NAME

namespace::clean – Keep imports and functions out of your namespace

SYNOPSIS

```
  package Foo;
  use warnings;
  use strict;

  use Carp qw(croak);   # 'croak' will be removed

  sub bar { 23 }        # 'bar' will be removed

  # remove all previously defined functions
  use namespace::clean;

  sub baz { bar() }     # 'baz' still defined, 'bar' still bound

  # begin to collection function names from here again
  no namespace::clean;

  sub quux { baz() }    # 'quux' will be removed

  # remove all functions defined after the 'no' unimport
  use namespace::clean;

  # Will print: 'No', 'No', 'Yes' and 'No'
  print +(__PACKAGE__->can('croak') ? 'Yes' : 'No'), "\n";
  print +(__PACKAGE__->can('bar')   ? 'Yes' : 'No'), "\n";
  print +(__PACKAGE__->can('baz')   ? 'Yes' : 'No'), "\n";
  print +(__PACKAGE__->can('quux')  ? 'Yes' : 'No'), "\n";

  1;
```

DESCRIPTION

Keeping packages clean

When you define a function, or import one, into a Perl package, it will naturally also be available as a method. This does not per se cause problems, but it can complicate subclassing and, for example, plugin classes that are included via multiple inheritance by loading them as base classes.

The `namespace::clean` pragma will remove all previously declared or imported symbols at the end of the current package's compile cycle. Functions called in the package itself will still be bound by their name, but they won't show up as methods on your class or instances.

By unimporting via `no` you can tell `namespace::clean` to start collecting functions for the next `use namespace::clean;` specification.

You can use the `-except` flag to tell `namespace::clean` that you don't want it to remove a certain function or method. A common use would be a module exporting an `import` method along with some functions:

```
  use ModuleExportingImport;
  use namespace::clean -except => [qw( import )];
```

If you just want to `-except` a single sub, you can pass it directly. For more than one value you have to use an array reference.

Late binding caveat

Note that the technique used by this module relies on perl having resolved all names to actual code references during the compilation of a scope. While this is almost always what the interpreter does, there are some exceptions, notably the sort SUBNAME style of the `sort` built-in invocation. The following example will not work, because `sort` does not try to resolve the function name to an actual code reference until **runtime**.

```
 use MyApp::Utils 'my_sorter';
 use namespace::clean;

 my @sorted = sort my_sorter @list;
```

You need to work around this by forcing a compile-time resolution like so:

```
 use MyApp::Utils 'my_sorter';
 use namespace::clean;

 my $my_sorter_cref = \&my_sorter;

 my @sorted = sort $my_sorter_cref @list;
```

Explicitly removing functions when your scope is compiled

It is also possible to explicitly tell `namespace::clean` what packages to remove when the surrounding scope has finished compiling. Here is an example:

```
  package Foo;
  use strict;

  # blessed NOT available

  sub my_class {
      use Scalar::Util qw( blessed );
      use namespace::clean qw( blessed );

      # blessed available
      return blessed shift;
  }

  # blessed NOT available
```

Moose

When using `namespace::clean` together with Moose you want to keep the installed `meta` method. So your classes should look like:

```
  package Foo;
  use Moose;
  use namespace::clean -except => 'meta';
  ...
```

Same goes for Moose::Role.

Cleaning other packages

You can tell `namespace::clean` that you want to clean up another package instead of the one importing. To do this you have to pass in the `-cleanee` option like this:

```
  package My::MooseX::namespace::clean;
  use strict;

  use namespace::clean (); # no cleanup, just load

  sub import {
      namespace::clean->import(
        -cleanee => scalar(caller),
        -except  => 'meta',
      );
  }
```

If you don't care about `namespace::cleans` discover–and–`-except` logic, and just want to remove subroutines, try "clean_subroutines".

METHODS

clean_subroutines

This exposes the actual subroutine-removal logic.

```
namespace::clean->clean_subroutines($cleanee, qw( subA subB ));
```

will remove `subA` and `subB` from `$cleanee`. Note that this will remove the subroutines **immediately** and not wait for scope end. If you want to have this effect at a specific time (e.g. `namespace::clean` acts on scope compile end) it is your responsibility to make sure it runs at that time.

import

Makes a snapshot of the current defined functions and installs a B::Hooks::EndOfScope hook in the current scope to invoke the cleanups.

unimport

This method will be called when you do a

```
no namespace::clean;
```

It will start a new section of code that defines functions to clean up.

get_class_store

This returns a reference to a hash in a passed package containing information about function names included and excluded from removal.

get_functions

Takes a class as argument and returns all currently defined functions in it as a hash reference with the function name as key and a typeglob reference to the symbol as value.

IMPLEMENTATION DETAILS

This module works through the effect that a

```
delete $SomePackage::{foo};
```

will remove the `foo` symbol from `$SomePackage` for run time lookups (e.g., method calls) but will leave the entry alive to be called by already resolved names in the package itself. `namespace::clean` will restore and therefor in effect keep all glob slots that aren't `CODE`.

A test file has been added to the perl core to ensure that this behaviour will be stable in future releases.

Just for completeness sake, if you want to remove the symbol completely, use `undef` instead.

SEE ALSO

B::Hooks::EndOfScope

THANKS

Many thanks to Matt S Trout for the inspiration on the whole idea.

AUTHORS

- Robert 'phaylon' Sedlacek <rs@474.at>
- Florian Ragwitz <rafl@debian.org>
- Jesse Luehrs <doy@tozt.net>
- Peter Rabbitson <ribasushi@cpan.org>
- Father Chrysostomos <sprout@cpan.org>

COPYRIGHT AND LICENSE