NAME

Exporter::Easy − Takes the drudgery out of Exporting symbols

SYNOPSIS

In module YourModule.pm:

```
package YourModule;
use Exporter::Easy (
  OK => [ '$munge', 'frobnicate' ] # symbols to export on request
);
```

In other files which wish to use YourModule:

```
use ModuleName qw(frobnicate);      # import listed symbols
frobnicate ($left, $right)          # calls YourModule::frobnicate
```

DESCRIPTION

Exporter::Easy makes using Exporter easy. In its simplest case, it allows you to drop the boilerplate code that comes with using Exporter, so

```
require Exporter;
use base qw( Exporter );
use vars qw( @EXPORT );
@EXPORT = ( 'init' );
```

becomes

```
use Exporter::Easy ( EXPORT => [ 'init' ] );
```

and more complicated situations where you use tags to build lists and more tags become easy, like this

```
use Exporter::Easy (
      EXPORT => [qw( init :base )],
      TAGS => [
              base => [qw( open close )],
              read => [qw( read sysread readline )],
              write => [qw( print write writeline )],
              misc => [qw( select flush )],
              all => [qw( :base :read :write :misc)],
              no_misc => [qw( :all !:misc )],
      ],
      OK => [qw( some other stuff )],
);
```

This will set `@EXPORT`, `@EXPORT_OK`, `@EXPORT_FAIL` and `%EXPORT_TAGS` in the current package, add Exporter to that package's `@ISA` and do a `use vars` on all the variables mentioned. The rest is handled as normal by Exporter.

HOW TO USE IT

Put

```
use Exporter::Easy ( KEY => value, ...);
```

in your package. Arguments are passes as key-value pairs, the following keys are available

TAGS

The value should be a reference to a list that goes like (TAG_NAME, TAG_VALUE, TAG_NAME, TAG_VALUE, ...), where TAG_NAME is a string and TAG_VALUE is a reference to an array of symbols and tags. For example

```
TAGS => [
  file => [ 'open', 'close', 'read', 'write'],
  string => [ 'length', 'substr', 'chomp' ],
  hash => [ 'keys', 'values', 'each' ],
  all => [ ':file', ':string', ':hash' ],
  some => [':all', '!open', ':hash'],
]
```

This is used to fill the `%EXPORT_TAGS` in your package. You can build tags from other tags − in

the example above the tag `all` will contain all the symbols from `file`, `string` and `hash`. You can also subtract symbols and tags − in the example above, `some` contains the symbols from all but with `open` removed and all the symbols from `hash` removed.

The rule is that any symbol starting with a ':' is taken to be a tag which has been defined previously (if it's not defined you'll get an error). If a symbol is preceded by a '!' it will be subtracted from the list, otherwise it is added.

If you try to redefine a tag you will also get an error.

All the symbols which occur while building the tags are automatically added your package's `@EXPORT_OK` array.

OK The value should be a reference to a list of symbols and tags (which will be exapanded). These symbols will be added to the `@EXPORT_OK` array in your package. Using OK and and OK_ONLY together will give an error.

OK_ONLY
The value should be a reference to a list of symbols and tags (which will be exapanded). The `@EXPORT_OK` array in your package will contains only these symbols.. This totally overrides the automatic population of this array. If you just want to add some symbols to the list that Exporter::Easy has automatically built then you should use OK instead. Using OK_ONLY and OK together will give an error.

EXPORT
The value should be a reference to a list of symbol names and tags. Any tags will be expanded and the resulting list of symbol names will be placed in the `@EXPORT` array in your package. The tag created by the ALL key is not available at this stage.

FAIL
The value should be a reference to a list of symbol names and tags. The tags will be expanded and the resulting list of symbol names will be placed in the `@EXPORT_FAIL` array in your package. They will also be added to the `@EXPORT_OK` list.

ALL
The value should be the name of tag that doesn't yet exist. This tag will contain a list of all symbols which can be exported.

ISA If you set this to 0 then Exporter will not be added to your `@ISA` list.

VARS
If this is set to 1 or not provided then all $, @ and % variables mentioned previously will be available to use in your package as if you had done a `use vars` on them. If it's set to a reference to a list of symbols and tags then only those symbols will be available. If it's set to 0 then you'll have to do your own `use vars` in your package.

PROCESSING ORDER

We need take the information provided and build `@EXPORT`, `@EXPORT_OK`, `@EXPORT_FAIL` and `%EXPORT_TAGS` in the calling package. We may also need to build a tag with all of the symbols and to make all the variables useable under strict.

The arguments are processed in the following order: TAGS, EXPORT, OK, OK_ONLY and FAIL, ALL, VARS and finally ISA. This means you cannot use the tag created by ALL anywhere except in VARS (although vars defaults to using all symbols anyway).

SEE ALSO

Exporter is the grandaddy of all Exporter modules, and bundled with Perl itself, unlike the rest of the modules listed here. Look at the documentation for this module to see more explanation of the OK, EXPORT and other variables.

Attribute::Exporter defines attributes which you use to mark which subs and variables you want to export, and how.

Exporter::Simple also uses attributes to control the export of functions and variables from your module.

Const::Exporter makes it easy to create a module that exports constants.

Constant::Exporter is another module that makes it easy to create modules that define and export constants.

Sub::Exporter is a ‘‘sophisticated exporter for custom-built routines’’; it lets you provide generators that can be used to customise what gets imported when someone uses your module.

Exporter::Tiny provides the same features as Sub::Exporter, but relying only on core dependencies.

Exporter::Shiny is a shortcut for Exporter::Tiny that provides a more concise notation for providing optional exports.

Exporter::Declare provides syntactic sugar to make the export status of your functions part of their declaration. Kind of.

AppConfig::Exporter lets you export part of an AppConfig−based configuration.

Exporter::Lexical lets you export lexical subs from your module.

Constant::Exporter::Lazy lets you write a module that exports function-style constants, which are instantiated lazily.

Exporter::Auto will export everything from your module that it thinks is a public function (name doesn’t start with an underscore).

Class::Exporter lets you export class methods as regular subroutines.

Xporter is like Exporter, but with persistent defaults and auto-ISA.

REPOSITORY

<https://github.com/neilb/Exporter−Easy>

AUTHOR

Written by Fergal Daly <fergal AT esatclear DOT ie>.

LICENSE

Under the same license as Perl itself