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!)

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 (without quotes) into your source file.

Dotted Decimal Versions

The more modern form of version assignment, with 3 (or potentially more) integers seperated 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.

See *VERSION OBJECT DETAILS* for further information.

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 OBJECT DETAILS* 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 can 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 OBJECT DETAILS* for more on version number conversion.

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
```

VERSION OBJECT DETAILS

Equivalence between Decimal and Dotted-Decimal Versions

When Perl 5.6.0 was released, the decision was made to provide a transformation between the old-style decimal versions and new-style dotted-decimal versions:

```
5.6.0    == 5.006000
5.005_04 == 5.5.40
```

The floating point number is taken and split first on the single decimal place, then each group of three digits to the right of the decimal makes up the next digit, and so on until the number of significant digits is exhausted, **plus** enough trailing zeros to reach the next multiple of three.

This was the method that version.pm adopted as well. Some examples may be helpful:

```
                          equivalent
decimal    zero-padded    dotted-decimal
-------    -----------    --------------
1.2        1.200          v1.200.0
1.02       1.020          v1.20.0
1.002      1.002          v1.2.0
```

```
1.0023     1.002300       v1.2.300
1.00203    1.002030       v1.2.30
1.002003   1.002003       v1.2.3
```

Quoting rules

Because of the nature of the Perl parsing and tokenizing routines, certain initialization values **must** be quoted in order to correctly parse as the intended version, especially when using the *declare* or *qv* methods. While you do not have to quote decimal numbers when creating version objects, it is always safe to quote **all** initial values when using version.pm methods, as this will ensure that what you type is what is used.

Additionally, if you quote your initializer, then the quoted value that goes **in** will be be exactly what comes **out** when your $VERSION is printed (stringified). If you do not quote your value, Perl's normal numeric handling comes into play and you may not get back what you were expecting.

If you use a mathematic formula that resolves to a floating point number, you are dependent on Perl's conversion routines to yield the version you expect. You are pretty safe by dividing by a power of 10, for example, but other operations are not likely to be what you intend. For example:

```
$VERSION = version->new((qw$Revision: 1.4)[1]/10);
print $VERSION;          # yields 0.14
$V2 = version->new(100/9); # Integer overflow in decimal number
print $V2;               # yields something like 11.111.111.100
```

Perl 5.8.1 and beyond are able to automatically quote v-strings but that is not possible in earlier versions of Perl. In other words:

```
$version = version->new("v2.5.4");  # legal in all versions of Perl
$newvers = version->new(v2.5.4);    # legal only in Perl >= 5.8.1
```

What about v-strings?

There are two ways to enter v-strings: a bare number with two or more decimal points, or a bare number with one or more decimal points and a leading 'v' character (also bare). For example:

```
$vs1 = 1.2.3; # encoded as \1\2\3
$vs2 = v1.2;  # encoded as \1\2
```

However, the use of bare v-strings to initialize version objects is **strongly** discouraged in all circumstances. Also, bare v-strings are not completely supported in any version of Perl prior to 5.8.1.

If you insist on using bare v-strings with Perl > 5.6.0, be aware of the following limitations:

1) For Perl releases 5.6.0 through 5.8.0, the v-string code merely guesses, based on some characteristics of v-strings. You **must** use a three part version, e.g. 1.2.3 or v1.2.3 in order for this heuristic to be successful.

2) For Perl releases 5.8.1 and later, v-strings have changed in the Perl core to be magical, which means that the version.pm code can automatically determine whether the v-string encoding was used.

3) In all cases, a version created using v-strings will have a stringified form that has a leading 'v' character, for the simple reason that sometimes it is impossible to tell whether one was present initially.

Alpha versions

For module authors using CPAN, the convention has been to note unstable releases with an underscore in the version string. (See *CPAN*.) version.pm follows this convention and alpha releases

will test as being newer than the more recent stable release, and less than the next stable release. For dotted-decimal versions, only the last element may be separated by an underscore:

```
# Declaring
use version 0.77; our $VERSION = version->declare("v1.2_3");

# Parsing
$v1 = version->parse("v1.2_3");
$v1 = version->parse("1.002_003");
```

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 ();
```

AUTHOR

John Peacock <jpeacock@cpan.org>

SEE ALSO

version::Internal.

perl.