Appendix D. Experimental Features

You can completely skip this appendix and the experimental features we show and not suffer for it. Or you can blindly follow the examples we show in the chapters and not worry about what is happening. But we think you'll want to use them and understand them because we want to use and understand them too.

Many of the new features in Perl aren't really "new." They're *experimental*. You have to do something to enable them, they might change, and they might disappear altogether. In fact, v5.24 removed two experimental features.

This is quite clever. People can install the latest *perl* and start using these new features. They can test them, see how they interact with other features, and best of all, develop unexpected idioms for them. Or they can completely ignore them and not worry about backward compatibility. The Perl 5 Porters, the people who develop and maintain the Perl code base, get to see how people use and react to a feature before they commit to making it permanent.

Learning Perl should show you the best and most exciting ways of working in Perl, but we also don't want you to rely on experimental features that might disappear a year after you buy this book. We show you some of the new features, but when we do, we point to this appendix so you can get the background we don't want to explain each time.

The `feature` module documentation lists most of the new features and gives a brief description of their use. You can also read the perldelta documentation in each release of Perl to learn about new developments. We show the state of most new features in Table D-1. Before we tell you about that, we give you some background.

A Short History of Perl Development

Perl has gone through several eras of development, each with its own story. Knowing what's come before can help you appreciate where Perl is now.

In the late 1980s, Larry Wall created the Perl language (although that was not the first name he tried to use). He mostly worked on his own with some feedback from the Usenet community. Kids today probably have never seen a newsgroup, but it was the social media of the time, and that's where he first released Perl in 1987.

Eventually Perl was interesting enough to have books about it (this book in particular, and *Programming perl*, with pink covers before our publisher moved Perl books to blue covers). Perl was promoted to version 4. That's also around the time Perl's popularity exploded, and when many people learned (or stopped learning) Perl. Quite frankly, it's this era that set most of the world's expectations of Perl, and more disappointedly, Perl programmers. But that's spilt milk.

But Perl 4 didn't have object-oriented features, good ways to make complex data structures, or lexical scoping. Around 1993, Larry started working on Perl 5, the current major version and the one we write about in this book.

To move from Perl 4 to Perl 5, a gang of Perlers created the Perl 5 Porters to ensure that Perl 5 was ported to hundreds of different platforms. Today the group is still there, although the people have changed. You can read more about their process in perlpolicy.

The perlhist documentation lists each Perl release, including its date and maintainer. After Larry released Perl 5.0 in 1994, other people took responsibility for some releases. After a new version release, other people typically came in to maintain the old version. It was a bit haphazard and ad hoc, but it worked for a while.

The Porters made big changes in Perl 5.6, and again in Perl 5.8. Perl was going through some growing pains, including the switch to handling Unicode. From Perl 5.004 to 5.005 was a little over a year, but from 5.005

to 5.6 was almost two years. The lag time from Perl 5.6 to 5.8 was over two years. However, the time between releases was increasing.

NOTE

Notice how differently we write the version numbers between 5.005 and v5.6. A Perler who has been around for a while will say “five double-oh five” but “five point six” for those. We started talking about the second (or *minor*) number as the release version. It’s a quirk of history.

After Perl 5.8, people knew the code needed drastic changes for continued development. Chip Salzenberg tried rewriting Perl in C++ in a secret project he had dubbed “Topaz.” It didn’t work out, although he learned interesting lessons in the process. Around the same time, Larry and a few others had the idea to start Perl 6, a complete rewrite of the code base to allow easier development and modern features.

Perl 5.10 and Beyond

Now we’re going to ignore half of the fork that happens at this point, and rather than spark historical debates, we’ll merely write that Perl 6 (now called “Raku”) did not become the next major version of Perl. It became a mostly separate language in its own right, but that’s another book, *Learning Perl 6*. For a few years it did distract some people from Perl 5 development, but then, suddenly, Perl 5 resuscitated itself. At the end of 2007, over five years since the previous Perl 5 release, Rafael Garcia-Suarez released v5.10. This release had some features stolen from the ongoing Perl 6 development, mostly `say` (Chapter 5), `state` (Chapter 4), `given-when`, and smart matching (those last two are experimental features we have since removed from this book).

Larry had moved on to Perl 6 development. For the first time, Larry was not in charge of Perl 5 development. Jesse Vincent stepped up to take on that role and started putting a post-Larry process in order, including a regular release cycle for development versions and yearly releases for stable versions.

Ricardo Signes later took over for Jesse and put more policies into place. New features would start as “experimental” features until they had proven themselves. After two stable releases, a new feature could move to a permanent feature. These experimental features won’t disturb your program if you don’t enable them, so you have backward compatibility at the same time. Or you can enable them at your own risk if you want to play with the latest stuff.

The Perl 5 Porters applied the same process to removing features. Perl has several warts (admit it, we all know it does) and a list of deprecated features and variables. Did you know that there was a variable to control the starting index of an array? You didn’t? Don’t worry, it’s gone now (but it’s not, because there’s an experimental feature that restores it). Through the new process, Perl marks a feature deprecated and warns about its use. After two stable releases (so, two years), the Porters could remove the feature safe in the knowledge that they’d given abundant warnings. And they are actually removing features now. They still support backward compatibility, but within reason.

NOTE

You can read the official Perl support policy in perlpolicy. Basically, the Porters offer official support for the previous two stable releases. If v5.34 is the latest release, they support v5.34 and v5.32 officially. They might update v5.30 or some other earlier version at their discretion.

Suppose you have something that needs the old features. What do you do? Simple—keep the old *perl* around! It’s the same *perl* you’ve been using for years and no one is taking it away from you. Oh, you are using the system *perl* and your system wants to upgrade it? Well, now you know one of the reasons you shouldn’t rely on the system *perl*. That’s for the system, not you! Install your own *perl* for your important applications.

NOTE

Installing your own *perl* not only guards you from the whims of your system upgrades, but it can also be faster. The system *perl* isn't tuned for your use. It's compiled to be the least hassle for everyone. If you don't want features such as debugging systems or threads, you can compile your own *perl* without those for a small speedup. You could even compile another *perl* with those features.

Installing a Recent Perl

Before you think about installing a new Perl, check if the one that you have is good enough. The `-v` command-line switch tells you which version you have:

```
$ perl -v

This is perl 5, version 34, subversion 0 (v5.34.0)
```

If you have a recent enough version, you don't have to do any more work. What you do next depends on how much work you'd like to do.

If you are stuck on a system without a compiler, you can try precompiled versions, including Strawberry Perl (Windows) or ActivePerl Community Edition (macOS, Windows, Linux, and others).

You can compile your own *perl*. We think that everyone should try compiling it themselves at least once in their lives. Part of being a programmer is understanding how actual computers work, and compiling source, managing libraries, and such are part of that. You can download the *perl* source from CPAN. We tend to have them all installed so we can play with any version we like.

NOTE

You may have to install development tools to compile code. We can't tell you exactly what that is since systems have so many ways of managing that. Figure out how to install *gcc* (the GNU C compiler) and you'll probably get the rest of the tools you need. For macOS users, install the "Command Line Tools for Xcode" from Apple. Cygwin for Windows provides a Unix-like environment.

Once you unpack the source, you can configure the installation to tell it where you want to install it. You don't need special privileges to do this since you can install it into any directory you control:

```
$ ./Configure -des -Dprefix=/path/where/you/want/perl
```

We like to install several *perl*s, so we create version-specific directories for each one:

```
$ ./Configure -des -Dprefix=/usr/local/perls/perl-5.34.0
```

From there, tell *make* to install it—this might take a bit:

```
$ make install
```

You might want to test the result before you install it. If the test step fails, *make* will not run the install step:

```
$ make test install
```

Once installed, we can use the new *perl* by specifying it in the shebang line:

```
#!/usr/local/perls/perl-5.34.0/bin/perl
```

You can also use the *perlbrew* application to install and manage several Perls. It's doing the same thing you did in the previous step, but automated. See http://perlbrew.pl for details.

Experimental Features

Let's get down to the actual features and how you use them. We aren't going to list every feature or fully explain the features we highlight. We want to show you how to use any new feature, not particular new features.

You can enable experimental features in several ways. The first is the `-E` command-line switch, introduced in v5.10. Like the `-e` switch, it specifies the program as an argument, but `-E` also enables all new features:

```
$ perl -E "say q(Hello World)"
```

Inside a program, you can enable new features with `use` and the version number in any format:

```
use v5.34;
use 5.34.0;
use 5.034;
```

Remember that since v5.12, specifying the version with `use` also implicitly turns on `strict`.

You can also specify the minimum version without loading new features; do it with `require`:

```
require v5.34;
```

The `feature` module allows you to load features when you want them. In the `use` example, Perl was doing this for you by implicitly calling it to load the tag associated with that version:

```
use feature qw(:5.10);
```

Instead of loading all the new features for a particular version, you can load them individually. In Chapter 4 we showed you `state` (a stable feature that showed up in v5.10) and `signatures` (an experimental feature introduced in v5.20):

```
use feature qw(state signatures);
```

If you want to turn off all new features, perhaps because you have an old script that doesn't work with the newer Perls, you can disable them:

```
no feature qw(:all);
```

Of course, you need a version of Perl that has the feature module for this to work. That probably means that you are running an old program on a new *perl*, or perhaps you're still getting used to the new features and don't want to accidentally use them.

NOTE

We don't cover the complexities of `no` in this book, but it's the opposite of `use`. You're actually *un*-importing something.

Turning Off Experimental Warnings

Having turned on some experimental features, you'll get some warnings when you use those features. *perl* doesn't emit the warning when you enable the features. This simple program enables `signatures` but has no warnings:

```
use v5.20;
use feature qw(signatures);
```

This program uses a subroutine signature:

```
use v5.20;
use feature qw(signatures);

sub division ( $m, $n ) {
  eval { $m / $n }
}
```

You get the warning even though you don’t call the subroutine:

```
The signatures feature is experimental at features.pl line 4.
```

To turn off these warnings, prefix the feature name with `experimental::`, like this:

```
no warnings qw(experimental::signatures);
```

If you want to turn off all experimental warnings, leave off the name of the feature:

```
no warnings qw(experimental);
```

Starting with v5.18, the `experimental` pragma enables the feature and disables its warnings in one step. This is a bit tidier:

```
use experimental qw(current_sub);
```

Enable or Disable Features Lexically

If you are a bit skittish about experimental features, you can enable them lexically and give them the smallest (or largest) scope that makes you comfortable.

Here’s a program where you’ve defined your own version of `say`. Perhaps you did that before v5.10 existed. You want to add some new code to use the built-in version of `say`. The `feature` pragma enables it only for the block where you declare it:

```
require v5.10;
sub say {
  print "Someone said \"@_\"\n";
}

say( "Hello Fred!" );

{ # use the built-in say in here
use feature qw(say);
say "Hello Barney!";
}

say( "Hello Dino!" );
```

The output shows you using both versions in the same program:

```
Someone said "Hello Fred!"
Hello Barney!
Someone said "Hello Dino!"
```

This also means you have to enable the features per file since Perl treats a file as a scope as if it had virtual braces around it. However, you'll have to keep reading in *Intermediate Perl* to learn more about multifile programs.

Don't Rely on Experimental Features

Experimental features are bright and shiny, novel, and expectedly attractive. But they might disappear and we don't quite know what they are going to do in the next version.

For code that won't make it to the outside world (even if that world is outside your group but still in your company), experiment all that you want. Remember, however, that virtually everything leaks out even if you try to contain it. You might have to rip out the shiny bits when the Porters decide to remove those experimental features.

If you know your code is destined for the outside world, realize that experimental features require a recent version of Perl. As much as everyone might wish that everyone used the latest version of Perl, we know that's not the case. If your creation is exciting enough, people may be motivated to migrate. The rest will complain that they are limited by local policy. You can't win.

No matter which situation you are in, try the experimental features. Learn what they do, see how they work, and tell people what you've found. That's why they are there for you, and feedback helps the Porters fix or adjust their behavior.

Table D-1 provides a breakdown of major new features and the version of *perl* that you will need.

Table D-1. Perl's new features

Feature	Introduced in	Experimental	Stable in	Documented in	Covered in
`array_base`	v5.10		v5.10	perlvar	
`bitwise`	v5.22		5.28	perlop	Chapter 12
`current_sub`	v5.16		v5.20	perlsub	
`declared_refs`	v5.26	✓		perlref	
`evalbytes`	v5.16		v5.20	perlfunc	
`fc`	v5.16		v5.20	perlfunc	
`isa`	v5.32	✓		perlfunc	
`lexical_subs`	v5.18		5.26	perlsub	
`postderef`	v5.20		v5.24	perlref	
`postderef_qq`	v5.20		v5.24	perlref	
`refaliasing`	v5.22	✓		perlref	

Feature	Introduced in	Experimental	Stable in	Documented in	Covered in
regex_sets	v5.18	✓		perlrecharclass	
say	v5.10		v5.10	perlfunc	Chapter 5
signatures	v5.20	✓		perlsub	Chapter 4
state	v5.10		v5.10	perlfunc, perlsub	Chapter 4
switch	v5.10	✓		perlsyn	
try-catch	v5.34	✓		perlsyn	
unicode_eval	v5.16			perlfunc	
unicode_strings	v5.12			perlunicode	
vlb	v5.30	✓		perlre	