```
require SomeModule;
require "SomeModule.pm";
```

differ from each other in two ways. In the first case, any double colons in the module name, such as `Some::Module`, are translated into your system's directory separator, usually "/". The second case does not, and would have to be specified literally. The other difference is that seeing the first `require` clues in the compiler that uses of indirect object notation involving "SomeModule", as in `$ob = purge SomeModule`, are method calls, not function calls. (Yes, this really can make a difference.)

Because the `use` statement implies a `BEGIN` block, the importing of semantics happens as soon as the `use` statement is compiled, before the rest of the file is compiled. This is how it is able to function as a pragma mechanism, and also how modules are able to declare subroutines that are then visible as list or unary operators for the rest of the current file. This will not work if you use `require` instead of `use`. With `require` you can get into this problem:

```
require Cwd;                # make Cwd:: accessible
$here = Cwd::getcwd();

use Cwd;                    # import names from Cwd::
$here = getcwd();

require Cwd;                # make Cwd:: accessible
$here = getcwd();           # oops! no main::getcwd()
```

In general, `use Module ()` is recommended over `require Module`, because it determines module availability at compile time, not in the middle of your program's execution. An exception would be if two modules each tried to `use` each other, and each also called a function from that other module. In that case, it's easy to use `require` instead.

Perl packages may be nested inside other package names, so we can have package names containing `::`. But if we used that package name directly as a filename it would make for unwieldy or impossible filenames on some systems. Therefore, if a module's name is, say, `Text::Soundex`, then its definition is actually found in the library file *Text/Soundex.pm*.

Perl modules always have a *.pm* file, but there may also be dynamically linked executables (often ending in *.so*) or autoloaded subroutine definitions (often ending in *.al*) associated with the module. If so, these will be entirely transparent to the user of the module. It is the responsibility of the *.pm* file to load (or arrange to autoload) any additional functionality. For example, although the POSIX module happens to do both dynamic loading and autoloading, the user can say just `use POSIX` to get it all.

57.1.6 Making your module threadsafe

Since 5.6.0, Perl has had support for a new type of threads called interpreter threads (ithreads). These threads can be used explicitly and implicitly.

Ithreads work by cloning the data tree so that no data is shared between different threads. These threads can be used by using the `threads` module or by doing fork() on win32 (fake fork() support). When a thread is cloned all Perl data is cloned, however non-Perl data cannot be cloned automatically. Perl after 5.7.2 has support for the `CLONE` special subroutine. In `CLONE` you can do whatever you need to do, like for example handle the cloning of non-Perl data, if necessary. `CLONE` will be called once as a class method for every package that has it defined (or inherits it). It will be called in the context of the new thread, so all modifications are made in the new area. Currently CLONE is called with no parameters other than the invocant package name, but code should not assume that this will remain unchanged, as it is likely that in future extra parameters will be passed in to give more information about the state of cloning.

If you want to CLONE all objects you will need to keep track of them per package. This is simply done using a hash and Scalar::Util::weaken().

57.2 SEE ALSO

See *perlmodlib* for general style issues related to building Perl modules and classes, as well as descriptions of the standard library and CPAN, *Exporter* for how Perl's standard import/export mechanism works, *perltoot* and *perltooc* for an in-depth tutorial on creating classes, *perlobj* for a hard-core reference document on objects, *perlsub* for an explanation of functions and scoping, and *perlxstut* and *perlguts* for more information on writing extension modules.