NAME

version - Perl extension for Version Objects

SYNOPSIS

```
  # Parsing version strings (decimal or dotted-decimal)

  use version 0.77; # get latest bug-fixes and API
  $ver = version->parse($string)

  # Declaring a dotted-decimal $VERSION (keep on one line!)

  use version 0.77; our $VERSION = version->declare("v1.2.3"); # formal
  use version 0.77; our $VERSION = qv("v1.2.3");               # shorthand
  use version 0.77; our $VERSION = qv("v1.2_3");               # alpha

  # Declaring an old-style decimal $VERSION (use quotes!)

  our $VERSION = "1.0203";                                     #
recommended
  use version 0.77; our $VERSION = version->parse("1.0203");   # formal
  use version 0.77; our $VERSION = version->parse("1.02_03");  # alpha

  # Comparing mixed version styles (decimals, dotted-decimals, objects)

  if ( version->parse($v1) == version->parse($v2) ) {
    # do stuff
  }

  # Sorting mixed version styles

  @ordered = sort { version->parse($a) <=> version->parse($b) } @list;
```

DESCRIPTION

Version objects were added to Perl in 5.10. This module implements version objects for older version of Perl and provides the version object API for all versions of Perl. All previous releases before 0.74 are deprecated and should not be used due to incompatible API changes. Version 0.77 introduces the new 'parse' and 'declare' methods to standardize usage. You are strongly urged to set 0.77 as a minimum in your code, e.g.

```
  use version 0.77; # even for Perl v.5.10.0
```

TYPES OF VERSION OBJECTS

There are two different types of version objects, corresponding to the two different styles of versions in use:

Decimal Versions

The classic floating-point number $VERSION. The advantage to this style is that you don't need to do anything special, just type a number into your source file. Quoting is recommended, as it ensures that trailing zeroes ("1.50") are preserved in any warnings or other output.

Dotted Decimal Versions

The more modern form of version assignment, with 3 (or potentially more) integers separated by

decimal points (e.g. v1.2.3). This is the form that Perl itself has used since 5.6.0 was released. The leading "v" is now strongly recommended for clarity, and will throw a warning in a future release if omitted.

DECLARING VERSIONS

If you have a module that uses a decimal $VERSION (floating point), and you do not intend to ever change that, this module is not for you. There is nothing that version.pm gains you over a simple $VERSION assignment:

```
our $VERSION = "1.02";
```

Since Perl v5.10.0 includes the version.pm comparison logic anyways, you don't need to do anything at all.

How to convert a module from decimal to dotted-decimal

If you have used a decimal $VERSION in the past and wish to switch to a dotted-decimal $VERSION, then you need to make a one-time conversion to the new format.

Important Note: you must ensure that your new $VERSION is numerically greater than your current decimal $VERSION; this is not always obvious. First, convert your old decimal version (e.g. 1.02) to a normalized dotted-decimal form:

```
$ perl -Mversion -e 'print version->parse("1.02")->normal'
v1.20.0
```

Then increment any of the dotted-decimal components (v1.20.1 or v1.21.0).

How to declare() a dotted-decimal version

```
use version 0.77; our $VERSION = version->declare("v1.2.3");
```

The `declare()` method always creates dotted-decimal version objects. When used in a module, you **must** put it on the same line as "use version" to ensure that $VERSION is read correctly by PAUSE and installer tools. You should also add 'version' to the 'configure_requires' section of your module metadata file. See instructions in *ExtUtils::MakeMaker* or *Module::Build* for details.

Important Note: Even if you pass in what looks like a decimal number ("1.2"), a dotted-decimal will be created ("v1.200.0"). To avoid confusion or unintentional errors on older Perls, follow these guidelines:

- Always use a dotted-decimal with (at least) three components
- Always use a leading-v
- Always quote the version

If you really insist on using version.pm with an ordinary decimal version, use `parse()` instead of declare. See the *PARSING AND COMPARING VERSIONS* for details.

See also *version::Internals* for more on version number conversion, quoting, calculated version numbers and declaring developer or "alpha" version numbers.

PARSING AND COMPARING VERSIONS

If you need to compare version numbers, but can't be sure whether they are expressed as numbers, strings, v-strings or version objects, then you should use version.pm to parse them all into objects for comparison.

How to parse() a version

The `parse()` method takes in anything that might be a version and returns a corresponding version object, doing any necessary conversion along the way.

- Dotted-decimal: bare v-strings (v1.2.3) and strings with more than one decimal point and a leading 'v' ("v1.2.3"); NOTE you can technically use a v-string or strings with a leading-v and only one decimal point (v1.2 or "v1.2"), but you will confuse both yourself and others.
- Decimal: regular decimal numbers (literal or in a string)

Some examples:

```
$variable   version->parse($variable)
---------   -------------------------
1.23        v1.230.0
"1.23"      v1.230.0
v1.23       v1.23.0
"v1.23"     v1.23.0
"1.2.3"     v1.2.3
"v1.2.3"    v1.2.3
```

See *version::Internals* for more on version number conversion.

How to check for a legal version string

If you do not want to actually create a full blown version object, but would still like to verify that a given string meets the criteria to be parsed as a version, there are two helper functions that can be employed directly:

`is_lax()`

The lax criteria corresponds to what is currently allowed by the version parser. All of the following formats are acceptable for dotted-decimal formats strings:

```
v1.2
1.2345.6
v1.23_4
1.2345
1.2345_01
```

`is_strict()`

If you want to limit youself to a much more narrow definition of what a version string constitutes, `is_strict()` is limited to version strings like the following list:

```
v1.234.5
2.3456
```

See *version::Internals* for details of the regular expressions that define the legal version string forms, as well as how to use those regular expressions in your own code if `is_lax()` and `is_strict()` are not sufficient for your needs.

How to compare version objects

Version objects overload the `cmp` and <=> operators. Perl automatically generates all of the other comparison operators based on those two so all the normal logical comparisons will work.

```
if ( version->parse($v1) == version->parse($v2) ) {
  # do stuff
}
```

If a version object is compared against a non-version object, the non-object term will be converted to a version object using `parse()`. This may give surprising results:

```
$v1 = version->parse("v0.95.0");
$bool = $v1 < 0.96; # FALSE since 0.96 is v0.960.0
```

Always comparing to a version object will help avoid surprises:

```
$bool = $v1 < version->parse("v0.96.0"); # TRUE
```

Note that "alpha" version objects (where the version string contains a trailing underscore segment) compare as less than the equivalent version without an underscore:

```
$bool = version->parse("1.23_45") < version->parse("1.2345"); # TRUE
```

See *version::Internals* for more details on "alpha" versions.

OBJECT METHODS

is_alpha()

True if and only if the version object was created with a underscore, e.g.

```
version->parse('1.002_03')->is_alpha;  # TRUE
version->declare('1.2.3_4')->is_alpha; # TRUE
```

is_qv()

True only if the version object is a dotted-decimal version, e.g.

```
version->parse('v1.2.0')->is_qv;        # TRUE
version->declare('v1.2')->is_qv;       # TRUE
qv('1.2')->is_qv;                      # TRUE
version->parse('1.2')->is_qv;          # FALSE
```

normal()

Returns a string with a standard 'normalized' dotted-decimal form with a leading-v and at least 3 components.

```
version->declare('v1.2')->normal;  # v1.2.0
version->parse('1.2')->normal;     # v1.200.0
```

numify()

Returns a value representing the object in a pure decimal form without trailing zeroes.

```
version->declare('v1.2')->numify;  # 1.002
version->parse('1.2')->numify;     # 1.2
```

stringify()

Returns a string that is as close to the original representation as possible. If the original representation was a numeric literal, it will be returned the way perl would normally represent it in a string. This method is used whenever a version object is interpolated into a string.

```
version->declare('v1.2')->stringify;    # v1.2
version->parse('1.200')->stringify;     # 1.200
version->parse(1.02_30)->stringify;     # 1.023
```

EXPORTED FUNCTIONS

qv()

This function is no longer recommended for use, but is maintained for compatibility with existing code. If you do not want to have it exported to your namespace, use this form:

```
use version 0.77 ();
```

is_lax()

(Not exported by default)

This function takes a scalar argument and returns a boolean value indicating whether the argument meets the "lax" rules for a version number. Leading and trailing spaces are not allowed.

is_strict()

(Not exported by default)

This function takes a scalar argument and returns a boolean value indicating whether the argument meets the "strict" rules for a version number. Leading and trailing spaces are not allowed.

AUTHOR

John Peacock <jpeacock@cpan.org>

SEE ALSO

version::Internals.

perl.