <FH>;
      if ($line =~ /Perl|Simon|important/i) {
         # Spare it.
         return;
      }
   }
   print "Deleting unimportant file $_\n";
   unlink $_ or print "oops, couldn't delete $_: $!\n";
}
```

You can of course alter this so it doesn't look for the words 'Perl', 'Simon' or 'important' in their first five lines and indeed so it doesn't look through and delete files from your entire directory structure.

Getopt::Std

We saw in Chapter 9 how the `-s` flag gave us a rudimentary way to get perl to pass command line options to our program – it will take flags from the command line and let us access them as Perl variables with the same name (for example, `-h` becomes `$h`). However, it had a few limitations:

- We couldn't use `-abc` to mean the `a` flag, the `b` flag and the `c` flag.
- We couldn't give values to flags.
- We had to have all flags as global variables.

The `Getopt::Long` and `Getopt::Std` modules get us round all these problems and provide us with more flexibility besides. `Getopt::Std` is the simpler of the two, providing us with a way to get single-letter switches with values and support for clustered flags. We can also arrange to have the flags placed in a hash. For instance, to provide our wonderful 'Hello World' program (from Chapter 1) with help, a version identifier and (heavens above!) internationalization, we could do this:

```
#!/usr/bin/perl
# hello3.plx
# Hello World (Deluxe)
use warnings;
use strict;

use Getopt::Std;
my %options;
getopts("vhl:",\%options);

if ($options{v}) {
   print "Hello World, version 3.\n";
   exit;
} elsif ($options{h}) {
   print <<EOF;

$0: Typical Hello World program

Syntax: $0 [-h|-v|-l <language>]

   -h : This help message
   -v : Print version on standard output and exit
   -l : Turn on international language support.
EOF
   exit;
} elsif ($options{l}) {
   if ($options{l} eq "french") {
      print "Bonjour, tout le monde.\n";
   } else {
      die "$0: unsupported language\n";
   }
} else {
   print "Hello, world.\n";
}
```

`getopts` takes the following as its arguments: a specification, the letters we want to know about, and a hash reference. If we follow a letter with a colon, we expect that a value will be stored in the hash. If we don't use a colon, then the hash value stored is just true or false depending on whether or not the option was given. We can now get output like this:

```
>perl hello3.plx -l french
Bonjour, tout le monde.
>
```

`Getopt::Std` also produces a warning if it sees options it's not prepared for:

```
>perl hello3.plx -f
Unknown option: f
Hello, world.
>
```

Getopt::Long

The Free Software Foundation, when they were developing the GNU project, decided that single-letter flags weren't friendly enough, so they invented 'long' flags. These use a double minus sign followed by a word. To give a value, you'd say something like `--language=french`.

The module `Getopt::Long` handles this style of options. Its documentation is extremely informative, but it's still useful to see an example. Let's convert the above program to GNU options:

```
#!/usr/bin/perl
# hellolong.plx
# Hello World (Deluxe) - with long flags
use warnings;
use strict;

use Getopt::Long;
my %options;
GetOptions(\%options, "language:s", "help", "version");

if ($options{version}) {
   print "Hello World, version 3.\n";
   exit;
} elsif ($options{help}) {
   print <<EOF;

$0: Typical Hello World program

Syntax: $0 [--help|--version|--language=<language>]

   --help     : This help message
   --version  : Print version on standard output and exit
   --language : Turn on international language support.
EOF
   exit;
} elsif ($options{language}) {
   if ($options{language} eq "french") {
      print "Bonjour, tout le monde.\n";
   } else {
      die "$0: unsupported language\n";
   }
} else {
   print "Hello, world.\n";
}
```

We can still use the previous syntax, but now we can also say:

```
>perl hellolong.plx --language=french
Bonjour, tout le monde.
>
```

File::Spec

If we want to write really portable programs in Perl, we have to be careful when doing things like dealing with file names. `File::Spec` is a module for handling, constructing and breaking apart file names. It's actually installed as an alias to another module: `File::Spec::Unix`, `File::Spec::Win32`, `File::Spec::VMS`, or whatever's relevant to the local system.

Normally it has an object-oriented interface, but it's much easier to use the subroutine interface, `File::Spec::Functions`. Here are some of the subroutines it provides:

Function and Syntax	**Description**
`canonpath (`*$path*`)`	Cleans up *$path* to its simplest form.
`catdir(`*$directory1, $directory2*`)`	Concatenates the two directories together to form a new path to a directory, ensuring an appropriate separator in the middle and removing the separator from the end.
`catfile(`*$directory, $file*`)`	Like `catdir`, but the path will end with a file name.
`tmpdir()`	Finds a writeable directory for temporary files (see the `File::Temp` module before working with temporary files!).
`splitpath($path)`	Splits up a path into volume (drive on Windows, nothing on UNIX), directories and filename.
`splitdir($path)`	Splits a path into its constituent directories: the opposite of `catdir`.
`path()`	Returns the search path for executable files.

So to find out if there's a copy of the `dir` program on this computer, I might do this:

```
#!/usr/bin/perl
# whereisit.plx
use warnings;
use strict;

use File::Spec::Functions;
foreach (path()) {
   my $test = catfile($_,"dir");
   print "Yes, dir is in the $_ directory.\n";
   exit;
}
print "dir was not found here.\n";
```

Benchmark

There's More Than One Way To Do It – that's our motto. However, some ways are always going to be faster than others. How can you tell though? You could analyze each of the statements for efficiency, or you could simply roll your sleeves up and try it out.

Our next module is for testing and timing code. `Benchmark` exports two subroutines: `timethis` and `timethese`, the first of which, `timethis`, is quite easy to use:

```
#!/usr/bin/perl
# benchtest.plx
use warnings;
use strict;

use Benchmark;
my $howmany = 10000;
my $what    = q/my $j=1; for (1..100) {$j*=$_}/;

timethis($howmany, $what);
```

So, we give it some code and a set number of times to run it. Make sure the code is in single quotes so that Perl doesn't attempt to interpolate it. You should, after a little while, see some numbers. These will, of course, vary depending on the speed of your CPU and how busy your computer is, but mine says this:

```
>perl benchtest.plx
timethis 10000:  3 wallclock secs ( 2.58 usr +  0.00 sys =  2.58 CPU) @ 3871.47/s (n=10000)
>
```

This tells us that we ran something 10,000 times, and it took 3 seconds of real time. These seconds were 2.58 spent in calculating ('usr' time) and 0 seconds interacting with the disk (or other non-calculating time). It also tells us that we ran through 3871.47 iterations of the test code each second.

To test several things and weigh them up against each other, we can use `timethese`. Instead of taking a string to represent code to be run, it takes an anonymous hash. The hash keys are names given to sections of the code, and the values are corresponding subroutine references, which we usually create anonymously.

To check the fastest way to read a file from the disk, we could do this:

```
#!/usr/bin/perl
# benchtest2.plx
use warnings;
use strict;

use Benchmark;
my $howmany = 100;

timethese($howmany, {
   line => sub {
      my $file;
      open TEST, "words" or die $!;
      while (<TEST>) { $file .= $_ }
      close TEST;
   },
   slurp => sub {
      my $file;
      local undef $/;
      open TEST, "words" or die $!;
      $file = <TEST>;
      close TEST;
   },
        join => sub {
           my $file;
           open TEST, "words" or die $!;
           $file = join "", <TEST>;
           close TEST;
        }
    });
```

One way reads the file in a line at a time, one slurps the whole file in at once, and one joins the lines together. As you might expect, the slurp method is quite considerably faster:

```
Benchmark: timing 100 iterations of join, line, slurp...
      join: 42 wallclock secs (35.64 usr +  3.78 sys = 39.43 CPU) @  2.54/s (n=100)
      line: 37 wallclock secs (29.77 usr +  3.17 sys = 32.94 CPU) @  3.04/s (n=100)
     slurp:  6 wallclock secs ( 2.87 usr +  2.65 sys =  5.53 CPU) @ 18.09/s (n=100)
```

Also bear in mind that each benchmark will not only time differently between each machine and the next, but often between times you run the benchtest – so *don't* base your life around benchmark tests. If a pretty way to do it is a thousandth of a second slower than an ugly way to do it, choose the pretty one. If speed is really *that* important to you, you should probably be programming in something other than Perl.

Win32

Those familiar with Windows' labyrinthine Win32 APIs will probably want to examine the `libwin32` modules. These all live in the `Win32::` hierarchy (older versions may have some in the `OLE::` hierarchy too, but this was moved to `Win32::OLE::`) and come as standard with ActiveState Perl. If you've compiled another Perl yourself on Windows, you can get a copy of the modules from CPAN – we'll see how in a second.

These modules, which give you access to such things as Semaphores, Services, OLE, the Clipboard, and a whole bunch of other things besides, will probably be of most interest to existing Windows programmers. For the rest of us though, there are two modules that will be of particular use:

Win32::Sound

The first, `Win32::Sound`, lets us play with the sound subsystem – we can play `.wav` files, set the speaker volume, and so on. We can also use it to play the standard system sounds.

Try It Out : Playing .wav Files

The following program will play all the .wav files in the current directory:

```
#!/usr/bin/perl
# wavplay.plx
use warnings;
use strict;
use Win32::Sound;

my $wav;
Win32::Sound::Volume(65535);
opendir (DIR, ".") or die "Couldn't open directory: $!";
while ($wav = readdir(DIR)) {
   Win32::Sound::Play($wav);
}
```

You won't see any output, but if you're in a directory containing `.wav` files, you should certainly be able to hear some!

How It Works

The `Win32::Sound` module provides us with a number of subroutines:

Function	Description
`Win32::Sound::Volume(`*$left, $right*`)`	Sets the left and right speaker volumes to the requested amount. If only `$left` is given, both speakers are set to that volume. If neither is given, the current volume is returned. You can give the volume either as a percentage or a number from 0 to 65535.
`Win32::Sound::Play(`*$name*`)`	Plays the named sound file, or the named system sound. (for example, `SystemStart`)
`Win32::Sound::Format(`*$filename*`)`	Returns information about the format of the given sound file.
`Win32::Sound::Devices()`	Lists all the available sound-related devices on the system.
`Win32::Sound::DeviceInfo(`*$device*`)`	Provides information on the given sound device.

You can get a full list of the subroutines from the `Win32::Sound` documentation page if you have the module installed.

Win32::TieRegistry

Windows uses a centralized system database to store information about applications, users and its own state. This is called the **registry**, and we can get at it by using Perl's `Win32::TieRegistry` module. This just provides a convenient layer around the `Win32::Registry` module that is rather more technical in nature. `Win32::TieRegistry` transforms the Windows registry into a Perl hash.

The registry is a complicated beast, and revolves around a hierarchical tree structure like a hash of hashes or a directory. For instance, information about users' software is stored under `HKEY_CURRENT_USER\Microsoft\Windows\CurrentVersion\`. Now we can get to this particular part of the hash by saying the following:

```
#!/usr/bin/perl
# registry.plx
use warnings;
use strict;
use Win32::TieRegistry (Delimiter => "/") ;
```

We load the module, and change the delimiter from a backslash to a forward slash so we don't end up drowning in a sea of backslashes:

```
my $users = $Registry->
  {HKEY_CURRENT_USER/Software/Microsoft/Windows/CurrentVersion/};
```

Now we've got that key, we can dig further into the depths of the registry. This is where the Windows Explorer tips are stored:

```
my $tips = $users->{Explorer/Tips};
```

and from there we can add our own tips:

```
$tips->{/186} = "It's easy to use Perl as a Registry editor with the
Win32::TieRegistry module.";
```

We can always delete them again, using ordinary hash techniques:

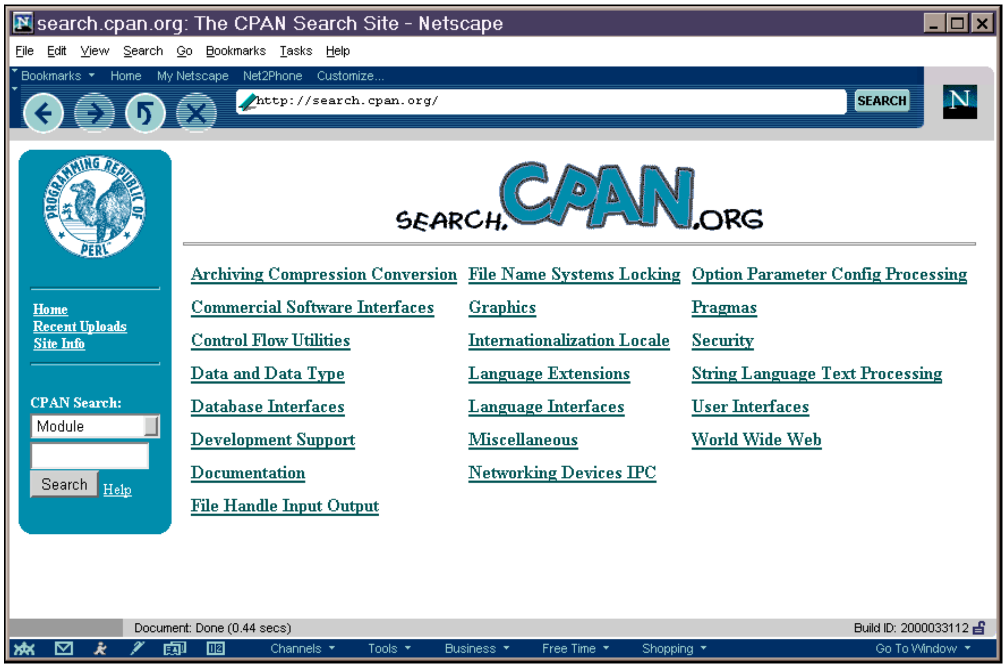
```
delete $tips->{/186};
```

Again, if you're after more information, it's available in the `Win32::TieRegistry` documentation, but I'd suggest you lay off reading that until you've digested the following chapter on object oriented Perl.

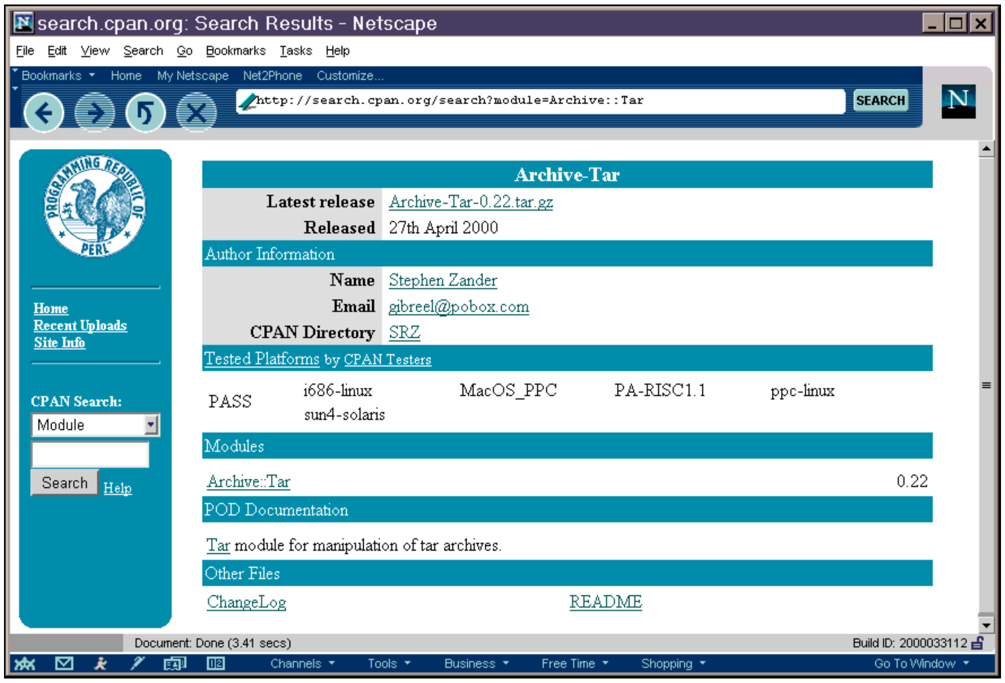
CPAN

So far we've been looking at standard modules provided with most perl distributions. However, as we mentioned in the introduction, there's also a central repository for Perl modules – collections of code that will do virtually any kind of job: the **Comprehensive Perl Archive Network**, or CPAN, which you can find on the web at http://www.cpan.org.

So before you ask 'how do I do...?' or start plugging away at any long task, it's always worth taking a quick look here to see if it's already been done. CPAN is searchable in plenty of different ways – the most common are by keyword, by topic, or by module name. There are also a few CPAN search engines, but the easiest for browsing is probably the web-based CPAN search engine at http://search.cpan.org/. Alternatively, if you know what you're looking for, http://theory.uwinnipeg.ca/search/cpan-search.html is rather good too.


search.cpan.org: The CPAN Search Site - Netscape
File Edit View Search Go Bookmarks Tasks Help
Bookmarks Home My Netscape Net2Phone Customize...
http://search.cpan.org/
SEARCH
SEARCH.CPAN.ORG
Home
Recent Uploads
Site Info
CPAN Search:
Module
Search
Help
Archiving Compression Conversion
Commercial Software Interfaces
Control Flow Utilities
Data and Data Type
Database Interfaces
Development Support
Documentation
File Handle Input Output
File Name Systems Locking
Graphics
Internationalization Locale
Language Extensions
Language Interfaces
Miscellaneous
Networking Devices IPC
Option Parameter Config Processing
Pragmas
Security
String Language Text Processing
User Interfaces
World Wide Web
Document: Done (0.44 secs)
Build ID: 2000033112
Channels Tools Business Free Time Shopping Go To Window

This lets us look up modules by category, as well as searching for words in the modules' documentation. Once we've found a module that might do what we want, we follow a link to get further information on it and get ourselves a download. For example, this is what we get for the `Archive::Tar` module:



search.cpan.org: Search Results - Netscape
File Edit View Search Go Bookmarks Tasks Help
Bookmarks Home My Netscape Net2Phone Customize...
http://search.cpan.org/search?module=Archive::Tar
SEARCH
Home
Recent Uploads
Site Info
CPAN Search:
Module
Search
Help
Document: Done (3.41 secs)
Build ID: 2000033112
Channels Tools Business Free Time Shopping Go To Window

Archive-Tar

Latest release	Archive-Tar-0.22.tar.gz
Released	27th April 2000

Author Information

Name	Stephen Zander
Email	gibreel@pobox.com
CPAN Directory	SRZ

Tested Platforms by CPAN Testers

PASS	i686-linux	MacOS_PPC	PA-RISC1.1	ppc-linux
	sun4-solaris			

Modules

Archive::Tar 0.22

POD Documentation

Tar module for manipulation of tar archives.

Other Files

ChangeLog README

Now that we've seen how to find the modules we want, we're ready to look at the various ways in which we can install them.

Installing Modules with PPM

If you're using ActivePerl, module installation is made very simple by the **Perl Package Manager** (PPM). This is a useful little tool that's provided along with installations of ActivePerl, which allows us to install modules from the command line with the minimum of effort.

So without further ado, let's install the `Net::FTP` module. This is part of the 'libnet' bundle, a collection of modules which all relate to networking and which we'll be seeing again in Chapters 12 and 14. Installation is quite simple – it involves installing the libnet bundle of which `Net::FTP` is a part.

1. Type **ppm** at the command line, this will give you the PPM prompt: `PPM>`
2. Now type **install libnet** – you may be asked to confirm your request. If so type **y**.
3. If you have libnet already installed, you may be asked if you would like to modify/update your libnet configuration. You don't have to, so just type **no**.
4. Exit the PPM prompt by typing **quit**, and now you have `Net::FTP` installed, ready for the next couple of chapters.

Installing a Module Manually

We'll now take a look at what's involved in doing the same installation for ourselves. If you search CPAN for the module `Net::FTP`, you should find yourself looking at the file `libnet-1.0702.tar.gz` (unless there's a newer version out by the time you read this…) Download and unpack this file. On UNIX systems, `gzip -dc libnet-1.0702.tar.gz | tar -xvf` should do the trick, while you can use Winzip to extract these files on Windows.

Note that you will encounter serious problems with the following procedures if you're running on Windows9x. This being the case, I'd suggest sticking with PPM if you have the option. For Windows NT and Windows 2000 users however, read on.

Every module should contain a `Makefile.PL`, which can be used to generate the instructions to install the module. Let's run that file first:

```
>perl Makefile.PL
Checking for Socket...ok
Checking for IO::Socket...ok
Checking if your kit is complete...
Looks good
Writing Makefile for Bundle::Libnet
>
```

If you can't install in Perl's site directories because you don't have the appropriate permissions, run:

```
>perl Makefile.PL PREFIX=/my/module/path
```

and it will arrange for the module to be installed there. You should then be sure to add that directory to `@INC` in all of your programs that will be making use of it. You can do this in three ways:

- use the `lib` pragma (as described above)
- use the `-I` flag at the command line
- modify the `PERL5LIB` environment variable

`Makefile.PL` first checks that we have all the modules it requires, and then that we've got everything we should have in the module archive itself – a file called `MANIFEST` contains a list of what should be in the archive. As we saw in step 3 of installing Libnet with PPM, we get asked for information concerning our networking configuration. The default option for each question is in brackets, and we'll get that if we press return. You can answer them as best you can or leave the defaults intact and once it's done with the questions, we'll finally see the message:

```
Writing Makefile for Net
```

Now we're ready to type `make` – assuming, of course, we have `make` on our system.

Windows users can download `nmake` from http://download.microsoft.com/download/vc15/Patch/1.52/W95/EN-US/Nmake15.exe. Just put it in one of your path directories and redirect the following calls from make to nmake.

We run make, and it creates a directory called blib, holding all the files that it'll eventually copy to the real installation directory:

```
>make
mkdir blib
mkdir blib/lib
mkdir blib/arch
mkdir blib/arch/auto
mkdir blib/arch/auto/Net
mkdir blib/lib/auto
mkdir blib/lib/auto/Net
```

We run it again, and it now copies the files it needs to there:

```
>make
cp Net/Config.pm blib/lib/Net/Config.pm
cp Net/Domain.pm blib/lib/Net/Domain.pm
cp Net/SMTP.pm blib/lib/Net/SMTP.pm
cp Net/DummyInetd.pm blib/lib/Net/DummyInetd.pm
cp Net/Time.pm blib/lib/Net/Time.pm
cp Net/NNTP.pm blib/lib/Net/NNTP.pm
cp Net/FTP/dataconn.pm blib/lib/Net/FTP/dataconn.pm
cp Net/PH.pm blib/lib/Net/PH.pm
cp Net/FTP/A.pm blib/lib/Net/FTP/A.pm
cp Net/FTP/I.pm blib/lib/Net/FTP/I.pm
cp libnet.cfg blib/lib/Net/libnet.cfg
cp Net/POP3.pm blib/lib/Net/POP3.pm
…
```

Once that's done, we check to see if our module's working:

```
>make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl -Iblib/arch -Iblib/lib -I/usr/local/lib/pe
rl5/5.6.0/cygwin -I/usr/local/lib/perl5/5.6.0 -e 'use Test::Harness qw(&runtests
 $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/ftp..............ok
t/hostname..........ok
t/nntp.............ok
t/ph...............ok
t/require...........ok
t/smtp.............ok
All tests successful
Files=6, Tests=11, 10 wallclock secs ( 4.25 cusr +  4.19 csys =  8.44 CPU)
>
```

Finally, we actually install it, moving the files from blib to the correct location, as stored in @INC:

```
>make install
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/Cmd.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/Config.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/Domain.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/DummyInetd.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/FTP.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/Net/libnet.cfg
…
```

Hooray! The module's now installed.
However, there's a much, much easier way of doing it.

The CPAN Module

Another easy way to navigate and install modules from CPAN is to use the standard module called `CPAN`. The 'CPAN Shell' is an extremely powerful tool for finding, downloading, building and installing modules.

Again, note that you will encounter serious problems with the following procedures if you're running on Windows9x. Windows NT/2000 users note also that this routine really doesn't like spaces in directory paths, but there is a fix, as follows:

In Windows Explorer, go to `your_perl_install_directory\lib\CPAN`, and open the file `Config.pm` in your Perl editor. This is the CPAN module's systemwide configuration file and contains all the information the module needs to run. Scan down the list and if any of the paths to files contain spaces, you'll need to change them to their 8.3 format.

For example, let's say your copy of nmake.exe can be found at `C:\Program Files\Microsoft Visual Studio\vc98\bin\nmake.exe` because you previously installed it with Visual C++. Unfortunately, this path has been copied to `Config.pm` which means the module itself looks for `nmake` in `C:\Program` and of course doesn't find it. By changing the make entry in `Config.pm` from:

```
'make' => q[C:\Program Files\Microsoft Visual Studio\VC98\bin\nmake.EXE],
```

to

```
'make' => q[C:\Progra~1\Micros~3\VC98\bin\nmake.EXE],
```

the problem is solved.

Try It Out : Using the CPAN module

To get into the CPAN shell, put:

```
>perl –MCPAN –e shell
```

This is actually just the same as saying:

```
#!/usr/bin/perl
use CPAN;
shell();
```

The whole shell is actually a function in the (massively complex) `CPAN` module. The first time we run it, we'll see something like this:

```
/usr/local/lib/perl5/5.6.0/CPAN/Config.pm initialized.
```

```
CPAN is the world-wide archive of perl resources. It consists of about
100 sites that all replicate the same contents all around the globe.
Many countries have at least one CPAN site already. The resources
found on CPAN are easily accessible with the CPAN.pm module. If you
want to use CPAN.pm, you have to configure it properly.

If you do not want to enter a dialog now, you can answer 'no' to this
question and I'll try to autoconfigure. (Note: you can revisit this
dialog anytime later by typing 'o conf init' at the cpan prompt.)

Are you ready for manual configuration? [yes]
```

Press the *Enter* key, and you'll be asked a series of questions about your computer and the nearest CPAN server. If you don't know, just keep hitting Enter through the answers. Eventually, you'll end up at a prompt like this:

```
cpan shell -- CPAN exploration and modules installation (v1.52)
ReadLine support available (try "install Bundle::CPAN")

cpan>
```

Now we're ready to issue commands. The `install` command, as shown in the prompt, will download and install a module. For example, we could install the `DBD::mysql` module by simply saying

```
cpan>install DBD::mysql
```

Alternatively, we could get information on a module with the `i` command. Let's get some information on the MLDBM module, another module that we'll look into when we investigate databases:

```
cpan>i MLDBM
Module id = MLDBM
   DESCRIPTION  Transparently store multi-level data in DBM
   CPAN_USERID  GSAR (Gurusamy Sarathy <gsar@ActiveState.com>)
   CPAN_VERSION 2.00
   CPAN_FILE    GSAR/MLDBM-2.00.tar.gz
   DSLI_STATUS  RdpO (released,developer,perl,object-oriented)
   INST_FILE    (not installed)
```

So what does this tell us? Well, the module is called MLDBM, and there's a description of it. It was written by the CPAN author GSAR, who translates to Gurusamy Sarathy in the real world. It's at version 2.00, and it's stored on CPAN in the directory `GSAR/MLDBM-2.00.tar.gz`.

The funny little code thing is the CPAN classification, which we mentioned in the introduction. It tells us this module has been released (the implication being that it's been released for a while), that you should contact the developer if you need any support on it, that it's written purely in Perl without any extensions in C, and that it's object-oriented – and finally, that we don't have it installed. So let's install it:

```
cpan> install MLDBM
```

In fact, you don't even have to go into the shell to install a module. As well as exporting the `shell` subroutine, `CPAN` provides us with `install`, with which we can simply say **perl -MCPAN -e 'install "MLDBM"'** *to produce the same results.*

You'll then see a few lines that will be specific to your computer. Different systems have different ways of downloading files and depend on whether or not you have the external programs `lynx`, `ftp`, or `ncftp` or the Perl `Net::FTP` module installed.

The `CPAN` module will download the file. Then, if you've got the `Digest::MD5` module installed, it will download a special file called a checksum – which provides a summary of that file so we make sure that what we've downloaded is what's on the server.

```
Checksum for /home/simon/.cpan/sources/authors/id/GSAR/MLDBM-2.00.tar.gz ok
```

Next, it'll unpack the file:

```
MLDBM-2.00/
MLDBM-2.00/Makefile.PL
MLDBM-2.00/t/
MLDBM-2.00/t/storable.t
MLDBM-2.00/t/freezethaw.t
MLDBM-2.00/t/dumper.t
MLDBM-2.00/lib/
MLDBM-2.00/lib/MLDBM/
MLDBM-2.00/lib/MLDBM/Serializer/
MLDBM-2.00/lib/MLDBM/Serializer/Storable.pm
MLDBM-2.00/lib/MLDBM/Serializer/FreezeThaw.pm
MLDBM-2.00/lib/MLDBM/Serializer/Data/
MLDBM-2.00/lib/MLDBM/Serializer/Data/Dumper.pm
MLDBM-2.00/lib/MLDBM.pm
MLDBM-2.00/Changes
MLDBM-2.00/README
MLDBM-2.00/MANIFEST
```

and then it will generate and run the Makefile:

```
CPAN.pm: Going to build GSAR/MLDBM-2.00.tar.gz

Checking if your kit is complete...
Looks good
Writing Makefile for MLDBM
mkdir blib
mkdir blib/lib
mkdir blib/arch
mkdir blib/arch/auto
mkdir blib/arch/auto/MLDBM
mkdir blib/lib/auto
mkdir blib/lib/auto/MLDBM
cp lib/MLDBM/Serializer/FreezeThaw.pm blib/lib/MLDBM/Serializer/FreezeThaw.pm
cp lib/MLDBM.pm blib/lib/MLDBM.pm
cp lib/MLDBM/Serializer/Storable.pm blib/lib/MLDBM/Serializer/Storable.pm
cp lib/MLDBM/Serializer/Data/Dumper.pm blib/lib/MLDBM/Serializer/Data/Dumper.pm
  /usr/local/bin/make  -- OK
```

Once that's successful, it'll test the module out:

```
Running make test
PERL_DL_NONLAZY=1 /usr/local/bin/perl -Iblib/arch -Iblib/lib -I/usr/local/lib/perl5/5.6.0/cygwin -
I/usr/local/lib/perl5/5.6.0 -e 'use Test::Harness qw(&runtests
 $verbose); $verbose=0; runtests @ARGV;' t/*.t
t/dumper............ok
t/freezethaw........skipped test on this platform
t/storable..........skipped test on this platform
All tests successful, 2 tests skipped.
Files=3, Tests=4,  4 wallclock secs ( 1.29 cusr +  1.52 csys =  2.81 CPU)
  /usr/local/bin/make test -- OK
```

and finally install it:

```
Running make install
Installing /usr/local/lib/perl5/site_perl/5.6.0/MLDBM.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/MLDBM/Serializer/FreezeThaw.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/MLDBM/Serializer/Storable.pm
Installing /usr/local/lib/perl5/site_perl/5.6.0/MLDBM/Serializer/Data/Dumper.pm
Writing /usr/local/lib/perl5/site_perl/5.6.0/cygwin/auto/MLDBM/.packlist
Appending installation info to /usr/local/lib/perl5/5.6.0/cygwin/perllocal.pod
  /usr/local/bin/make install  -- OK

cpan>
```

Successfully installed, and with the minimum of effort!

How about if we don't actually know the name of the module we're looking for? Well, `CPAN` lets us use a regular expression match to locate modules. For instance, if we're about to do some work involving MIDI electronic music files, we could search for 'MIDI'. This is what we might see:

```
cpan>i /MIDI/
Distribution    F/FO/FOOCHRE/MIDI-Realtime-0.01.tar.gz
Distribution    S/SB/SBURKE/MIDI-Perl-0.75.tar.gz
Module          MIDI            (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Event     (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Opus      (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Realtime  (F/FO/FOOCHRE/MIDI-Realtime-0.01.tar.gz)
Module          MIDI::Score     (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Simple    (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
Module          MIDI::Track     (S/SB/SBURKE/MIDI-Perl-0.75.tar.gz)
```

'Distributions' are archive files: zips or tar.gz files containing one or more Perl modules. We see that `MIDI-Realtime` contains just the `MIDI::Realtime` module, and Sean Burke's `MIDI-Perl` contains a few more modules, so perhaps we'd check that one out.

Bundles

Some modules depend on other modules being installed. For instance, the `Win32::TieRegistry` module needs `Win32::Registry` to do the hard work of getting at the registry. If you're downloading packages from CPAN manually, you'll have to try each package, find out what's missing and download another repeatedly until you've got everything you need. The `CPAN` module does a lot of this work for you. It can detect dependencies in packages and download and install everything that's missing.

This is fine for making sure that things work, but as well as *needing* other modules, some merely *suggest* other modules. For instance, the `CPAN` module itself works fine with nothing other than what's in the core, but if you have `Term::Readline` installed, it gives you a much more flexible prompt, with tab-completion, a command history meaning you can use the up and down arrows to scroll through previous commands, and other niceties.

Enter **bundles** – collections of packages that go well together. The CPAN bundle, `Bundle::CPAN`, for instance, contains various modules that make the CPAN shell easier to use: `Term::ReadLine` as we mentioned above, `Digest::MD5` for security checking the files downloaded, some `Net::` modules to make network communication with the CPAN servers nicer, and so on.

We'll now look here at two particularly useful bundles, which contain modules that I personally wouldn't go *anywhere* without.

Bundle::LWP

`Bundle::LWP` contains modules for *everything* to do with the Web. It has modules for dealing with HTML, HTTP, MIME types, handling URLs, downloading and mirroring remote web sites, creating web spiders and robots more advanced than our earlier webchecker program in Chapter 8, and so on.

The main chunk of the bundle is the `LWP` (`libwww-perl`) distribution, containing the modules for getting remote web sites. Back in Chapter 8, I mentioned that you could use `LWP::Simple` to get a `get` subroutine. Let's have a look at what else it gives us.

This module will export five subroutines to our current package.

- The **`get`** subroutine does exactly what we were previously trying to do with `lynx` – fetch a web site and return you the underlying HTML. However, this subroutine knows all about proxies, error codes, and the other things that we didn't properly check for:

```
$file = get("http://www.perl.com/");
```

- The **`head`** subroutine fetches the header of the site and returns a few headers: what type of document the page is (it's usually going to be `text/html`) how big it is in bytes, when it was last modified, when it should be regarded as old (these are both UNIX times suitable for feeding to `localtime`) and what the server has to say about itself. Some servers may not return all these headers:

```
($content_type, $document_length, $modified_time, $expires, $server) =
  head("http://www.perl.com/");
```

The next three routines are all quite similar in that they all involve retrieving an HTML page.

- The first, **`getprint`**, retrieves the HTML file and then prints it out to standard output – useful if you're redirecting to a file or using a filter as some sort of HTML formatter. You can copy a web page to a local file like this:

```
getprint("http://www.perl.com/");
```

- Alternatively, you can use the **`getstore`** subroutine to store it to a file:

```
perl -MLWP::Simple -e
   'getprint("http://www.perl.com/")' > perlpage.html
```

- Finally, **`mirror`** is like `getstore`, except it checks to see if the remote site's page is newer than the one we've already got:

```
perl -MLWP::Simple -e
   'getstore("http://www.perl.com/","perlpage.html")'
```

`Bundle::LWP` functions as a standard exporter, so, as we saw earlier in the chapter, we can prevent any of the five being pulled into our namespace by saying use `LWP::Simple()` – if you've already got a subroutine called `get` defined, this may be a good idea for example. Be sure to read the main `LWP` documentation and the `lwpcook` page which contains a few ideas for things to do with `LWP`.

Bundle::libnet

Similarly, `Bundle::libnet` contains a bunch of stuff for dealing with the network, although it's not nearly as big as LWP. The modules in `Bundle::libnet` and its dependencies allow you to use FTP, telnet, SMTP mail, and other network protocols. These are object-oriented modules, and we'll be looking at them some more in the next chapter.

Submitting Your Own Module to CPAN

CPAN contains almost everything you'll ever need. Almost. There'll surely come a day when you're faced with a problem where no known module can help you. If you think it's a sufficiently general problem that other people are going to come across, why not consider making your solution into a module and submitting it to CPAN? Think of it as a way of giving something back to the community that gave you all this…

Seriously, if you do have something you think would be useful to others, there are a few things you need to do to get it to CPAN. It will require a reasonable amount of work, and you may want to leave it until you've finished this book and had a look at the next book in our series, Professional Perl (*Due out October 2000 at time of publication -Ed*). Here's what you should do, though:

- Check it's not been written before. Search CPAN. Look at the main modules list at http://www.cpan.org/modules/00modlist.long.html – there's also a lot of good advice about how to lay out and prepare your module there.
- Read the `perlmod` and `perlmodlib` documentation pages until you really understand them.
- Learn about the `Carp` module, and use `carp` and `croak` instead of `warn` and `die`.
- Learn about the `Test` module and how to produce test suites for modules.
- Learn about documenting your modules in POD, Plain Old Documentation.
- Look at the source to a few simple modules like `Text::Wrap` and `Text::Tabs` to get a feel of how modules are written.
- Take a deep breath, and issue the following command:

```
>h2xs -AXn Your::Module::Name
```

- Edit the files produced, remembering to create a test suite and provide really good documentation.
- Run **perl Makefile.PL** and then **make**.

Your module's now ready to ship!

Summary

Modules save you time. Modules do things well. In essence, a module is just a package stored in a file, which we load with the `use` statement. We can load in other Perl code from files, using `require` or `do`, but `use` also calls the module's `import` subroutine, so that modules that want to can use `Exporter` to move subroutines into our package.

Perl provides a number of standard modules. You can check out the full list in Appendix D, and you can get documentation on each and every one by running `perldoc`. We looked briefly at `File::Find` (for examining files in directory trees), the `Getopt` modules (for reading options from the command line), the `File::Spec::Functions` module (for portable filename handling), the `Benchmark` module (for timing and testing code), and the `Win32` modules (for access to the Windows system and registry).

CPAN is the Comprehensive Perl Archive Network. It's a repository of good code. You can search it from http://search.cpan.org/ or use the Perl module `CPAN` for easy searching and installation. The `CPAN` module has the advantage of knowing about file dependencies and can therefore download and install files in the correct order.

Bundles provide sets of related modules. We looked at `LWP::Simple` (from the `libwww` bundle), and we'll look more at the `libnet` bundle in our chapter on networking. Finally, we looked at some of what's involved in abstracting your code and putting it into a module.