



Perl Training

a journey to the most exotic language on Earth

Ettore Di Giacinto - @mudler

SUSE

- Material available @
https://github.com/mudler/perl_training
- Modern Perl is an AWESOME reference - http://modernperlbooks.com/books/modern_perl_2016/index.html
- If you following the training with a laptop, install Devel::REPL (`cpanm -n Devel::REPL` or `zypper in perl-Devel-REPL`, start it with: `re.pl`)

Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

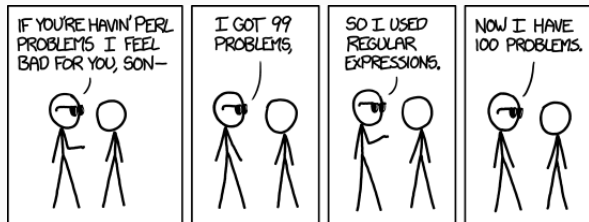
References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



Introduction - Ultra-fast Perl overview

Pros:

- Huge library archive - CPAN
- Extremely flexible language
- Performs quite well to be interpreted
- Lots of functionalities

but. . . cons:

- TIMTOWTDI - good/bad thing
- Lot of caveats
- Difficult to deal when we want optimizations
- Not all things from CPAN are good
- Lots of functionalities. . . which usage should be discouraged!

Introduction - The Life Cycle of a Perl Program

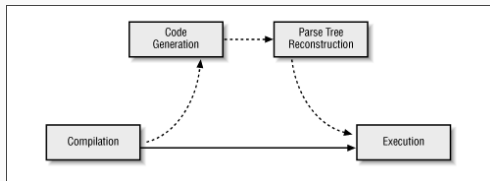


Figure 1: from https://docstore.mik.ua/orelly/perl3/prog/ch18_01.htm

- The Compilation - Tree building (**use** and **no** declarations, Lexical declaration with no assignment. BEGIN are executed in FIFO, later interpreter is called again to re-evaluate CHECK blocks in LIFO)
- The Code Generation Phase (optional) - if CHECK blocks were specified, Perl will generate intermediate C code (or Bytecode) compiling them so your machine can execute that image directly
- The Parse Tree Reconstruction Phase (optional) - if Bytecode, then Perl needs to reconstruct the Parse tree before being able to execute
- Execution - The interpreter takes the parse tree and executes it.

Introduction - Compiletime vs Runtime

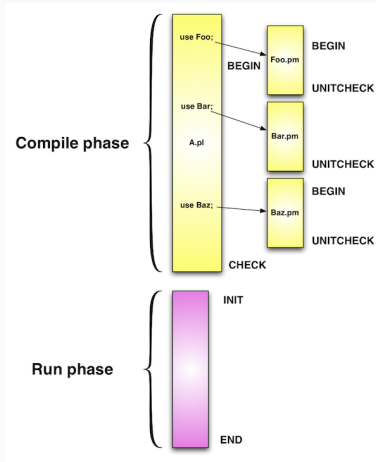


Figure 2: source: <https://www.effectiveperlprogramming.com/2011/03/know-the-phases-of-a-perl-programs-execution/>

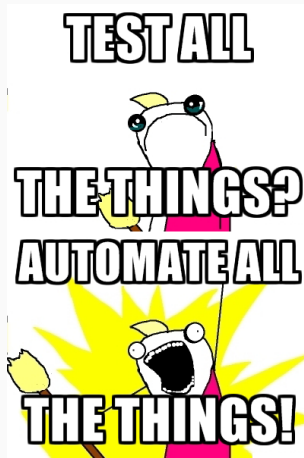


Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

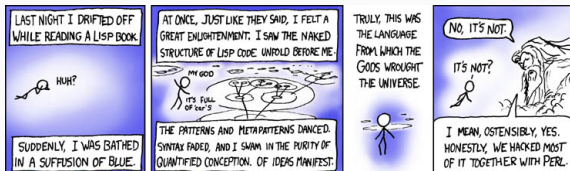
References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



Context is the only pivot of Perl. It was strongly inspired by human language: 'I want twenty bottle(s)' - 'I want one bottle', depending on the context you get what you ask for.

Everything is subject to context: operators enforces context to operands, and expressions are evaluated by their context. So if we are forgetting about context, we are doing something wrong for sure.

```
1  sub array_or_string { wantarray ? qw(1 2 3) : 3 }  
2  
3  my @array = array_or_string(); # yields 1 2 3 - list context  
4  my $string = array_or_string(); # gives 3 - scalar context
```

There are mainly 3 types of context in Perl:

- Scalar context
- List context
- void context

Void context is a special case of Scalar context.

```
1 sub array_or_scalar_or_void {
2     !defined wantarray ? print "Void context\n"
3     : wantarray        ? qw(1 2 3)
4     :                   3;
5 }
6
7 my @array = array_or_scalar_or_void(); # yields 1 2 3 - list ↔
8         context
9
10 my ($one, $two) = array_or_scalar_or_void();
11 my ($one, $two, undef) = array_or_scalar_or_void();
12 my $three = array_or_scalar_or_void(); # returns 3 - scalar ↔
13         context
14
15 array_or_scalar_or_void; # will print "Void context".
```

Listing 1: Full example at examples/01_intro/1_context.pl

Perl - “Value” Context

Even if Perl gives the possibility to not specify what a scalar can contain, some operators can enforce context.

```
1 my $number = 42;
2 my $string = "Foo";
3
4 $number    == 42;
5
6 $string    == 'Foo'; # In numeric context, strings evaluate to↔
                        0 (Numeric Coercion)
7 $string    eq 'Foo'; # eq enforces string context
```

Perl - Force “Value” Context

```
1  my $var      = "20";
2
3  my $numeric = 0 + $var; # forces numeric context
4
5  my $string  = '' . $var; # forces string context
6
7  my $bool    = !!$var; # double-bang forces boolean context
8
9
10 my $string = "43 Boo";
11
12 $string--;
13
14 print "$string\n"; # Prints '42'
15
16 my $result = (2**2 + 38) . " is the answer";
```

Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



Introduction - Perl Data structures - Overview

Perl is a peculiar language, and have different kinds of data structures from other languages, and are prefixed with sigils (\$, @, %, &) :

- SCALAR - \$
- ARRAY - @
- HASH - %
- CODE - &
- GLOB / Symbols - the meta-data type.

Perl Data structures - Overview

```
1 my $var = "foo"; $var = "1"; $var = 1 # Always scalar
2
3 my @var = ('var'); # array
4
5 my %var = (bla => boo); # hash
6
7 sub { 1 } # code
8
9 *var = \"bar\" # Glob — ignore it for now
```

it is only the context holding them together ...

Table 1: Even if Perl doesn't distinguish between number or string, it has different operators for the data that the scalar holds

Comparison	Numeric	String
Equal	==	eq
Not Equal	!=	ne
Less Than	<	lt
Greater Than	>	gt
Less Than or Equal To	<=	le
Greater Than or Equal To	>=	ge

Perl Data structures - Basic operations

While declaring strings, you may want to enclose it in double or single quote:

```
1  my $a = 'foo';
2  my $b = "bar";
3  $a = 'it\'s awesome!';
4  my $c = 'ends with a backslash , not a quote: \\\';
5
6  # Double quote forces perl to encode
7  my $tab      = "\t";
8  my $newline  = "\n";
9  my $carriage = "\r";
10 my $backspace = "\b";
11
12 # Interpolation of scalar
13 my $assertion = "$b $a"; # bar it's awesome!
14
15 # qq -> double quotes, q -> single quotes - accepts delimiter
16 my $doublequote = qq{"What the hell" said him};
17 my $singlequote = q^no need to escape "'" !^;
18 my $different_delimiter = q{Different delimiters};
```

Perl Data structures - Basic operations

Access to nested elements

```
1 my @AoA = (  
2     [ "foo", "bar" ],  
3     [ "baz", "geeko", "test" ],  
4 );  
5 print $AoA[1][1];    # prints "geeko"  
6 print $AoA[1]->[1];  # same - more clear  
7  
8 my $ref_AoA = [  
9     [ "foo", "bar" ],  
10    [ "baz", "geeko", "test" ],  
11 ];  
12  
13 print $ref_AoA->[1][1];  # prints "geeko"  
14 print $ref_AoA->[1]->[1];  # same
```

Perl Data structures - Basic operations, contd.

```
1  sub access { print shift()."\n" }
2
3  access(qw( geeko isnotvisible ));
4
5  my @array = qw(geeko is on vacation);
6
7  print "@array[1..2]\n"; # Watch out the context! – prints is ↵
   on
8
9  print @array[1..2]."\n"; # This doesn't do what it's expected!
10
11 print shift(@array)."\n"; # prints geeko, and removes it from ↵
   @array
12
13 print pop(@array)."? \n"; # prints 'vacation?'
14
15 @array = qw(geeko is on vacation);
16
17 print splice(@array, 0, 1)."\n"; # prints 'geeko' and removes ↵
   it (behaves like shift)
18 print "@array\n"; # prints 'is on vacation'
```

Perl Data structures - Basic operations, contd.

```
1  sub access { print shift()."\n" }
2
3  my @array = qw(geeko is on vacation);
4  access(@array);
5
6  print "@array\n"; # unchanged — prints 'geeko is on vacation'
7
8  sub access { print shift(@{$_[0]})."\n" } # prints and removes↔
    the first element of arrayref
9
10 @array = qw(geeko is on vacation);
11 access(\@array);
12
13 print "@array\n"; # prints now 'is on vacation'
14
15 access([@array]); # prints 'is'
16
17 print "@array\n"; # prints still 'is on vacation'
18
19 unshift @array, 'geeko';
20
21 print "@array\n"; # prints 'geeko is on vacation'
```

Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



Killing old myths

Perl have a lot of linguistic shortcuts:

```
1  BEFOREHAND: close door, each window & exit; wait until time.
2      open spellbook, study, select it, confess, tell, deny;
3  write it, print the hex while each watches,
4      reverse "its length", write again;es,
5      values aside, each one;
6      die sheep, die, reverse system
7      you accept (reject, respect)
8      kill spiders, pop them, chop, split, kill them.
9      unlink arms, shift, wait & listen (listening, wait),
10 sort the flock (then, warn "the goats". kill "the sheep");
11 kill them, dump qualms, shift moraliti;
```

Listing 2: extract of an adaptation for Perl5 by ovid of “Black Perl” poem written by Larry Wall

Killing old myths - \$_

\$_ is by definition the *default scalar variable* or, called also *topic variable*. You notice it when a variable is absent. \$_ is the English equivalent of *it*.

```
1  use feature 'say';
2
3  # Equivalent:
4  chomp $_;
5  chomp;
6
7  # If you don't specify a variable, $_ is implied
8  print;
9  say;
10 s/boo/moo/;
11
12 say for qw(1 2 3);
13
14 say for map { $_ * $_ } 1 .. 5;
15 say for grep { /awesome/ } qw(awesome sad);
16 say for grep { /awesome/ } map { $_." is awesome" } qw(Perl ↵
    bar);
```


@_ inside functions, is the copy of the values passed to the function. Is the English equivalent of *them*.

```
1 sub example { print "@_\n"; }
2 sub example2 { print shift."\n"; }
3
4 example qw(This is an example);
5 example2 qw(Only first element will be printed);
```

Killing old myths - undef

Undef represent unassigned, undefined or not known value. Variables that are declared contain undef.

```
1  my $var = undef;    # Not necessary
2  my $var;            # both are undef
```

In boolean context, undef it's false. Common mistake:

Evaluating undef in string context (e.g. interpolation);

```
1  my $undefined;
2  my $text = $undefined . ' bla ';
3  #Yields the classic:
4  #    Use of uninitialized value $undefined in
5  #    concatenation (.) or string...
```

Killing old myths - ()

Empty list in scalar context evaluates undef, but in list context represent an empty list. Consider this code:

```
1 my $count = () = @array;
```

The assignment to context list gets rid of the values in list context, but the assignment is done in scalar context(\$count) such as it's evaluated to the number of items, and the variable holds the number of items in the array.

Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

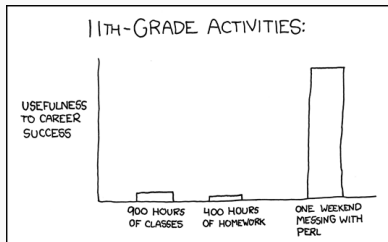
References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



**YES! we have pointers! ... or
something that looks similar to
them**

... not like Python ...



Pointers in Perl - References

Perl does not need variables. We use them to help ourselves, we can still refer to contents only with references. References can be accessed by using '\'

```
1 my $arrayref = \@array;  
2 my $hashref  = \%hash;  
3 my $scalarref = \ $var;
```

Note, you can use '\ ' also to get reference of inline-declared scalars:

```
1 my $scalarref = "\" still valid";
```

Pointers in Perl - References

Dereferencing is the act of getting the real value out of the reference.
As always in Perl, **TIMTOWTDI**.

```
1  my $hashref = { foo => 'bar' } # Reference to inline declared ↔  
    hash  
2  
3  print ${$hashref}{foo}."\n";   # Will print 'bar'  
4  print $$hashref{foo}."\n";    # this too  
5  print $hashref->{foo}."\n";    # this as well
```

Pointers in Perl - References

There is no general rule for dereferencing, but as Perl is context dependent, as a rule of thumb you usually force a sigil context to dereference the variable type.

```
1  my $arrayref = [qw( one two tree )]; # Reference to inline ↔  
    array  
2  
3  print join(" ", @{$arrayref})."\n"; # Will print 'one two ↔  
    tree '  
4  
5  my $scalarref = \"foo"; # Reference to inline scalar  
6  print $$scalarref."\n"; # Will print 'one two tree '
```


Pointers in Perl - References and Context

A good example is Hash slicing. Which is just forcing dereferencing a hash into an array of their values, so you can perform operations directly on the hash. It can be seen as a combination of forcing array dereferencing on part of the hash while imposing list context.

```
1 my $hashref = { test => { 1 => 1, 2 => 2, 3 => 3 } };
2 @{$hashref->{test}}{qw(1 2 3)} = qw(4 5 6);
3 # ^^^ context forced to array.
4
5 my %test = (1 => 1, 2 => 2, 3 => 3);
6 # ^^^ context forced to hash
7 @test{qw(1 2 3)} = qw(4 5 6);
8
9 # Stolen from OpenQA :)
10 @{$job->{settings}}{keys %$worker_settings} = values %←
    $worker_settings; # BAD!
11 # Can you tell why this example *works* but it is a bad ←
    practice?
```

Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

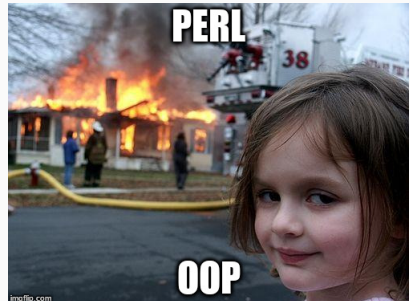
References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



Perl Objects - Introduction

Perl has a large number of libraries giving out Object system in a OOTB fashion:

- Moo
- Moose
- MOP
- Mojo::Base
- Class::* (Rabbit hole)
- even I had my own implementation based off on Mojo::Base
- ...you name it

Perl Objects - Introduction

What is MOP?

- Meta-Object Protocol
- API on top of Object
- Abstraction to normal Object model, yielding to the Reflective property

Why Perl have so many object implementation, and why everything is so complicated?

- TIMTOWTDI lead to a lot of implementations
- Every developer grabbed "something new" from other OOP Model
- Unsatisfaction with general state of OOP in Perl
- Official MOP proposals have been attempted already to land in Perl 5, but never happened.

OOP State in Perl, key points:

- Packages != Objects
- Packages can bring Objects to you
- Basic OOP in Perl allows you to construct your own OOP Model
- Perl is not hiding nothing to you
- DO NOT create Packages that are named all lowercase (reserved to perl pragmas), or all uppercase (Built-in types)

Perl Objects - Are not packages!

Packages are Namespaces that represents a group of symbols that can be identified by its name. Example: `Foo::Bar::Baz` or `Bar::Baz::Foo` - This does not mean any form of inheritance or protection, it's just to identify namespaces, so any package can access other package symbols.

Inside a Namespace, you can refer to inner by not using the fully qualified name, but it's needed outside.

Perl Packages - Are not objects!

The scope of a variable determine the accessibility of that variable. Scopes are determined by Files, by packages thru lexical scopes as you can create new ones by using curly brackets {}

```
1  ...
2      package Baz::Bar;
3
4      my $bar = 'foo'; # Can be only accessed in this context
5      # not e.g. modified using fully qualified name $Baz::Bar::↵
6          bar
7
8      package Baz::Foo;
9
10     # $bar still same
11 ...
```

What makes a package inherit methods from another one?

ISA

Each package contains a special array called **ISA**. The **ISA** array contains a list of that class's parent classes, if any. This array is examined when Perl does method resolution.

Let's have a look on the fundamentals of OOP now.

Perl Objects - Introduction

Everything starts with the **bless** keyword.

```
1  bless REF , CLASSNAME
```

From Perldoc:

"This function tells the **thingy** referenced by REF that it is now an object in the CLASSNAME package" - Thus everything can be "Objectified" :)

Perl Objects - Introduction

Bless is just marking the reference belonging to a Package, thus inheriting the functions (Note: **new()** constructor is just a convention).

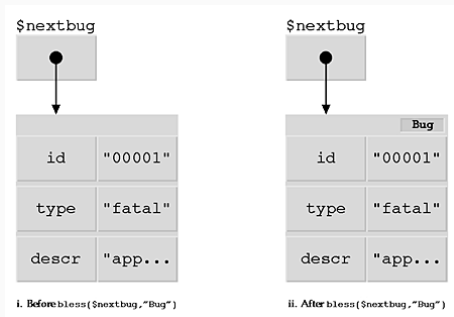


Figure 3: from: Damian Conway's Bless My Referents

That means that a function of a package can return a blessed reference of the type of the Package (thus inheriting functions).

Perl Objects - Introduction

Example Package that offer it's Objectified variant:

```
1  {
2  package Foo;
3  sub new() { bless {}, 'Foo' } # bless {} is equivalent
4  sub newBar() { bless {}, 'Bar' }
5  1;
6  };
7  {
8  package Bar;
9
10 sub printer { shift; print "@_\n" }
11
12 !!42;
13 };
14 my $foo = Foo->new();
15 my $bar = Foo->newBar();
16 my $bar2 = $foo->newBar();
17 Foo->newBar()->printer(" Hello");
```

Perl Objects - Mojo::Base

Mojo::Base object type.

```
1  {
2  package Cat;
3  use Mojo::Base -base;
4
5  has name => 'Nyan';
6  has ['age', 'weight'] => 4;
7  }
8
9  {
10 package Tiger;
11 use Mojo::Base 'Cat';
12
13 has friend => sub { Cat->new }; # Lazy-loaded if no value is ←
    set!
14 has stripes => 42;
15 }
16 # ...
17 my $mew = Cat->new(name => 'Longcat');
18 say $mew->age;
19 say $mew->age(3)->weight(5)->age;
```

Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



Map fun - Schwartzian Transform

The first known online appearance of the Schwartzian Transform is a December 16, 1994 posting by Randal Schwartz.

How to sort this list by last name?

```
1  adjn:Joshua Ng
2  adktk:KaLap Timothy Kwong
3  admg:Mahalingam Gobieramanan
4  admln:Martha L. Nangalama
```

Map fun - Schwartzian Transform

- Lisp roots
- Compact transformation
- Caches expensive calculations
- Variables can be elided
- Pipeline of map-sort-map transforms a data structure into another form easier for sorting and then transforms it back into the first or another form

We have then to sort the list by it's values, not by the keys. . .

Map fun - Schwartzian Transform

Let's assume that we have parsed the list before as a hash (optional, but makes easier to read)

```
1  my %name_list = (  
2      'adjn:Joshua Ng' => 'Ng',  
3      'adtk:KaLap Timothy Kwong' => 'Kwong',  
4      'admng:Mahalingam Gobieramanan' => 'Gobieramanan',  
5      'admin:Martha L. Nangalama' => 'Nangalama',  
6  );  
7  
8  # So, by instinct you would do:  
9  my @sorted_names = sort values %name_list;  
10 # But we want to preserve the data, and not to have only an ↔  
    array of values.
```


Map fun - Schwartzian Transform

Let's first convert the hash into a list of data structures that will keep the information, and will also allow us to sort it easier:

```
1 my @data = map { [ $_, $name_list{$_} ] } keys %name_list;
```

Then we sort it by the hash value contained in that array:

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

sort

The code block supplied to sort receives arguments in \$a and \$b, that are package-scoped variables.. (See perldoc -f sort)

Map fun - Schwartzian Transform

The `cmp` operator performs string comparisons and the `i==j` performs numeric comparisons. Now we put the data in a form that we use it to display:

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

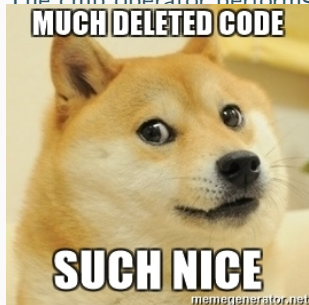
thus, the whole operation can be shortened as:

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

Reading from right to left: For each key in the hash create an anonymous array of two items containing the key and the value, then sort the list of arrays by their second element - Then format the string.

Map fun - Schwartzian Transform

The cmp operator performs string comparisons and the j=i performs we put the data in a form that we use it to



```
ap { "$_ -> [1], ext. $_ -> [0]" } ←
```

can be shortened as:

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

Reading from right to left: For each key in the hash create an anonymous array of two items containing the key and the value, then sort the list of arrays by their second element - Then format the string.

Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

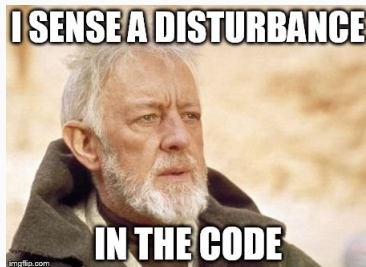
References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



Idioms and common patterns can be found in any language.

Perl may have too many of them.

Perl has idioms, some of them are neat, some of them looks like casting spells. They are actually language features and design techniques - makes your code look “Perlish”. You don't need them, but they leverage Perl features to get the job done :)

Some of them are common rules:

- Object are represented (inside methods) as `$self`
- Named Parameters preferred vs anonym ones e.g.
`func(option=>1,option2=>2)`
- Hash or Hash Ref? To avoid odd sized list to be interpreted as Hash, better Hashrefs (also for inspection).

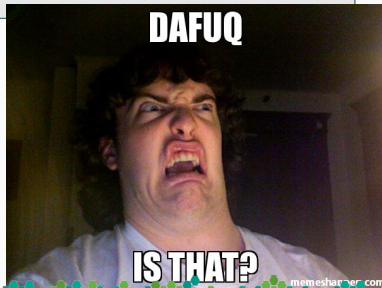
Some of them are driven by design - we will look at few examples of them.

Idioms - Globs - Automa(g/t)ic assignment

```
1
2 *var = \"test\";
3
4 no strict 'refs';
5
6 print ${*{"var"}}{SCALAR}}.\"\\n\";
```

Idioms - Globs - Automa(g/t)ic assignment

```
1
2 *var = \"test\";
3
4 no strict 'refs';
5
6 print $*{"var"}{SCALAR}."\n";
```



Being able to manipulate and force context, allows us to have few tricks:

```
1 my $count = ()= @array; # Forces list context
2 my $count = scalar @array; # Forces scalar context
3 my $one = (qw(1 2 3))[0]; # Force list context and retrieve an↔
    element in array
4
5 my $string = '42 is the right answer';
6 my $back_to_number =0+ $string; # 42 (Venus operator)
7 $string--; # $string now is 41
8 my $bangbang      = !!$string; # 1
9
10 @{[ qw(1 2 3) ]}; # Baby cart operator
```

And always thanks to Perl context fun, we can create arrays or hashes based on variable options easily:

```
1  my $dog = 1; # Try to flip them!
2  my $cat = 1;
3  my @array = (
4      ('wof' ) x !( $dog) ,
5      ('meow' ) x !( $cat)
6  );
7
8  $dog = 1;
9  $cat = 1;
10 my %hash = (
11     (wof => $dog) x !( $dog) ,
12     (meow => $cat) x !( $cat)
13 );
```

`x` is the string multiplier, and the double bang (`!!`) reduces the expression in the right to a boolean.

Table of contents

Introduction

Context

Perl Data Structures

Killing old myths

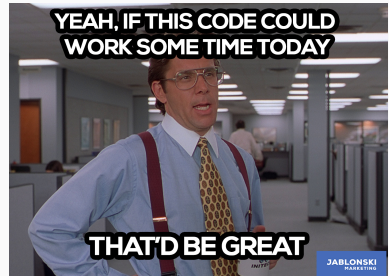
References

Perl Objects (Moo, Moose, Mojo::Base, MOP, ...)

Map fun

Idioms

Common pitfalls



Common pitfalls

To not break version compatibility, Perl still supports very old syntax.

```
1 # outdated style; avoid
2 my $result = &calculate_result( 52 );
3 # very outdated; truly avoid
4 my $result = do &calculate_result( 42 );
```

[1]

- Disables prototypes
- It passes @_ if no arguments are specified

Making a good usage of parenthesis, it helps both Perl and you.

```
1  nasty_func weird_func 4, @array, 'Postfix';
```

- Perl heuristics can be confusing
- `weird_func` will gobble up everything.

Common pitfalls

`==` it's different from `eq` and the relationship with them can be somewhat confusing.

`eq` can work both on numbers and on string - this only because stringification can be inferred on numbers.

```
1 say "It's true!" if "Boo" == "Baaz"; # Throws a warning, but ↵  
    will always work  
2 # while, 'eq' still works on numbers, because string ↵  
    representation of numbers can be easily inferred  
3 say "Works" if 2 eq 2;
```

Common pitfalls - Memory Leaking

Perl has a garbage collector, that reaps the references that are not anymore in scope, calling **DESTROY** method on them. Due to this design, it's possible to have memory leaking. How it is possible? What happens in circular reference scenarios?

Common pitfalls - Identify memory leaking code

Consider this code snippet:

```
1 sub circular_ref {  
2     my $a;  
3     my $b;  
4  
5     $a->{b} = \"$b;  
6     $b->{a} = \"$a;  
7 }  
8  
9 circular_ref() for 1..100000;
```

Will be able the Perl **GC** to reap all the unused memory? - short answer,
NO!

Common pitfalls - Identify memory leaking code



```
5     $a->{b} = \"$b;  
6     $b->{a} = \"$a;  
7 }  
8  
9 circular_ref() for 1..100000;
```

Will be able the Perl **GC** to reap all the unused memory? - short answer, NO!

Common pitfalls - Avoid to leak memory in your code!

Use `weaken` to escape to circular references, so GC doesn't loop while trying to clean up the references out of scope.

```
1  use Scalar::Util 'weaken';
2
3  sub circular_ref {
4      my $a;
5      my $b;
6
7      $a->{b} = \$b;
8      $b->{a} = \$a;
9      weaken $b->{a};
10 }
11
12 circular_ref() for 1..100000;
```

Common pitfalls - Analyze Code that may leak

Wrap the statement that you think could leak memory with
`Memory::Usage`.

```
1  use Memory::Usage;
2
3  my $mu = Memory::Usage->new();
4  my $mu_w = Memory::Usage->new();
5
6  $mu_w->record('before');
7
8  circular_ref() for 1..100000;
9
10 $mu_w->record('after');
11
12 $mu_w->dump();
13
14 sub circular_ref {
15     my $a;
16     my $b;
17
18     $a->{b} = \$b;
19     $b->{a} = \$a;
20 }
```

Common pitfalls - Analyze Code that may leak

```
+ 99_extra git:(master) x perl mem_leak.pl
time   vsz ( diff)   rss ( diff) shared ( diff)   code ( diff)   data ( diff)
0 18292 ( 18292) 4608 ( 4608) 3900 ( 3900) 1944 ( 1944) 1084 ( 1084) before
0 59080 ( 40788) 45324 ( 40716) 3964 ( 64) 1944 ( 0) 41872 ( 40788) after
+ 99_extra git:(master) x perl mem_leak_weaken.pl
time   vsz ( diff)   rss ( diff) shared ( diff)   code ( diff)   data ( diff)
0 20672 ( 20672) 5336 ( 5336) 4368 ( 4368) 1944 ( 1944) 1368 ( 1368) before
0 20672 ( 0) 5336 ( 0) 4368 ( 0) 1944 ( 0) 1368 ( 0) after
+ 99_extra git:(master) x
```

Importing

The Perl **use** built-in call automatically the `import()` function on the class that is supplied. Modules then, can provide their own `import` that can make visible to the caller some or all functions in it's Package scope.

So:

```
1 use strict; # strict->import();
2 use foo; # foo->import();
3 use bar; # bar->import();
4 use strict 'refs'; # strict->import('refs');
5 use strict qw(subs vars); # strict->import('subs','vars');
```

Importing

The Perl **use** built-in call automatically the `import()` function on the class that is supplied. Modules then, can provide their own `import` that can make visible to the caller some or all functions in it's Package scope.

So:

```
1 use strict; # strict->import();
2 use foo; # foo->import();
3 use bar; # bar->import();
4 use strict 'refs'; # strict->import('refs');
5 use strict qw(subs vars); # strict->import('subs','vars');
```

So this has the same effect :

```
1 BEGIN {
2     require strict;
3     strict->import( 'refs' );
4 }
```

At this point - the role of Exporter should be a bit more easier to gasp.

- The **use** pragma is calling `import()` functions on the Specified class
- The `import()` function can do virtually *anything*
- it is done in Compile time, so Perl can parse the tree and do not throw odd errors - like bareword key not found
- ...so if we define an `import()` in our package, we could emulate Exporter's behavior!

Killing old myths - Exporter

We now have all the elements to implement our own Exporter module.

Of course, we can also use it without having to inherit from it, or we can also use Exporter's functionalities without inheriting from the Class.

So if we want to export some functions with Exporter:

```
1 package Foo;
2 our @EXPORT_OK = qw(baz);
3 use base Exporter; # Do not use base! and it's Not needed! ←
   This will just inherit the Exporter import(), the only ←
   thing needed to make it work.
4 use parent Exporter; # Again, do not need to inherit ←
   completely from it.
5 use Exporter 'import'; # Compact and exactly what it's just ←
   needed! Exporter's import();
```

Exporter import() is looking into @EXPORT_OK and in @EXPORT for functions to export into the caller's namespace.

How hard than can be to replicate the Exporter's functionalities?
Can't we make our own Exporter as well?

Killing old myths - CustomExporter

How hard than can be to replicate the Exporter's functionalities?

Can't we make our own Exporter as well?

Yes we can!



Killing old myths - CustomExporter

Let's try now to add capabilities to an our package to export all symbols into the Packages that will load it on compile phase.

Killing old myths - CustomExporter

Let's try now to add capabilities to an our package to export all symbols into the Packages that will load it on compile phase.

Our Package then should have an import method, that will take care of exporting all the desired functions into the caller namespace. Will look something like this:

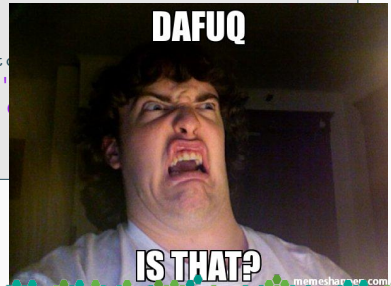
```
1 package CustomExporter;
2 use strict;
3
4 sub import {
5     no strict 'refs';
6     do {
7         next if not defined *{$CustomExporter::{$_}}{CODE} ←
8                               or $_ eq 'import';
9         *{caller() . "::$$_"} = \&{"CustomExporter::$_"};
10    } for keys %CustomExporter::;
11 }
```

Killing old myths - CustomExporter

Let's try now to add capabilities to our package to export all symbols into the Packages that will load it on compile phase.

Our Package then should have an import method, that will take care of exporting all the desired functions into the caller namespace. Will look something like this:

```
1 package CustomExporter;
2 use strict;
3
4 sub import {
5     no strict 'refs';
6     do {
7         next if not defined *{$CustomExporter::} or $_ eq '
8         *{caller() . "::$-"} = \&{"
9     } for keys %CustomExporter::;
10 }
```



Killing old myths - CustomExporter

Our Commented and cleaned Exporter:

```
1 package CustomExporter;
2 use strict;
3
4 sub import {
5     my $pkg = caller; # who's calling us?
6     no strict 'refs'; # Avoid strict to be dushbag with ↵
7         symbolic dereference
8
9     foreach my $glob (keys %CustomExporter::) { # Get all meta↵
10         data-types in the package
11         next if not defined *{$CustomExporter::{ $glob }}{CODE} ↵
12             # We want to put in caller only functions
13             or $glob eq 'import'; # But not import itself
14
15         *{$pkg . "::$glob"} = \&{"CustomExporter::$glob"}; # ↵
16             Create a glob for the package.
17     }
18     # Note, this is a bit magical, since globs can inference ↵
19     the assigned type.
20 }
```

Killing old myths - CustomExporter

Our Commented and cleaned Exporter:

```
1 package CustomExporter;
2 use strict;
3
4 sub import {
5     my $pkg = caller; # who's calling us?
6     no strict 'refs'; # Avoid strict to be dushbag with ↵
7                         symbolic dereference
8
9     foreach my $glob (keys %CustomExporter::) { # Get all meta↵
10        data-types in the package
11        next if not defined *{$Custo
12            # We want to put in cal
13            or $glob eq 'import
14
15        *{$pkg . "::$glob"} = \&{"C
16            Create a glob for the p
17
18        # Note, this is a bit magical,
19        the assigned type.
20    }
21 }
```



Killing old myths - OOP - Liskov

The Liskov substitution principle suggest that an object should be as more general as possible with the expectations, and at least as specific about what it produces as the object it replaces.

What this means? Suppose we have two classes:

```
1 {  
2   package Fruit;  
3   # ... object implementation  
4 }  
5 {  
6   package Apple;  
7   use parent Fruit;  
8   # ... object implementation  
9 }
```

The Liskov principle enforces objects design. To satisfy the Liskov principle, in a testsuite we should be able to replace Apple with Fruit, without the need of further changes.

Killing old myths - Easy File Slurping

local is essential to managing Perl's magic global variables. Scoping is important to use local effectively but if you do, you can use tight and lightweight scopes in interesting ways. For example, to slurp files into a scalar in a single expression:

```
1
2   my $file = do { local $/; <$fh> };
3
4   # or
5   my $file; { local $/; $file = <$fh> };
```

\$/

`$/` is a string of zero or more characters which denotes the end of a record when reading input a record at a time. By default, this is your platform-specific newline character sequence. See Modern perl for a complete reference

...

HOW ABOUT A NICE CUP OF



CODE REVIEW

memegenerator.net



Modern Perl.

Onyx Neon Press, 2016.

Available at

http://modernperlbooks.com/books/modern_perl_2016/.