## NAME

Exporter - Implements default import method for modules

## SYNOPSIS

In module *YourModule.pm*:

```
package YourModule;
require Exporter;
@ISA = qw(Exporter);
@EXPORT_OK = qw(munge frobnicate);  # symbols to export on request
```

or

```
package YourModule;
use Exporter 'import'; # gives you Exporter's import() method directly
@EXPORT_OK = qw(munge frobnicate);  # symbols to export on request
```

In other files which wish to use `YourModule`:

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

Take a look at *Good Practices* for some variants you will like to use in modern Perl code.

## DESCRIPTION

The Exporter module implements an `import` method which allows a module to export functions and variables to its users' namespaces. Many modules use Exporter rather than implementing their own `import` method because Exporter provides a highly flexible interface, with an implementation optimised for the common case.

Perl automatically calls the `import` method when processing a `use` statement for a module. Modules and `use` are documented in *perlfunc* and *perlmod*. Understanding the concept of modules and how the `use` statement operates is important to understanding the Exporter.

### How to Export

The arrays `@EXPORT` and `@EXPORT_OK` in a module hold lists of symbols that are going to be exported into the users name space by default, or which they can request to be exported, respectively. The symbols can represent functions, scalars, arrays, hashes, or typeglobs. The symbols must be given by full name with the exception that the ampersand in front of a function is optional, e.g.

```
@EXPORT    = qw(afunc $scalar @array);   # afunc is a function
@EXPORT_OK = qw(&bfunc %hash *typeglob); # explicit prefix on &bfunc
```

If you are only exporting function names it is recommended to omit the ampersand, as the implementation is faster this way.

### Selecting What To Export

Do **not** export method names!

Do **not** export anything else by default without a good reason!

Exports pollute the namespace of the module user. If you must export try to use `@EXPORT_OK` in preference to `@EXPORT` and avoid short or common symbol names to reduce the risk of name clashes.

Generally anything not exported is still accessible from outside the module using the

`YourModule::item_name` (or `$blessed_ref->method`) syntax. By convention you can use a leading underscore on names to informally indicate that they are 'internal' and not for public use.

(It is actually possible to get private functions by saying:

```
my $subref = sub { ... };
$subref->(@args);            # Call it as a function
$obj->$subref(@args);        # Use it as a method
```

However if you use them for methods it is up to you to figure out how to make inheritance work.)

As a general rule, if the module is trying to be object oriented then export nothing. If it's just a collection of functions then `@EXPORT_OK` anything but use `@EXPORT` with caution. For function and method names use barewords in preference to names prefixed with ampersands for the export lists.

Other module design guidelines can be found in *perlmod*.

### How to Import

In other files which wish to use your module there are three basic ways for them to load your module and import its symbols:

`use YourModule;`

This imports all the symbols from YourModule's `@EXPORT` into the namespace of the `use` statement.

`use YourModule ();`

This causes perl to load your module but does not import any symbols.

`use YourModule qw(...);`

This imports only the symbols listed by the caller into their namespace. All listed symbols must be in your `@EXPORT` or `@EXPORT_OK`, else an error occurs. The advanced export features of Exporter are accessed like this, but with list entries that are syntactically distinct from symbol names.

Unless you want to use its advanced features, this is probably all you need to know to use Exporter.

## Advanced features

### Specialised Import Lists

If any of the entries in an import list begins with !, : or / then the list is treated as a series of specifications which either add to or delete from the list of names to import. They are processed left to right. Specifications are in the form:

```
[!]name         This name only
[!]:DEFAULT     All names in @EXPORT
[!]:tag         All names in $EXPORT_TAGS{tag} anonymous list
[!]/pattern/    All names in @EXPORT and @EXPORT_OK which match
```

A leading ! indicates that matching names should be deleted from the list of names to import. If the first specification is a deletion it is treated as though preceded by :DEFAULT. If you just want to import extra names in addition to the default set you will still need to include :DEFAULT explicitly.

e.g., *Module.pm* defines:

```
@EXPORT      = qw(A1 A2 A3 A4 A5);
@EXPORT_OK   = qw(B1 B2 B3 B4 B5);
%EXPORT_TAGS = (T1 => [qw(A1 A2 B1 B2)], T2 => [qw(A1 A2 B3 B4)]);
```

```
    Note that you cannot use tags in @EXPORT or @EXPORT_OK.
    Names in EXPORT_TAGS must also appear in @EXPORT or @EXPORT_OK.
```

An application using Module can say something like:

```
    use Module qw(:DEFAULT :T2 !B3 A3);
```

Other examples include:

```
    use Socket qw(!/^[AP]F_/ !SOMAXCONN !SOL_SOCKET);
    use POSIX  qw(:errno_h :termios_h !TCSADRAIN !/^EXIT/);
```

Remember that most patterns (using //) will need to be anchored with a leading ^, e.g., `/^EXIT/` rather than `/EXIT/`.

You can say `BEGIN { $Exporter::Verbose=1 }` to see how the specifications are being processed and what is actually being imported into modules.

## Exporting without using Exporter's import method

Exporter has a special method, 'export_to_level' which is used in situations where you can't directly call Exporter's import method. The export_to_level method looks like:

```
    MyPackage->export_to_level($where_to_export, $package,
@what_to_export);
```

where `$where_to_export` is an integer telling how far up the calling stack to export your symbols, and `@what_to_export` is an array telling what symbols *to* export (usually this is `@_`). The `$package` argument is currently unused.

For example, suppose that you have a module, A, which already has an import function:

```
    package A;

    @ISA = qw(Exporter);
    @EXPORT_OK = qw ($b);

    sub import
    {
 $A::b = 1;     # not a very useful import method
    }
```

and you want to Export symbol `$A::b` back to the module that called package A. Since Exporter relies on the import method to work, via inheritance, as it stands Exporter::import() will never get called. Instead, say the following:

```
    package A;
    @ISA = qw(Exporter);
    @EXPORT_OK = qw ($b);

    sub import
    {
 $A::b = 1;
 A->export_to_level(1, @_);
    }
```

This will export the symbols one level 'above' the current package - ie: to the program or module that used package A.

Note: Be careful not to modify `@_` at all before you call export_to_level - or people using your package will get very unexplained results!

### Exporting without inheriting from Exporter

By including Exporter in your `@ISA` you inherit an Exporter's import() method but you also inherit several other helper methods which you probably don't want. To avoid this you can do

```
package YourModule;
use Exporter qw( import );
```

which will export Exporter's own import() method into YourModule. Everything will work as before but you won't need to include Exporter in `@YourModule::ISA`.

Note: This feature was introduced in version 5.57 of Exporter, released with perl 5.8.3.

### Module Version Checking

The Exporter module will convert an attempt to import a number from a module into a call to `$module_name->require_version($value)`. This can be used to validate that the version of the module being used is greater than or equal to the required version.

The Exporter module supplies a default `require_version` method which checks the value of `$VERSION` in the exporting module.

Since the default `require_version` method treats the `$VERSION` number as a simple numeric value it will regard version 1.10 as lower than 1.9. For this reason it is strongly recommended that you use numbers with at least two decimal places, e.g., 1.09.

### Managing Unknown Symbols

In some situations you may want to prevent certain symbols from being exported. Typically this applies to extensions which have functions or constants that may not exist on some systems.

The names of any symbols that cannot be exported should be listed in the `@EXPORT_FAIL` array.

If a module attempts to import any of these symbols the Exporter will give the module an opportunity to handle the situation before generating an error. The Exporter will call an export_fail method with a list of the failed symbols:

```
@failed_symbols = $module_name->export_fail(@failed_symbols);
```

If the `export_fail` method returns an empty list then no error is recorded and all the requested symbols are exported. If the returned list is not empty then an error is generated for each symbol and the export fails. The Exporter provides a default `export_fail` method which simply returns the list unchanged.

Uses for the `export_fail` method include giving better error messages for some symbols and performing lazy architectural checks (put more symbols into `@EXPORT_FAIL` by default and then take them out if someone actually tries to use them and an expensive check shows that they are usable on that platform).

### Tag Handling Utility Functions

Since the symbols listed within `%EXPORT_TAGS` must also appear in either `@EXPORT` or `@EXPORT_OK`, two utility functions are provided which allow you to easily add tagged sets of symbols to `@EXPORT` or `@EXPORT_OK`:

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);
```

```
Exporter::export_tags('foo');     # add aa, bb and cc to @EXPORT
Exporter::export_ok_tags('bar');  # add aa, cc and dd to @EXPORT_OK
```

Any names which are not tags are added to @EXPORT or @EXPORT_OK unchanged but will trigger a warning (with -w) to avoid misspelt tags names being silently added to @EXPORT or @EXPORT_OK. Future versions may make this a fatal error.

## Generating combined tags

If several symbol categories exist in %EXPORT_TAGS, it's usually useful to create the utility ":all" to simplify "use" statements.

The simplest way to do this is:

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

# add all the other ":class" tags to the ":all" class,
# deleting duplicates
{
  my %seen;

  push @{$EXPORT_TAGS{all}},
    grep {!$seen{$_}++} @{$EXPORT_TAGS{$_}} foreach keys %EXPORT_TAGS;
}
```

*CGI.pm* creates an ":all" tag which contains some (but not really all) of its categories. That could be done with one small change:

```
# add some of the other ":class" tags to the ":all" class,
# deleting duplicates
{
  my %seen;

  push @{$EXPORT_TAGS{all}},
    grep {!$seen{$_}++} @{$EXPORT_TAGS{$_}}
      foreach qw/html2 html3 netscape form cgi internal/;
}
```

Note that the tag names in %EXPORT_TAGS don't have the leading ':'.

## AUTOLOADed Constants

Many modules make use of AUTOLOADing for constant subroutines to avoid having to compile and waste memory on rarely used values (see *perlsub* for details on constant subroutines). Calls to such constant subroutines are not optimized away at compile time because they can't be checked at compile time for constancy.

Even if a prototype is available at compile time, the body of the subroutine is not (it hasn't been AUTOLOADed yet). perl needs to examine both the ( ) prototype and the body of a subroutine at compile time to detect that it can safely replace calls to that subroutine with the constant value.

A workaround for this is to call the constants once in a BEGIN block:

```
package My ;

use Socket ;
```

```
    foo( SO_LINGER );     ## SO_LINGER NOT optimized away; called at runtime
    BEGIN { SO_LINGER }
    foo( SO_LINGER );     ## SO_LINGER optimized away at compile time.
```

This forces the AUTOLOAD for SO_LINGER to take place before SO_LINGER is encountered later in My package.

If you are writing a package that AUTOLOADs, consider forcing an AUTOLOAD for any constants explicitly imported by other packages or which are usually used when your package is used.

## Good Practices

### Declaring @EXPORT_OK and Friends

When using Exporter with the standard strict and warnings pragmas, the our keyword is needed to declare the package variables @EXPORT_OK, @EXPORT, @ISA, etc.

```
  our @ISA = qw(Exporter);
  our @EXPORT_OK = qw(munge frobnicate);
```

If backward compatibility for Perls under 5.6 is important, one must write instead a use vars statement.

```
  use vars qw(@ISA @EXPORT_OK);
  @ISA = qw(Exporter);
  @EXPORT_OK = qw(munge frobnicate);
```

### Playing Safe

There are some caveats with the use of runtime statements like require Exporter and the assignment to package variables, which can very subtle for the unaware programmer. This may happen for instance with mutually recursive modules, which are affected by the time the relevant constructions are executed.

The ideal (but a bit ugly) way to never have to think about that is to use BEGIN blocks. So the first part of the *SYNOPSIS* code could be rewritten as:

```
  package YourModule;

  use strict;
  use warnings;

  our (@ISA, @EXPORT_OK);
  BEGIN {
     require Exporter;
     @ISA = qw(Exporter);
     @EXPORT_OK = qw(munge frobnicate);  # symbols to export on request
  }
```

The BEGIN will assure that the loading of *Exporter.pm* and the assignments to @ISA and @EXPORT_OK happen immediately, leaving no room for something to get awry or just plain wrong.

With respect to loading Exporter and inheriting, there are alternatives with the use of modules like base and parent.

```
  use base qw( Exporter );
  # or
  use parent qw( Exporter );
```

Any of these statements are nice replacements for `BEGIN { require Exporter; @ISA = qw(Exporter); }` with the same compile-time effect. The basic difference is that `base` code interacts with declared `fields` while `parent` is a streamlined version of the older `base` code to just establish the IS-A relationship.

For more details, see the documentation and code of *base* and *parent*.

Another thorough remedy to that runtime vs. compile-time trap is to use *Exporter::Easy*, which is a wrapper of Exporter that allows all boilerplate code at a single gulp in the use statement.

```
use Exporter::Easy (
    OK => [ qw(munge frobnicate) ],
);
# @ISA setup is automatic
# all assignments happen at compile time
```

### What not to Export

You have been warned already in *Selecting What To Export* to not export:

- method names (because you don't need to and that's likely to not do what you want),
- anything by default (because you don't want to surprise your users... badly)
- anything you don't need to (because less is more)

There's one more item to add to this list. Do **not** export variable names. Just because `Exporter` lets you do that, it does not mean you should.

```
@EXPORT_OK = qw( $svar @avar %hvar ); # DON'T!
```

Exporting variables is not a good idea. They can change under the hood, provoking horrible effects at-a-distance, that are too hard to track and to fix. Trust me: they are not worth it.

To provide the capability to set/get class-wide settings, it is best instead to provide accessors as subroutines or class methods instead.

## SEE ALSO

`Exporter` is definitely not the only module with symbol exporter capabilities. At CPAN, you may find a bunch of them. Some are lighter. Some provide improved APIs and features. Peek the one that fits your needs. The following is a sample list of such modules.

```
Exporter::Easy
Exporter::Lite
Exporter::Renaming
Exporter::Tidy
Sub::Exporter / Sub::Installer
Perl6::Export / Perl6::Export::Attrs
```

## LICENSE