

## Re: Opinions on "new SomeObject" vs. "SomeObject->new()"

*Source:* <http://coding.derkeiler.com/Archive/Perl/comp.lang.perl.misc/2003-10/0900.html>

---

*From:* Abigail ([abigail\\_at\\_abigail.nl](mailto:abigail_at_abigail.nl))

*Date:* 10/09/03

Date: 08 Oct 2003 23:15:37 GMT

Matija Papec ([perl@my-header.org](mailto:perl@my-header.org)) wrote on MMMDCLXXXIX September MCMXCIII in <[URL:news:gj06ov8h7gkrv458kt9hda03n53senaj2@4ax.com](mailto:URL:news:gj06ov8h7gkrv458kt9hda03n53senaj2@4ax.com)>:

<>

- <> As for cargo cult programming, how about some general programming guidelines
- <> which should have community consensus? Like `perlstyle.pod` which could make
- <> some guidelines more official?

Please no. *\*Personal\** programming guidelines are ok. But let's not wade into the swamps of trying to derive guidelines with community consensus. Remember that are hundreds of thousands of Perl programmers, but only a few hundred of them are active on Perl forums – but there are dozens of those forums.

At best you reach consensus between an insignificant fraction of the "community", but you'll falsely label it as a "generally accepted style guide". If you want a language that comes with a notion on what is good style, and what should be avoided, you find it thataway ----> [www.python.org](http://www.python.org).

Anyway, I do have some personal guidelines. People who have heard my talk at this years YAPC::NA already know it, but I'd include it here. Note that these are *\*my\** guidelines, and I'm not trying to enforce them on anyone else. Nor do I want to claim they have "community consensus".

=over 4

=item Warnings SHOULD be turned on.

Turning on warnings helps you finding problems in your code. But it's only useful if you understand the messages generated. You should also know when to disable warnings – they are warnings after all, pointing out potential problems, but not always bugs.

=item Larger programs SHOULD use strictness.

The three forms of strictness can help you to prevent making certain mistakes by restricting what you can do. But you should know when it is appropriate to turn off a particular strictness, and regain your freedom.

=item The return values of system calls SHOULD be checked.

NFS servers will be down, permissions will change, file will disappear, disk will fill up, resources will be used up. System calls can fail for a number of reasons, and failure is not uncommon. Programs should never assume a system call will succeed – they should check for success and deal with failures. The rare case where you don't care whether the call succeeded should have a comment saying so.

All system calls should be checked, including, but not limited to, C<close>, C<seek>, C<flock>, C<fork> and C<exec>.

=item Programs running on behalf of someone else MUST use tainting; Untainting SHOULD be done by checking for allowed formats.

Daemons listening to sockets (including, but not limited to CGI programs) and suid and sgid programs are potential security holes. Tainting can help securing your programs by tainting data coming from untrusted sources. But it's only useful if you untaint carefully: check for accepted formats.

=item Programs MUST deal with signals appropriately.

Signals can be sent to the program. There are default actions – but they are not always appropriate. If not, signal handlers need to be installed. Care should be taken since not everything is reentrant. Both pre-5.8.0 and post-5.8.0 have their own issues.

=item Programs MUST deal with early termination appropriately.

C<END> blocks and C<\_\_DIE\_\_> handlers should be used if the program needs to clean up after itself, even if the program terminates unexpectedly – for instance due to a signal, an explicit C<die> or a fatal error.

=item Programs MUST have an exit value of 0 when running successfully, and a non-0 exit value when there's a failure.

Why break a good UNIX tradition? Different failures should have different exit values.

=item Daemons SHOULD never write to STDOUT or STDERR but SHOULD use the syslog service to log messages. They should use an appropriate facility and appropriate priorities when logging

messages.

Daemons run with no controlling terminal, and usually its standard output and standard error disappear. The syslog service is a standard UNIX utility especially geared towards daemons with a logging need. It allows the system administration to determine what is logged, and where, without the need to modify the (running) program.

=item Programs SHOULD use Getopt::Long to parse options. Programs MUST follow the POSIX standard for option parsing.

Getopt::Long supports historical style arguments (single dash, single letter, with bundling), POSIX style, and GNU extensions. Programs should accept reasonable synonyms for option names.

=item Interactive programs MUST print a usage message when called with wrong, incorrect or incomplete options or arguments.

Users should know how to call the program.

=item Programs SHOULD support the C<--help> and C<--version> options.

C<--help> should print a usage message and exit, while C<--version> should print the version number of the program.

=item Code SHOULD have an exhaustive regression test suite.

Regression tests help catch breakage of code. The regression tests should 'touch' all the code – that is, every piece of code should be executed when running the regression suite. All border should be checked. More tests is usually better than less test. Behaviour on invalid inputs needs to be tested as well.

=item Code SHOULD be in source control.

And a code source control tool will take care of keeping track of a history or changes log, version numbers and who made the most recent change(s).

=item All database modifying statements MUST be wrapped inside a transaction.

Your data is likely to be more important than the runtime or codesize of your program. Data integrity should be retained at all costs.

=item Subroutines in standalone modules SHOULD perform argument checking and MUST NOT assume valid arguments are passed.

Perl doesn't compile check the types of or even the number of arguments. You will have to do that yourself.

=item Objects SHOULD NOT use data inheritance unless it is appropriate.

This means that "normal" objects, where the attributes are stored inside anonymous hashes or arrays should not be used. Non-OO programs benefit from namespaces and strictness, why shouldn't objects? Use objects based on keying scalars, like fly-weight objects, or inside-out objects. You wouldn't use public attributes in Java all over the place either, would you?

=item Comment SHOULD be brief and to the point.

If you need lots of comments to explain your code, you may consider rewriting it. Subroutines that have a whole blob of comments describing arguments and return values are suspect. But do document invariants, pre- and postconditions, (mathematical) relationships, theorems, observations and other relevant things the code assumes. Variables with a broad scope might warrant comments too.

=item POD SHOULD not be interleaved with the code, and is not an alternative for comments.

Comments and POD have two different purposes. Comments are there for the programmer. The person who has to maintain the code. POD is there to create user documentation from. For the person using the code. POD should not be interleaved with the code because this makes it harder to find the code.

=item Comments, POD and variable names MUST use English.

English is the current Lingua Franca.

=item Variables SHOULD have an as limited scope as is appropriate.

"No global variables", but better. Just disallowing global variables means you can still have a loop variant with a file-wide scope. Limiting the scope of variables means that loop variants are only known in the body of the loop, temporary variables only in the current block, etc. But sometimes it's useful for a variable to be global, or have a file-wide scope.

=item Variables with a small scope SHOULD have short names, variables with a broad scope SHOULD have descriptive names.

`C<$array_index_counter>` is silly;  
`C<for (my $i = 0; $i < @array; $i++) { .. }>` is perfect.  
But a variable that's used all over the place needs a descriptive name.

=item Constants (or variables intended to be constant) SHOULD have names in all capitals, (with underscores separating words), so SHOULD IO handles. Package and class names SHOULD use title case, while other

variables (including subroutines) SHOULD use lower case, words separated by underscores.

This seems to be quite common in the Perl world.

=item Custom delimiters SHOULD be tall and skinny.

C</>, C<!>, C<|> and the four sets of braces are acceptable, C<#>, C<@> and C<\*> are not. Thick delimiters take too much attention. An exception is made for: C<q \$Revision: 1.1.1.1\$>, because RCS and CVS scan for the dollars.

=item Operators SHOULD be separated from their operands by whitespace, with a few exceptions.

Whitespace increases readability. The exceptions are:

=over 4

=item Unary C<+>, C<->, C<|>, C<~> and C<!>.

=item No whitespace between a comma and its left operand.

=back

Note that there is whitespace between C<+> and C<-> and their operands, and between C<-> and its operands.

=item There SHOULD be whitespace between an identifier and its indices. There SHOULD be whitespace between successive indices.

Taking an index is an operation as well, so there should be whitespace. Obviously, we cannot apply this rule in interpolative contexts.

=item There SHOULD be whitespace between a subroutine name and its parameters, even if the parameters are surrounded by parens.

Again, readability.

=item There SHOULD NOT be whitespace after an opening parenthesis, or before a closing parenthesis. There SHOULD NOT be whitespace after an opening indexing bracket or brace, or before a closing indexing bracket or brace.

That is: C<\$array [\$key]>, C<\$hash {\$key}> and C<sub (\$arg)>.

=item The opening brace of a block SHOULD be on the same line as the keyword and the closing brace SHOULD align with the keyword, but short blocks are allowed to be on one line.

comp.lang.perl.misc: Re: Opinions on "new SomeObject" vs. "SomeObject->new()"

This is K&R style bracing, except that we require it for subroutines as well.  
We do allow `C<map {$_ * $_} @args>` to be on one line though.

=item No cuddled elses or elsifs. But the `C<while>` of a  
`C<do { } while>` construct should be on the same line  
as the closing brace.

It just looks better that way! `C<;->`

=item Indents SHOULD be 4 spaces wide. Indents MUST NOT contain tabs.

4 spaces seems to be an often used compromise between the need to make  
indents stand out, and not getting cornered. Tabs are evil.

=item Lines MUST NOT exceed 80 characters.

There is just no excuse for that. More than 80 characters means it  
will wrap in too many situations, leading to hard to read code.

=item Align code vertically.

This makes code look more pleasing, and it brings attention to the  
fact similar things are happening on close by lines. Example:

```
my $var = 18;
my $long_var = "Some text";
```

=back

Abigail

```
--
perl -we '$@="\145\143\150\157\040\042\112\165\163\164\040\141\156\157\164".
"\150\145\162\040\120\145\162\154\040\110\141\143\153\145\162".
"\042\040\076\040\057\144\145\166\057\164\164\171";`$@`'
```