Perl Beyond Syntax

Perl 5 is a large language, like any language intended to solve problems in the real world. Effective Perl programs require more than mere understanding of syntax; you must also begin to understand how Perl's features interact and common ways of solving well-understood problems in Perl.

Prepare for the second learning curve of Perl: thinking Perlishly through the effective use of common patterns of behavior and builtin shortcuts which, when used well, allow you to write concise, powerful code.

Idioms

Any language--programming or natural--develops *idioms*, or common patterns of expression. The earth revolves, but we speak of the sun rising or setting. We talk of clever hacks and nasty hacks and slinging code.

As you learn Perl 5 more clearly, you will begin to see and understand common idioms. They're not quite language features--you don't *have* to use them--and they're not quite large enough that you can encapsulate them away behind functions and methods. Instead, they're mannerisms. They're ways of writing Perl with a Perlish accent.

The Object as $self

Perl 5's object system (*moose*) treats the invocant of a method as a mundane parameter. The invocant of a class method--a string containing the name of the class--is that method's first parameter. The invocant of an object or instance method--the object itself--is that method's first parameter. You are free to use or ignore it as you see fit.

Idiomatic Perl 5 uses `$class` as the name of the class method and `$self` for the name of the object invocant. This is a convention not enforced by the language itself, but it is a convention strong enough that useful extensions such as `MooseX::Method::Signatures` assume you will use `$self` as the name of the invocant by default.

Named Parameters

Without a module such as `signatures` or `MooseX::Multimethods`, Perl 5's argument passing mechanism is simple: all arguments flatten into a single list accessible through `@_` (*function_parameters*). While this simplicity is occasionally too simple--named parameters can be very useful at times--it does not preclude the use of idioms to provide named parameters.

The list context evaluation and assignment of `@_` allows you to unpack named parameters as pairs in a natural and Perlish fashion. Even though this function call is equivalent to passing a comma-separated or `qw//`-created list, arranging the arguments as if they were true pairs of keys and values makes the caller-side of the function appear to support named parameters:

```
make_ice_cream_sundae(
    whipped_cream => 1,
    sprinkles     => 1,
    banana        => 0,
    ice_cream     => 'mint chocolate chip',
);
```

The callee side can unpack these parameters into a hash and treat the hash as if it were the single argument:

```
sub make_ice_cream_sundae
{
    B<my %args = @_;>
```

```
    my $ice_cream = get_ice_cream( $args{ice_cream}) );
    ...
}
```

Perl Best Practices suggests passing a hash reference instead. This allows Perl to check that you've constructed a valid hash on the caller side. It also uses slightly less memory than the other approach.

This technique works well with `import()` (*importing*); you can process as many parameters as you like before slurping the remainder into a hash:

```
sub import
{
    B<my ($class, %args)  = @_;>
    my $calling_package = caller();


    ...
}
```

The Schwartzian Transform

People new to Perl sometimes overlook the importance of lists and list processing as a fundamental component of expression evaluation. Put more simply, the ability for Perl programmers to chain expressions which evaluate to variable-length lists provides countless opportunities to manipulate data effectively.

The *Schwartzian transform* is an elegant demonstration of that principle as an idiom handily borrowed from the Lisp family of languages.

Suppose you have a Perl hash which associates the names of your co-workers with their phone extensions:

```
my %extensions =
(
     4 => 'Jerryd',
     5 => 'Rudy',
     6 => 'Juwan',
     7 => 'Brandon',
    10 => 'Joel',
    21 => 'Marcus',
    24 => 'Andre',
    23 => 'Martell',
    52 => 'Greg',
    88 => 'Nic',
);
```

Suppose you want to print a list of extensions and co-workers sorted by their names, not their extensions. In other words, you need to sort this hash by its values. Sorting the values of the hash in string order is easy:

```
my @sorted_names = sort values %extensions;
```

... but that loses the association of names with extensions. Instead, use the Schwartzian transform to transform the data before and after sorting it to preserve the necessary information. First, convert the hash into a list of data structures which contain the vital information in sortable fashion. In this case, convert the hash pairs into two-element anonymous arrays:

```
my @pairs = map { [ $_, $extensions{$_} ] } keys %extensions;
```

Reversing the hash *in place* would work if no one had the same name. This particular data set presents no such problem, but code defensively.

`sort` takes the list of anonymous arrays and compares their second elements (the names) with a stringwise comparison:

```
my @sorted_pairs = sort { $a->[1] cmp $b->[1] } @pairs;
```

Given `@sorted_pairs`, a second `map` operation converts the data structure to a more usable form:

```
my @formatted_exts = map { "$_->[1], ext. $_->[0]" } @sorted_pairs;
```

... and now you can print the whole thing:

```
say for @formatted_exts;
```

Of course, this uses several temporary variables (with admittedly bad names). It's a worthwhile technique and good to understand, but the real magic is in the combination:

```
say for
    map  { " $_->[1], ext. $_->[0]"          }
    sort {   $a->[1] cmp   $b->[1]           }
    map  { [ $_      =>    $extensions{$_} ] }
        keys %extensions;
```

Read the expression from right to left, in the order of evaluation. For each key in the extensions hash, make a two-item anonymous array containing the key and the value from the hash. Sort that list of anonymous arrays by their second elements, the values from the hash. Format a string of output from those sorted arrays.

The Schwartzian transform is this pipeline of `map`-`sort`-`map` where you transform a data structure into another form easier for sorting and then transform it back into your preferred form for modification.

This transformation is simple. Consider the case where calculating the right value to sort is expensive in time or memory, such as calculating a cryptographic hash for a large file. In that case, the Schwartzian transform is also useful because you can perform those expensive operations once (in the rightmost `map`), compare them repeatedly from a de facto cache in the `sort`, and then remove them in the leftmost `map`.

Easy File Slurping

Perl 5's magic global variables are truly global in many cases. It's all too easy to clobber their values elsewhere, unless you use `local` everywhere. Yet this requirement has allowed the creation of several interesting idioms. For example, you can slurp files into a scalar in a single expression:

```
my $file = do { local $/ = <$fh> };

# or

my $file = do { local $/; <$fh> };
```

`$/` is the input record separator. `local`izing it sets its value to `undef`, pending assignment. That `local`ization takes place *before* the assignment. As the value of the separator is undefined, Perl happily reads the entire contents of the filehandle in one swoop and assigns that value to `$/`.

Because a `do` block evaluates to the value of the last expression evaluated within the block, this evaluates to the value of the assignment, or the contents of the file. Even though `$/` immediately reverts to its previous state at the end of the block, `$file` now contains the contents of the file.

The second example is similar, except that it performs no assignment and merely returns the single line read from the filehandle. You may see either example; they both work the same way in this case.

This can be useful (and, admittedly, maddening for people unfamiliar with this particular combination of Perl 5 features) if you don't have `File::Slurp` installed from the CPAN.

Controlled Execution

The effective difference between a program and a module is in its intended use. Users invoke programs directly, while programs load modules after execution has already begun. The technical difference between a program and a module is whether it's meaningful to invoke the entity directly.

You may encounter this when you wish to use Perl's testing tools (*testing*) to test functions in a standalone program or when you wish to make a module users can run directly. All you need to do is to discover *how* Perl began to execute a piece of code. For this, use `caller`.

`caller`'s single optional argument is the number of call frames which to report. (A *call frame* is the bookkeeping information which represents a function call.) You can get information about the current call frame with `caller(0)`. To allow a module to run correctly as a program *or* a module, write an appropriate `main()` function and add a single line to the start of the module:

```
main() unless caller(0);
```

If there's *no* caller for the module, someone invoked it directly as a program (with `perl path/to/Module.pm` instead of `use Module;`).

Checking the eighth element of the list returned from `caller` in list context may be more accurate in most cases, but it's rare. This value is true if the call frame represents `use` or `require` and undef otherwise.

Handling Main

Perl requires no special syntax for creating closures (*closures*); you can close over a lexical variable inadvertently. This is *rarely* a problem in practice, apart from specific concerns in mod_perl situations... and `main()` functions.

Many programs commonly set up several file-scoped lexical variables before handing off processing to other functions. It's tempting to use these variables directly, rather than passing values to and returning values from functions, especially as programs grow to provide more features. Worse yet, these programs may come to rely on subtleties of what happens when during Perl 5's compilation process; a variable you *thought* would be initialized to a specific value may not get initialized until much later.

There is a simple solution. Wrap the main code of your program in a simple function, `main()`. Encapsulate all of the variables you don't need as true globals. Then add a single line to the beginning of your program, after you've used all of the modules and pragmas you need:

```
#!/usr/bin/perl

use Modern::Perl;
use autodie;

...

B<main( @ARGS );>
```

Calling `main()` *before* performing anything else in the program forces you to be explicit about initialization and order of compilation. It also helps to remind you to encapsulate the behavior of your program into functions and modules. (It works nicely with files which can be programs and libraries--*controlled_execution*.)

Postfix Parameter Validation

Even if you don't use a CPAN module such as `Params::Validate` or `MooseX::Params::Validate` to verify that the parameters your functions receive are correct, you can still benefit from occasional checks for correctness. The `unless` control flow modifier is an easy and readable way to assert your expectations at the beginning of a function.

Suppose your function takes two arguments, no more and no less. You *could* write:

```
use Carp;

sub groom_monkeys
{
    if (@_ != 2)
    {
        croak 'Monkey grooming requires two monkeys!';
    }
}
```

... but from a linguistic perspective, the consequences are more important than the check and deserve to be at the *start* of the expression:

```
croak 'Monkey grooming requires two monkeys!' if @_ != 2;
```

... which, depending on your preference for reading postfix conditions, you can simplify to:

```
croak 'Monkey grooming requires two monkeys!' unless @_ == 2;
```

This is easier to read if you focus on the text of the message ("You need to pass two parameters!") and the test (@_ should contain two items). It's almost a single row in a truth table.

Regex En Passant

Many Perl 5 idioms rely on the language design where expressions evaluate to values, as in:

```
say my $ext_num = my $extension = 42;
```

It's bad form to write code like that, but it demonstrates the point: you can use the value of one expression in another expression. This isn't a new idea; you've likely used the return value of a function in a list or as an argument to another function before. You may not have realized its implications.

Suppose you have a whole name and you want to extract the first name. This is easy to do with a regular expression:

```
my ($first_name) = $name =~ /($first_name_rx)/;
```

... where `$first_name_rx` is a precompiled regular expression. In list context, a successful regex match returns a list of all captures, and Perl assigns the first one to `$first_name`.

Now imagine if you want to modify the name, perhaps removing all non-word characters to create a useful username for a system account. You can write:

```
(my $normalized_name = $name) =~ tr/A-Za-z//dc;
```

Unlike the previous example, this one reads right to left. First, assign the value of `$name` to `$normalized_name`. Then, perform the transliteration on `$normalized_name`The parentheses here affect the precedence so that the assignment happens first.. The assignment expression evaluates to the *variable* `$normalized_name`. This technique works on all sorts of in-place modification operators:

```
my $age = 14;
(my $next_age = $age)++;

say "Next year I will be $next_age";
```

Unary Coercions

Perl 5's type system often does the right thing, at least if you choose the correct operators. To concatenate strings, use the string concatenation operator, and Perl will treat both scalars as strings. To add two numbers, use the addition operator and Perl will treat both scalars as numeric.

Sometimes you have to give Perl a hint about what you mean. Several *unary coercions* exist, by which you can use Perl 5 operators to force the evaluation of a value a specific way.

To ensure that Perl treats a value as numeric, add zero:

```
my $numeric_value = 0 + $value;
```

To ensure that Perl treats a value as boolean, double negate it:

```
my $boolean_value = !! $value;
```

To ensure that Perl treats a value as a string, concatenate it with the empty string:

```
my $string_value = '' . $value;
```

Though the need for these coercions is vanishingly rare, you should understand these idioms if you encounter them.

Global Variables

Perl 5 provides several *super global variables* that are truly global, not restricted to any specific package. These super globals have two drawbacks. First, they're global; any direct or indirect modifications may have effects on other parts of the program. Second, they're terse. Experienced Perl 5 programmers have memorized some of them. Few people have memorized all of them. Only a handful are ever useful. `perldoc perlvar` contains the exhaustive list of such variables.

Managing Super Globals

The best approach to managing the global behavior of these super globals is to avoid using them. When you must use them, use `local` in the smallest possible scope to constrain any modifications. You are still susceptible to any changes code you *call* makes to those globals, but you reduce the likelihood of surprising code *outside* of your scope.

Workarounds exist for some of this global behavior, but many of these variables have existed since Perl 1 and will continue as part of Perl 5 throughout its lifetime. As the easy file slurping idiom (*easy_file_slurping*) demonstrates, this is often possible:

```
my $file = do { B<local $/> = <$fh> };
```

The effect of `local`izing `$/` lasts only through the end of the block. There is a low chance that any Perl code will run as a result of reading lines from the filehandleA tied filehandle is one of the few possibilities. and change the value of `$/` within the `do` block.

Not all cases of using super globals are this easy to guard, but this often works.

Other times you need to *read* the value of a super global and hope that no other code has modified it. Catching exceptions with an `eval` block can be susceptible to race conditionsUse `Try::Tiny` instead!, in that `DESTROY()` methods invoked on lexicals that have gone out of scope may reset `$@`:

```
local @;

eval { ... };

if (B<my $exception = $@>) { ... }
```

Copy `$@` *immediately* to preserve its contents.

English Names

The core `English` module provides verbose names for the punctuation-heavy super globals. Import them into a namespace with:

```
use English '-no_match_vars';
```

Subsequently you can use the verbose names documented in `perldoc perlvar` within the scope of this namespace.

Three regex-related super globals (`$&`, `$``, and `$'`) impose a global performance penalty for *all* regular expressions within a program. If you neglect to provide that import flag, your program will suffer the penalty even if you don't explicitly read from those variables. This is not the default behavior for backwards-compatibility concerns.

Modern Perl programs should use the `@-` variable as a replacement for the terrible three.

Useful Super Globals

Most modern Perl 5 programs can get by with using only a couple of the super globals. Several exist for special circumstances you're unlikely to encounter. While `perldoc perlvar` is the canonical documentation for most of these variables, some deserve special mention.

... or characters, in a Unicode mode?

Many places within Perl 5 itself make system calls without your knowledge. The value of this variable can change out from under you, so copy it *immediately* after making such a call yourself.

Alternatives to Super Globals

The worst culprits for action at a distance relate to IO and exceptional conditions. Using `Try::Tiny` (*exception_caveats*) will help insulate you from the tricky semantics of proper exception handling. `local`izing and copying the value of `$!` can help you avoid strange behaviors when Perl makes implicit system calls.

You can use methods on lexical filehandles (*lexical_filehandles*) rather than IO-related super globals by `use`ing `IO::Handle`. In place of `select`ing a filehandle, then manipulating `$|`, call the `autoflush()` method on the lexical filehandle directly. Use the `input_line_number()` method to get the equivalent of `$.` for that specific filehandle. See the `IO::Handle` documentation for other appropriate methods.