Quick? Perl Intro

Francisco Jurado

2017/08/15

Outline

Topic

About

- Based on the Modern Perl book

About

- Based on the Modern Perl book
- Quick introduction of the Perl language

About

- Based on the Modern Perl book
- Quick introduction of the Perl language
- Provide the information to start writing simple code

Topic

Motivation (virtues of a programmer)

- Laziness

Motivation (virtues of a programmer)

- Laziness
- Impatience

Motivation (virtues of a programmer)

- Laziness
- Impatience
- Hubris

Motivation (Pragmatic)

- Perl is a language for getting your job done

Motivation (Pragmatic)

- Perl is a language for getting your job done
- It will mold itself to do what you mean.

Motivation (Pragmatic)

- Perl is a language for getting your job done
- It will mold itself to do what you mean.
- Won't enforce programming paradigm

Motivation (Pragmatic)

- Perl is a language for getting your job done
- It will mold itself to do what you mean.
- Won't enforce programming paradigm
- . . . Because only you know what you need.

Motivation (Principles)

- Things that are different, should look different

Motivation (Principles)

- Things that are different, should look different
- Common constructions should be short

Motivation (Principles)

- Things that are different, should look different
- Common constructions should be short
- Important information should come first

Motivation (Motto)

- TIMTOWTDI (Tim Toady)

Motivation (Motto)

- TIMTOWTDI (Tim Toady)
- Easy things should be easy and hard things should be possible

Baby Perl

- Analogy when a person starts learning a new spoken language

Baby Perl

- Analogy when a person starts learning a new spoken language
- You start by learning a few words, then start building simple sentences

Baby Perl

- Analogy when a person starts learning a new spoken language
- You start by learning a few words, then start building simple sentences
- Continue with simple conversations

Baby Perl

- Analogy when a person starts learning a new spoken language
- You start by learning a few words, then start building simple sentences
- Continue with simple conversations
- Don't need to know a whole language to express ideas and concepts

Baby Perl

- Analogy when a person starts learning a new spoken language
- You start by learning a few words, then start building simple sentences
- Continue with simple conversations
- Don't need to know a whole language to express ideas and concepts
- Keep practicing and become a native speaker

Topic

CPAN The Comprehensive Perl Archive Network

- `http://www.cpan.org`

CPAN The Comprehensive Perl Archive Network

- `http://www.cpan.org`
- `http://www.metacpan.org`

CPAN The Comprehensive Perl Archive Network

- `http://www.cpan.org`
- `http://www.metacpan.org`
- Tens of thousands of reusable packages

CPAN The Comprehensive Perl Archive Network

- `http://www.cpan.org`
- `http://www.metacpan.org`
- Tens of thousands of reusable packages
- All kinds of problem solvers

Perl Sites

- Perl's homepage `http://www.perl.org`

Perl Sites

- Perl's homepage `http://www.perl.org`
- Perl Monks `http://perlmonks.org` (25 years)

Perl Sites

- Perl's homepage `http://www.perl.org`
- Perl Monks `http://perlmonks.org` (25 years)
- Perl blogs `http://blogs.perl.org`

Perl Sites

- Perl's homepage `http://www.perl.org`
- Perl Monks `http://perlmonks.org` (25 years)
- Perl blogs `http://blogs.perl.org`
- Perl Weekly `http://perlweekly.com`

Perl Sites

- Perl's homepage `http://www.perl.org`
- Perl Monks `http://perlmonks.org` (25 years)
- Perl blogs `http://blogs.perl.org`
- Perl Weekly `http://perlweekly.com`
- Perl Buzz `http://perlbuzz.com`

IRC and Events

- Yet Another Perl Conference `http://yapc.org`
- IRC Server: `irc://irc.perl.org`
- Channels: #perl-help #perl-qa #perl (also in Freenode)

Topic

Download and Install

Unix

Installed in most Unix Systems

Windows

Strawberry Perl `http://strawberryperl.com/`

App::cpanminus

Unix

curl -L `https://cpanmin.us` | perl - App::cpanminus

Windows

cpan App::cpanminus

Devel::REPL

- Nice to have for this presentation

Install with cpanm (see previous slide)

```
cpanm Devel::REPL
```

Topic

perldoc

- Documentation written in POD (sometimes embedded to source code)
- Command line tool to view Perl documentation (language, modules, etc)
- Example:

Command line

```
$ perldoc perl
$ perldoc -l List::Util
$ perldoc -m List::Util
$ perldoc -f map
$ perldoc -v '$/'
```

Built-in Data Structures (sigils)

- Scalars variables ($)

Built-in Data Structures (sigils)

- Scalars variables (`$`)
- Array variables (`@`)

Built-in Data Structures (sigils)

- Scalars variables ($)
- Array variables (@)
- Hash variables (%)

Built-in Data Structures (sigils)

- Scalars variables (`$`)
- Array variables (`@`)
- Hash variables (`%`)
- They allow to separate variables into different namespaces: `$name`, `@name`, `%name`

Context

- Governs how many items you expect from an operation

Amount Context (void, scalar, list)

```
build_sales_report();                    # void
my $earnings  = build_earnings_report(); # scalar
my @details   = build_earnings_report(); # list
my ($report)  = build_earnings_report(); # list
process( build_earnings_report() );      # list
save_amount( scalar build_earnings_report() )
```

Type Context (String, Numeric, Boolean)

- Defines how Perl interprets a piece of data
- In numeric context strings that don't look like numbers evaluate to 0

Example (re.pl)

```
my $name = 'Francisco'

say 'Really the same person?'
  if $name  == 'Frank';    # numeric context
say 'Definetly not the same person'
  unless $name eq 'Frank'; # string context
say 'He does exist though...'
  if $name;                # Boolean context
```

Perl Pronouns: Default Scalar $_ (topic)

- Most notable in its absence, many builtin operations work on this variable
- Equivalent to the pronoun *it*

Code

```
<$fh>   # read ...it
uc      # Upper case ...it
```

Perl Pronouns: Default Scalar $_ (topic)

- Multiple built-ins operate on this variable: uc, say, print, lc, length, Perl Regex.
- Looping directives default to $_ as the iteration variable

Code (re.pl)

```
say for 1 .. 10;
say uc reverse while (<STDIN>);
```

Perl Pronouns: Default Arrays

- `@_` is equivalent to the pronoun *they* and *them*.

Perl Pronouns: Default Arrays

- @_ is equivalent to the pronoun *they* and *them*.
- @ARGV contains the command line arguments to the program.

Perl Pronouns: Default Arrays

- `@_` is equivalent to the pronoun *they* and *them*.
- @ARGV contains the command line arguments to the program.
- `shift` and `pop` operate on these to variable by default.

Perl Pronouns: Default Arrays

- `@_` is equivalent to the pronoun *they* and *them*.
- @ARGV contains the command line arguments to the program.
- `shift` and `pop` operate on these to variable by default.
- When operating on an empty filehandle, each element of @ARGV will be treated as a file name to open for reading.

Topic

Names

- Used for naming everything: functions, variables, packages, etc.

Names

- Used for naming everything: functions, variables, packages, etc.
- Valid names begin with underscore or a letter

Names

- Used for naming everything: functions, variables, packages, etc.
- Valid names begin with underscore or a letter
- Followed by any combination of letters, numbers and underscores

Names

- Used for naming everything: functions, variables, packages, etc.
- Valid names begin with underscore or a letter
- Followed by any combination of letters, numbers and underscores
- invalid names `‘invalid name‘`, `3rdStrike`, `~rare`, `lisp-like-name`

Variables (sigils again)

- The sigil defines the type of aggregate data structure we're trying to access

Variables (sigils again)

- The sigil defines the type of aggregate data structure we're trying to access
- Governs to manipulate the data of a variable

Variables (sigils again)

- The sigil defines the type of aggregate data structure we're trying to access
- Governs to manipulate the data of a variable
- The $ will access a single element as accessor or as lvalue

Variables (sigils again)

- The sigil defines the type of aggregate data structure we're trying to access
- Governs to manipulate the data of a variable
- The $ will access a single element as accessor or as lvalue
  - Of an hash: `$hash{ $key }`

Variables (sigils again)

- The sigil defines the type of aggregate data structure we're trying to access
- Governs to manipulate the data of a variable
- The $ will access a single element as accessor or as lvalue
  - Of an hash: `$hash{ $key }`
  - Of an array: `$array[ $index ]`

Namespaces

- Namespace is a collection of symbols grouped under a globally unique name

Namespaces

- Namespace is a collection of symbols grouped under a globally unique name
- Multi-level namespaces are allowed by joining names with `::` like in `Modern::Perl`

Namespaces

- Namespace is a collection of symbols grouped under a globally unique name
- Multi-level namespaces are allowed by joining names with `::` like in `Modern::Perl`
- Within the namespace the only symbol name can be used

Namespaces

- Namespace is a collection of symbols grouped under a globally unique name
- Multi-level namespaces are allowed by joining names with `::` like in `Modern::Perl`
- Within the namespace the only symbol name can be used
- Outside the namespace use the fully qualified name

Namespaces

- Namespace is a collection of symbols grouped under a globally unique name
- Multi-level namespaces are allowed by joining names with `::` like in `Modern::Perl`
- Within the namespace the only symbol name can be used
- Outside the namespace use the fully qualified name
- The default namespace is main

Values (Strings)

- Piece of textual or binary data with no particular formatting, delimited by single or double quotes.

Values (Strings)

- Piece of textual or binary data with no particular formatting, delimited by single or double quotes.
- For escaping use backslash \.

Values (Strings)

- Piece of textual or binary data with no particular formatting, delimited by single or double quotes.
- For escaping use backslash `\`.
- To use escaped sequence of meta-characters do it under double quotes `"\t \n \b"`.

Values (Strings)

- Piece of textual or binary data with no particular formatting, delimited by single or double quotes.
- For escaping use backslash `\`.
- To use escaped sequence of meta-characters do it under double quotes `"\t \n \b"`.
- Concatenate stings with the concatenation operator ‘ . ‘

Values (Strings)

- Piece of textual or binary data with no particular formatting, delimited by single or double quotes.
- For escaping use backslash \.
- To use escaped sequence of meta-characters do it under double quotes "\t \n \b".
- Concatenate stings with the concatenation operator ‘ . ‘
- use qq{} as an altenative to ("") to prevent escaping repeatedly the double quotes

Values (Strings)

- Piece of textual or binary data with no particular formatting, delimited by single or double quotes.
- For escaping use backslash \.
- To use escaped sequence of meta-characters do it under double quotes "\t \n \b".
- Concatenate stings with the concatenation operator ‘ . ‘
- use qq{} as an altenative to ("") to prevent escaping repeatedly the double quotes
- use q{} as an alternative to (') to prevent escaping repeatedly the single quote

Values (Strings)

- Piece of textual or binary data with no particular formatting, delimited by single or double quotes.
- For escaping use backslash \.
- To use escaped sequence of meta-characters do it under double quotes "\t \n \b".
- Concatenate stings with the concatenation operator ‘ . ‘
- use qq{} as an altenative to ("") to prevent escaping repeatedly the double quotes
- use q{} as an alternative to (') to prevent escaping repeatedly the single quote
- Heredocs are available

Values (Strings)

Code

```
   my $single_quoted = 'To be or not to be.';
   my $double_quoted = "Interpolate $name.\n";

   say $single_quoted . $double_quoted;

   # see perlop for good examples of heredocs
   # ========================================
   my $here_doc_str =<<"END_STRING"
   This is a text where variables can be
   interpolated for example the variable
   \$number has a value of $number
END_STRING
```

Values (Numbers)

- Support integers and floating-point values, can be any popular notation
- Support of _ as number separator: 1_000_000_000
- Perl treats everything that looks like a number as a number in numeric context

Numbers in Perl

```
my $integer = 5;
my $float = 0.01;
my $sci_float = 1.02e10;
my $binary = 0b1101;
my $octal = 012;
my $hex = 0x12;
```

Values (undef)

- Represents an unassigned, undefined and unknown value

Values (undef)

- Represents an unassigned, undefined and unknown value
- To set a variable to an undefined value: `$var = undef;`

Values (undef)

- Represents an unassigned, undefined and unknown value
- To set a variable to an undefined value: `$var = undef;`
- To test for a defined value: `defined $var;`

Values (Lists)

- () Denote a list, in scalar context evaluates to undef
- () in list context it's an empty list and in lvalue imposes list context
- `my $count = () = get_list_of_colors();`
- The comma operator (,) creates a list, and it has very low precedence
- The range operator also creates lists (. .)

Code (re.pl)

```
my @numbers  = (1, 2, 3, 4, 5);
my @numbers2 =  1 .. 5;
my @farm    = qw!horse chicken goat pig cow!
my ($package, $filename, $line) = caller();
```

Scalars

- Fundamental data type: single, discrete value. String, number, fh, reference, etc.

Scalars

- Fundamental data type: single, discrete value. String, number, fh, reference, etc.
- Identified by the $ sigil

Scalars

- Fundamental data type: single, discrete value. String, number, fh, reference, etc.
- Identified by the $ sigil
- Any value type can be [re]assigned arbitrarily

Scalars

- Fundamental data type: single, discrete value. String, number, fh, reference, etc.
- Identified by the $ sigil
- Any value type can be [re]assigned arbitrarily
- Contains a numeric slot and a string slot

Scalars

- Fundamental data type: single, discrete value. String, number, fh, reference, etc.
- Identified by the $ sigil
- Any value type can be [re]assigned arbitrarily
- Contains a numeric slot and a string slot
- Subject to string interpolation

Arrays

- Arrays are used to store a collection of scalars

Arrays

- Arrays are used to store a collection of scalars
- Access by index starting on 0

Arrays

- Arrays are used to store a collection of scalars
- Access by index starting on 0
- Use the scalar sigil to access an individual element

Arrays

- Arrays are used to store a collection of scalars
- Access by index starting on 0
- Use the scalar sigil to access an individual element
- They grow or shrink as you manipulate them

Arrays

- Arrays are used to store a collection of scalars
- Access by index starting on 0
- Use the scalar sigil to access an individual element
- They grow or shrink as you manipulate them
- `each` operator can iterate over an array

Arrays

- Arrays are used to store a collection of scalars
- Access by index starting on 0
- Use the scalar sigil to access an individual element
- They grow or shrink as you manipulate them
- `each` operator can iterate over an array
- Slices : access multiple elements at the same time

Arrays

- Arrays are used to store a collection of scalars
- Access by index starting on 0
- Use the scalar sigil to access an individual element
- They grow or shrink as you manipulate them
- `each` operator can iterate over an array
- Slices : access multiple elements at the same time
- In list context, arrays flatten into lists

Arrays

- Arrays are used to store a collection of scalars
- Access by index starting on 0
- Use the scalar sigil to access an individual element
- They grow or shrink as you manipulate them
- `each` operator can iterate over an array
- Slices : access multiple elements at the same time
- In list context, arrays flatten into lists
- Array interpolate into strings as the stringification of each element separated by $"

Code Example for Arrays (re.pl)

```
my @zero_to_nine = 0 .. 9;

# Single element access
$zero_to_nine[5];

# select the last element
$zero_to_nine[ $#zero_to_nine ]
$zero_to_nine[ @zero_to_nine-1 ]
$zero_to_nine[-1]
```

Code Example for Arrays (re.pl)

```
# Arrays in different contexts
# scalar assignment
$count = @zero_to_nine;

#scalar string concatenation
say 'I got ' . @zero_to_nine . ' numbers';

#string interpolation
say "These are my @zero_to_nine numbers";

#boolean context
say 'I do have numbers.' if @zero_to_nine;
```

Code Example for Arrays (re.pl)

```
#Slice
my @indexes =  (8, 4 .. 6, 2);
my @selected_data = @zero_to_nine[ @indexes ];

# indexes in splices is evaluated in list context
@zero_to_nine[ @indexes ] = (0) x @indexes;
```

Array Operations (destructive) (See examples)

- `push`: Add to the end
- `pop`: Pull from the end
- `unshift`: Push to the front
- `shift`: Pull from the front
- `splice`: remove, replaces elements from an array

Push Example

```
# Merge multiple arrays in one push
push my @pets, @dogs, @cats, @brids;
```

Hashes

- Also known as tables, associative arrays, dictionaries, etc

Hashes

- Also known as tables, associative arrays, dictionaries, etc
- This structure has no order, don't rely on this.

Hashes

- Also known as tables, associative arrays, dictionaries, etc
- This structure has no order, don't rely on this.
- Declare with the % sigil

Hashes

- Also known as tables, associative arrays, dictionaries, etc
- This structure has no order, don't rely on this.
- Declare with the % sigil
- Access a single element with $ and {}

Hashes

- Also known as tables, associative arrays, dictionaries, etc
- This structure has no order, don't rely on this.
- Declare with the % sigil
- Access a single element with $ and {}
- Test key existence with `exists`

Hashes

- Also known as tables, associative arrays, dictionaries, etc
- This structure has no order, don't rely on this.
- Declare with the % sigil
- Access a single element with `$` and `{}`
- Test key existence with `exists`
- Use `keys %hash` to extract a list containing the `%key` values

Hashes

- Also known as tables, associative arrays, dictionaries, etc
- This structure has no order, don't rely on this.
- Declare with the % sigil
- Access a single element with `$` and `{}`
- Test key existence with `exists`
- Use `keys %hash` to extract a list containing the `%key` values
- Use `values %hash` to extract a list containing the `%hash` valeus

Hashes

- Iterate over a hash with `each %hash`, it'll return a key/value pair and undef when done

Hashes

- Iterate over a hash with `each %hash`, it'll return a key/value pair and undef when done
- To reset the iterator use either `keys` or `values`

Hashes

- Iterate over a hash with `each %hash`, it'll return a key/value pair and undef when done
- To reset the iterator use either `keys` or `values`
- Slices just like in arrays but using braces

Hashes

- Iterate over a hash with `each %hash`, it'll return a key/value pair and undef when done
- To reset the iterator use either `keys` or `values`
- Slices just like in arrays but using braces
- `@hash{ @keys }`

Hashes

- Iterate over a hash with `each %hash`, it'll return a key/value pair and undef when done
- To reset the iterator use either `keys` or `values`
- Slices just like in arrays but using braces
- `@hash{ @keys }`
- To initialize a hash:

Hashes

- Iterate over a hash with `each %hash`, it'll return a key/value pair and undef when done
- To reset the iterator use either `keys` or `values`
- Slices just like in arrays but using braces
- `@hash{ @keys }`
- To initialize a hash:
- `%initialized_hash = map { $_ => 1 } @keys`

Hashes Code Example

```
# initialize a hash
my %music_genere_for = (
  # watch for the fat comma '=>'
  'the beatles' => "rock",
   rush         => "progressive rock",
 );

# access a single element
say $music_genere_for{'the beatles'};

join ', ', keys   %music_genere_for;
join ', ', values %music_genere_for;
```

Hash idioms

```
# To merge two hashes, use slices
# you can rely in the order of keys and values
@hash_1{ keys %hash_2 } = values %hash_2;

# extract uniq values in an array
my %uniq;

undef @uniq{ @items };
my @unique_items = keys %uniq;
```

Lexical Scope

- Lexical scope governed by the syntax of the program usually within ({ and }) or entire files

Lexical Scope

- Lexical scope governed by the syntax of the program usually within ({ and }) or entire files
- Declared with `my`

Lexical Scope

- Lexical scope governed by the syntax of the program usually within ({ and }) or entire files
- Declared with `my`
- They are visible in the declaration scope and in any scopes nested within it

Lexical Scope

- Lexical scope governed by the syntax of the program usually within ({ and }) or entire files
- Declared with `my`
- They are visible in the declaration scope and in any scopes nested within it
- Not visible in sibiling scopes

Lexical Scope (Our)

- Creates a local alias to a package variable and still enforces lexical scoping of the alias

Lexical Scope (our) Example

```
package Fun::Package {
  our $fun = "howdy our!";
  say $fun;

  package Fun::Package::Nested {
    say "This is nested fun $fun";
  }
};
package Another::Package {
  say "Another::Package $Fun::Package::fun"
}
package main;
say "From main: ", $Fun::Package::fun;
```

Dynamic Scope

- Also applies to visibility but instead of looking on compile time scopes, lookup through the calls stack

Dynamic Scope

- Also applies to visibility but instead of looking on compile time scopes, lookup through the calls stack
- Dynamic scope applies only to global and package global variables

Dynamic Scope

- Also applies to visibility but instead of looking on compile time scopes, lookup through the calls stack
- Dynamic scope applies only to global and package global variables
- While a package global variable may be visible within all scopes, its value may change depending on `local`-ization

Dynamic Scope (Example)

Dynamic Scope Example

```
our $scope;
sub inner {
  say $scope;
}
sub middle {
  say $scope;
  inner();
}
sub main {
    say $scope;
    local $scope = 'main() scope';
    middle();
}
$scope = 'outer scope';
main();
say $scope;
```

The output

```
outer scope
main() scope
main() scope
outer scope
```

Scope (State)

- `state` Declares a lexical variable which has a one time initialization

Scope (State) Example

```
use feature qw/state say/

sub sub_with_state {
  state $state = 10;
  return $state++;
}

say sub_with_state for 1 .. 10;
```

The output

```
10
11
12
13
14
15
16
17
18
19
```

Control Flow (Conditionals)

- The condition is evaluated in boolean context

Conditionals Example

```
# prefix form
if ($true_val)  {
    say "This is true";
}

unless ($true_val) {
    say "This is false";
}

# postfix form
say "This is true" if ($true_val);
say "This is false" unless ($true_val);

# Ternary conditional operator
my $time_postfix = after_noot($time) ? 'PM' : 'AM';
```

Control flow (Loops: for foreach)

- The for loop aliases the iterator variable to the values in the iteration

Loops Example (for foreach)

```
# C style
for (my $i = 0; $i <= 10; $i++) {
   say "$i * $i = ", $i * $i;
}

# Prefix notation
foreach(1 .. 10) {
  say "$_ * $_ = ", $_ * $_;
}

# Postfix notation
say "$_ * $_ = ", $_ * $_ for 1 .. 10;

# named lexical iterator
foreach my $num (1 .. 10) {
  say "$num * $num = ", $num * $num;
}
```

Control flow (Loops: while and until)

Loops Example (while until)

```
# shifting in the control block
while (@values) {
  say( shift @values );
}

# shift in the condition
while (my $value = shift @values) {
  say $value;
}

until ($finished) {
   $finished = finished_yet();
}

# iterate over an open filehandle
# this construct is equivalent to while (defined($_ = <$fh>)) {}
while (<$fh>) {
    chomp and say;
}
```

Control flow (Loop control)

Loop Control Example

```
# loop control
while (<$fh>) {
  next if /\A#/;
  last if /\A__END__/;
}

# named loops and continue
# SEE EXAMPLES
LINE:
while (<$fh>) {
    chomp;

    PREFIX
    for my $prefix (@prefixes) {
        next LINE unless $prefix;
    }
}
continue {
  say "Force the execution of this block ....";
}
```

Coercion Context

- Boolean Coercion

Coercion Context

- Boolean Coercion
- String Coercion

Coercion Context

- Boolean Coercion
- String Coercion
- Numeric Coercion

Coercion Context

- Boolean Coercion
- String Coercion
- Numeric Coercion
- Reference Coercion (Autovivification)

Coercion Context

- Boolean Coercion
- String Coercion
- Numeric Coercion
- Reference Coercion (Autovivification)
- Scalar::Util::dualvar to manipulate scalar variable coercion

Packages

- Encapsulation of named entities in a single namespace

Packages

- Encapsulation of named entities in a single namespace
- `package` declares a package and a namespace

Packages

- Encapsulation of named entities in a single namespace
- `package` declares a package and a namespace
- Everything declared within a `package` block refer to symbols in that package's table

Packages

- Encapsulation of named entities in a single namespace
- `package` declares a package and a namespace
- Everything declared within a `package` block refer to symbols in that package's table
- The scope of a package continues until the next `package` or until the end of the file

Packages

- Encapsulation of named entities in a single namespace
- `package` declares a package and a namespace
- Everything declared within a `package` block refer to symbols in that package's table
- The scope of a package continues until the next `package` or until the end of the file
- The default package is `main`

Packages

- Encapsulation of named entities in a single namespace
- `package` declares a package and a namespace
- Everything declared within a `package` block refer to symbols in that package's table
- The scope of a package continues until the next `package` or until the end of the file
- The default package is `main`
- A package has a version and three implicit methods: `import`, `unimport`, `VERSION`

Packages

- Encapsulation of named entities in a single namespace
- `package` declares a package and a namespace
- Everything declared within a `package` block refer to symbols in that package's table
- The scope of a package continues until the next `package` or until the end of the file
- The default package is `main`
- A package has a version and three implicit methods: `import`, `unimport`, `VERSION`
- Perl has *open namespaces*, you can add definitions at anytime

Packages

package Example

```
# new way to version packages
package Pinball:Wizard v123.45.6 { ... }

# old way
package Pinball::Wizard { our $VERSION = 123.45.6; ... }
```

References

- I does what you expect for references

References

- I does what you expect for references
- Use the reference operator (\) on a variable to extract its reference

References

- I does what you expect for references
- Use the reference operator (\) on a variable to extract its reference
- References are scalar values

References

- I does what you expect for references
- Use the reference operator (\) on a variable to extract its reference
- References are scalar values
- To dereference a reference use the corresponding sigil for the referenced variable

References

- I does what you expect for references
- Use the reference operator (\) on a variable to extract its reference
- References are scalar values
- To dereference a reference use the corresponding sigil for the referenced variable
- Another way to dereference use the arrow operator;

Scalar References

Scalar Reference

```
my $name = 'Larry';
my $name_ref = \$name;

# to modify the value from the reference
$$name_ref = 'Moe';
```

ArrayReferences

To create a new unamed array reference use [ ]

```
my @names = qw(Larry Moe Curly);
my $names_ref = \@names;

# Access one element
$$names_ref[0]  = 'Moe';
$names_ref->[1] = 'Curly';

# Access the entire array
my $name_count = @$names_ref;

# or slice
my @last_two = @{ $names_ref }[-1, -2];

# create an un-named refernece
my $pets_ref = [qw/cat dog bird/];
```

Hash References

To create a new unamed hash reference use { }

```
my %spanish_color_for = (
    blue   => 'azul',
    yellow => 'amarillo',
);
# Extract reference, keys and vlues
my $spanish_color_for_ref = \%spanish_color_for;
my @spanish_colors = values %{ $spanish_color_for_ref };
my @english_colors = keys   %{ $spanish_color_for_ref };

# Access a single element
my $cool_color = $spanish_color_for_ref->{'blue'}
my $same_cool_color = ${ $spanish_color_for_ref }{'yellow'}

# slice
my @colores = @{ $spanish_color_for_ref }{ qw/blue yellow/ }

# create an unnamed reference
my $spanish_colors_ref = {
    blue   => 'azul',
    yellow => 'amarillo',
};
```

Function References

- Functions in perl are data types
- To create an unamed function use sub without a name
- To extract the reference of an existing named function use the \ followed by the function sigil &

Function references

```
# Extract the reference
sub bake_cake { say 'Baking a wonderful cake!' };
my $cake_sub_ref = \&bake_cake;

# Call the function from the reference
$cake_sub_ref->();

# or ...But this is old don't use it
&$cake_sub_ref;

# or ...create an unnamed function
my $cake_sub_ref_2 = sub { say 'Baking a wonderful cake2!' };
```

Filehandle References

- The lexical filehandle form of `open` and `opendir` operate on filehandles references
- The references are object of `IO::File`

Filehandle references

```
use autodie 'open';
open my $out_fh, '>', 'output_file.txt';

# write to the file handle
$out_fh->say( 'Have some text!' );

# ...or
say $out_fh 'Have some text!';
```

References and Memory Collection

- Perl's memory management technique is reference count.

References and Memory Collection

- Perl's memory management technique is reference count.
  1. Keeps track of the number of places where a reference is being used

References and Memory Collection

- Perl's memory management technique is reference count.
  1. Keeps track of the number of places where a reference is being used
  2. When the count drops to 0, perl knows that it's safe to claim the memory.

References

- Nested data structures `perldoc perldsc`

Reference Example (Autovivification)

```
my %band_members_in = (
    'The Beatles' => {
        'John Lennon'     => [ qw/guitar voice keyboards/ ],
        'Paul McCartney'  => [ qw/bass voice guitar drums piano/ ],
        'George Harrison' => [ qw/guitar voice bass/ ],
        'Ringo Starr'     => [ qw/drums voice tambourine/ ],
    },
    'Minutemen' => {
        'D. Boon'         => [ qw/guitar voice/ ],
        'Mike Watt'       => [ qw/bass voice/ ],
        'George Hurley'   => [ qw/drums/ ],
    },
    'Cafe Tacuba' => {
        'Ruben Albarran'    => [ qw/voice/ ],
        'Emmanuel del Real' => [ qw/keyboards voice/ ],
        'Joselo Rangel'     => [ qw/guitar voice/ ],
        'Enrique Rangel'    => [ qw/bass/]
    }
)
```

Reference Example Cont...

```
sub john_lennon_played {
    say join( q/, /, @{ $band_members_in{'The Beatles'}->{'John Lennon'} } );
}

sub band_member_played {
   my %params = @_;
   my ($band, $member) = @params{ qw/band member/ };

   if ( $band && member
         && exists $band_members_in{$band}->{$member} {

       local $" = q/, /;
       say "@{ $band_members_in{'$band'}{'$member} }"

        # ...what happened to the -> between {$band} and {$member}?
   }
}

band_member_played(member => 'Mike Watt', band => 'Minutemen');
# to debug nested data structures you can *use Data::Dumper*
```

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`
  2. Associativity, whether left to right or right to left:

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`
  2. Associativity, whether left to right or right to left:
     - `2 ** 3 ** 4`

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`
  2. Associativity, whether left to right or right to left:
     - `2 ** 3 ** 4`
  3. Arity, number of operands on which the operator operates: unary, binary, trinary

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`
  2. Associativity, whether left to right or right to left:
     - `2 ** 3 ** 4`
  3. Arity, number of operands on which the operator operates: unary, binary, trinary
  4. fixity, the position relative to its operands

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`
  2. Associativity, whether left to right or right to left:
     - `2 ** 3 ** 4`
  3. Arity, number of operands on which the operator operates: unary, binary, trinary
  4. fixity, the position relative to its operands
     1. Infix: `$length * $width`

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`
  2. Associativity, whether left to right or right to left:
     - `2 ** 3 ** 4`
  3. Arity, number of operands on which the operator operates: unary, binary, trinary
  4. fixity, the position relative to its operands
     1. Infix: `$length * $width`
     2. Prefix and postfix: `++$x and $x++`

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`
  2. Associativity, whether left to right or right to left:
     - `2 ** 3 ** 4`
  3. Arity, number of operands on which the operator operates: unary, binary, trinary
  4. fixity, the position relative to its operands
     1. Infix: `$length * $width`
     2. Prefix and postfix: `++$x and $x++`
     3. Circumfix: `qw[one two three four]`

Operators "Perl an Operator-oriented language"

- Reference about operators `perldoc perlop` and `perldoc perlsyn`
- Definitions about operators:
  1. Precedence: `(1 + 5) * 6p`
  2. Associativity, whether left to right or right to left:
     - `2 ** 3 ** 4`
  3. Arity, number of operands on which the operator operates: unary, binary, trinary
  4. fixity, the position relative to its operands
     1. Infix: `$length * $width`
     2. Prefix and postfix: `++$x and $x++`
     3. Circumfix: `qw[one two three four]`
     4. Postcircumfix: `$hash{$x}` where `{}` come after `$hash` and surround `$x`

Operators

- Operator types

Operators

- Operator types
  1. Numeric Operators: +, -, *, /, **, %, +=, −, etc

Operators

- Operator types
  1. Numeric Operators: +, -, *, /, **, %, +=, –, etc
  2. String Operators: =~, !~, eq, ne, gt, lt, ge, le, cmp

Operators

- Operator types
  1. Numeric Operators: +, -, *, /, **, %, +=, –, etc
  2. String Operators: =~, !~, eq, ne, gt, lt, ge, le, cmp
  3. Logical Operators: &&, and, ||, or, //, etc

Operators

- Operator types
  1. Numeric Operators: +, -, *, /, **, %, +=, –, etc
  2. String Operators: =~, !~, eq, ne, gt, lt, ge, le, cmp
  3. Logical Operators: &&, and, ||, or, //, etc
  4. Bitwise Operators: », «, &, |, ^

Operators

- Operator types
  1. Numeric Operators: +, -, *, /, **, %, +=, –, etc
  2. String Operators: =~, !~, eq, ne, gt, lt, ge, le, cmp
  3. Logical Operators: &&, and, ||, or, //, etc
  4. Bitwise Operators: », «, &, |, ^
  5. Repetition operator: x

Operators

- Operator types
  1. Numeric Operators: +, -, *, /, **, %, +=, –, etc
  2. String Operators: =~, !~, eq, ne, gt, lt, ge, le, cmp
  3. Logical Operators: &&, and, ||, or, //, etc
  4. Bitwise Operators: », «, &, |, ^
  5. Repetition operator: x
  6. Range operator: 1 .. 10, but in boolean context it's the flip flop operator

Functions

- Declaration: Use the `sub` builtin followed by a name and a code block

Functions

- Declaration: Use the `sub` builtin followed by a name and a code block
- When invoking a function use postcircumfix parentheses (optional most of the times)

Functions

- Declaration: Use the `sub` builtin followed by a name and a code block
- When invoking a function use postcircumfix parentheses (optional most of the times)
- Arguments can be arbitrary expressions

Function Parameters

- A function receives its parameters in a single array `@_`

Function Parameters

- A function receives its parameters in a single array `@_`
- You need to unpack the arguments in `@_` for one parameter use `shift`

Function Parameters

- A function receives its parameters in a single array `@_`
- You need to unpack the arguments in `@_` for one parameter use `shift`
- Starting on v5.20 signatures are now supported as "Experimental

Function Parameters

- A function receives its parameters in a single array `@_`
- You need to unpack the arguments in `@_` for one parameter use `shift`
- Starting on v5.20 signatures are now supported as "Experimental
- If you operate directly on the contents of `@_` you're operating directly the calling values

Function Parameters

Real signatures

```
use experimental 'signatures';

sub greet($name = 'Juan') {
    say "Hello, $name";
}
```

Function Parameters

- Every function has a containing namespace
- A function can be contained in another namespace anywhere in the code
- Lexical subs are available starting on v5.18 `perldoc perlsub`

Code

```
sub Some::Package::my_function { ... }
```

Importing from other packages

- When loading a module with `use` perl calls `import()` with any arguments passed to it

What happens when using 'use'

```
use strict 'refs';
use strict qw/subs vars/

# is equivalent to
BEGIN {
    require strict;
    strict->import('refs');
    strict->import( qw/subs vars/ );
}
```

call stack

- `caller` inspect the calling context:

call stack

- `caller` inspect the calling context:
  - `my ($package, $file, $line) = caller()`

call stack

- `caller` inspect the calling context:
  - `my ($package, $file, $line) = caller()`
- `caller(n)` where n is the stack frame if n == 0, then stack from top

call stack

- `caller` inspect the calling context:
  - `my ($package, $file, $line) = caller()`
- `caller(n)` where n is the stack frame if n == 0, then stack from top
- `Carp::croack` and `Carp::carp` to report from the caller's point of view

Closure Example

```
sub gen_fib {
        my @fibs = (0, 1);

        return sub {
            my $item = shift;

            if ($item >= @fibs) {
                for my $calc (@fibs .. $item) {
                    $fibs[$calc] = $fibs[$calc - 2]
                                 + $fibs[$calc - 1];
                }
            }
            return $fibs[$item];
        }
    }

    # calculate 42nd Fibonacci number
    my $fib = gen_fib();
    say $fib->( 42 );
```

AUTOLOAD

- This is the default function to call in a package when calling to an non-existing function

AUTOLOAD

- This is the default function to call in a package when calling to an non-existing function
- The arguments passed to the non-existing functions are passed to AUTOLOAD via @_

AUTOLOAD

- This is the default function to call in a package when calling to an non-existing function
- The arguments passed to the non-existing functions are passed to AUTOLOAD via @_
- The package global $AUTOLOAD will contain the name of the non-existing function

AUTOLOAD

- This is the default function to call in a package when calling to an non-existing function
- The arguments passed to the non-existing functions are passed to AUTOLOAD via @_
- The package global $AUTOLOAD will contain the name of the non-existing function
- The caller to the non-existing sub will get whatever AUTOLOAD returns

AUTOLOAD Example

```
sub AUTOLOAD {
    our $AUTOLOAD;
    say "Hello from AUTOLOAD: user tried to run $AUTOLOAD"

    # if want to register the non-existent name into the current package
    my $method = sub { ... };

    no strict 'refs';
    *{ $AUTLOAD } = $method;
    return $method->(@_);       # or return goto &$method;
}
non_existing( one => 'argument' );
```

Topic

Regexp basic operators

- `m//` or the shorter `//` identifies a regular expression
- `=~` is the binding operator, when evaluated in scalar context a match evaluates to boolean value representing the success or failure of the match
- `!~` is the negated version of the binding operator
- `s////`

Regexp Examples

```
my $mood = "Because I'm happy";
say 'I found a happy in string' if $mood =~ /happy/;

my $mood =~ s/happy/sad/;
say $mood;
```

qr// Operator

- Creates first-class regexes that can be stored in variables
- can be used to create complex regex patterns

qw// Example

```
my $happy = qr/happy/i;
my $sad   = qr/sad/i;

say "You're being emotional" if $mood =~ m{ $happy|$sad }
```

Quantifiers

- ? Matches zero or more preceeding expressions

Quantifiers

- ? Matches zero or more preceeding expressions
- + Matches one or more

Quantifiers

- ? Matches zero or more preceeding expressions
- + Matches one or more
- * Matches zero or more

Quantifiers

- `?` Matches zero or more preceeding expressions
- `+` Matches one or more
- `*` Matches zero or more
- `{m[,[n]]}` Matches m but not more than n

Metacharacters

- `.` Match `any` character except a newline

Metacharacters

- `.` Match `any` character except a newline
- `\w` represents all Unicode alphanumeric characters. Negated as: `\W`

Metacharacters

- `.` Match `any` character except a newline
- `\w` represents all Unicode alphanumeric characters. Negated as: `\W`
- `\d` Matches a numeric character. Negated as `\D`

Metacharacters

- `.` Match `any` character except a newline
- `\w` represents all Unicode alphanumeric characters. Negated as: `\W`
- `\d` Matches a numeric character. Negated as `\D`
- `\s` Matches whitespace characters (tab, space, CR, LF, newline). Negated as `\S`

Metacharacters

- `.` Match `any` character except a newline
- `\w` represents all Unicode alphanumeric characters. Negated as: `\W`
- `\d` Matches a numeric character. Negated as `\D`
- `\s` Matches whitespace characters (tab, space, CR, LF, newline). Negated as `\S`
- `\Q\E` Disable metacharacter interpretation `m/\Q$literal_text\E/`

Character Classes

- `[]` Group alternatives as `[aeiou]` matches any of the vowels or `[A-Za-z0-9]` to match ranges

Character Classes

- `[]` Group alternatives as `[aeiou]` matches any of the vowels or `[A-Za-z0-9]` to match ranges
- `[^]` To negate the atoms within the class like `[^aeiou]`

Anchors

- Fixes the position of a regex

Anchors

- Fixes the position of a regex
- `\A` start of string

Anchors

- Fixes the position of a regex
- \A start of string
- \Z Match at the end of the string or before newline at the end of the string

Anchors

- Fixes the position of a regex
- \A start of string
- \Z Match at the end of the string or before newline at the end of the string
- \z end of string

Anchors

- Fixes the position of a regex
- \A start of string
- \Z Match at the end of the string or before newline at the end of the string
- \z end of string
- ^ start of a line

Anchors

- Fixes the position of a regex
- \A start of string
- \Z Match at the end of the string or before newline at the end of the string
- \z end of string
- ^ start of a line
- $ end of a line

Anchors

- Fixes the position of a regex
- \A start of string
- \Z Match at the end of the string or before newline at the end of the string
- \z end of string
- ^ start of a line
- $ end of a line
- \b boundary between a word character \w and a non-word character \W

Assertions

- Assertions are zero-width and don't consume characters from the match
- (?=) Positive look-ahead assertion
- (?!) Negative look-ahead assertion
- (?<=) Positive look-behind assertion
- \K Variable positive look-behind assertion

Examples

```
# (?=)
$disastrous_feline = qr/cat(?=astrophe)/
# (?!)
$safe_feline = qr/cat(?!astrophe)/
# (?<=)
$space_cat = qr/(?<=\s)cat/
# \K
s/foo\Kbar//g   #same as ... s/(foo)bar/$1/g
```

Named Captures

- Capture matches for later use using `(?<name>$regex)`
- This will create a new entry in the %+ hash with the key `name` and the matched text as the value
- To remove capturing from parentheses use `(?:)`

Named Captures Example

```
my $contact_info = '(202) 456-1111';

# build regex
my $area_code = qr/\(\d{3}\)/;
my $local_number = qr/\d{3}-?\d{4}/;
my $phone_number = qr/$area_code\s?$local_number/;

# match and capture
if ($contact_info =~ /?<phone>$phone_number/) {
    say "You can call this guy at $+{ phone }"
}
# Also used in substitutions
my $mood = "I'm feeling happy";
$mood =~ s/feeling\s*(?<mood>\w+)/so, you're $+{mood}/;
```

Numbered Captures

- On unamed captures, captures with parentheses with store the matches in variables $1, $2, ...

Numbered Captures Example

```
if ( $contact_info =~ /($phone_number)/ ) {
    say "You can call this guy at $1";
}

# also used in substitutions
my $mood = "I'm feeling happy";
$mood =~ s/feeling (\w+)/not feeling $1/;
```

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`
- Or they can appear within the expression

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`
- Or they can appear within the expression
- `i` : Match ignoring case

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`
- Or they can appear within the expression
- `i` : Match ignoring case
- `m` : Allows the `^` and `$` anchors to match at any newline embedded within the string

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`
- Or they can appear within the expression
- `i` : Match ignoring case
- `m` : Allows the `^` and `$` anchors to match at any newline embedded within the string
- `s` : Treats the source string as a single line so `.` wll match the newline character

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`
- Or they can appear within the expression
- `i` : Match ignoring case
- `m` : Allows the `^` and `$` anchors to match at any newline embedded within the string
- `s` : Treats the source string as a single line so `.` wll match the newline character
- `r` : Substitution operation returns the result of the substitution without modifying the source

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`
- Or they can appear within the expression
- `i` : Match ignoring case
- `m` : Allows the `^` and `$` anchors to match at any newline embedded within the string
- `s` : Treats the source string as a single line so `.` wll match the newline character
- `r` : Substitution operation returns the result of the substitution without modifying the source
- `x` : Allows the regexp to have embedded additional whitespace and comments

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`
- Or they can appear within the expression
- `i` : Match ignoring case
- `m` : Allows the `^` and `$` anchors to match at any newline embedded within the string
- `s` : Treats the source string as a single line so `.` wll match the newline character
- `r` : Substitution operation returns the result of the substitution without modifying the source
- `x` : Allows the regexp to have embedded additional whitespace and comments
- `g` : matches a regex globally throughout a string

Modifiers

- Configure the behavior of the regular expression, they can be appear at the end of `m//` or `qr//`
- Or they can appear within the expression
- `i` : Match ignoring case
- `m` : Allows the `^` and `$` anchors to match at any newline embedded within the string
- `s` : Treats the source string as a single line so `.` wll match the newline character
- `r` : Substitution operation returns the result of the substitution without modifying the source
- `x` : Allows the regexp to have embedded additional whitespace and comments
- `g` : matches a regex globally throughout a string
- `e` : allows to write arbitrary code on the right side of a substitution operation

Modifiers

```
my $re = qr/text/i
$re = qr/(?i)text/

# disable the modifiers by prepending a -
$re = qr/(?-i)text/

# using named captures
$re = /(?<name>(?i)text/

# multi-line regex
my $attr_re = qr{
    \A                      # start of line

    (?:
      [;\n\s]*              # spaces and semicolons
      (?:/\*.*?\*/)?        # C comments
    )*
    ATTR
    \s+
    (   U?INTVAL
      | FLOATVAL
      | STRING\s+\*
    )
}x;
```

Topic

Vanilla OO

- Perl's core object system is deliberately minimal

Vanilla OO

- Perl's core object system is deliberately minimal
- it only has three rules

Vanilla OO

- Perl's core object system is deliberately minimal
- it only has three rules
  1. A class is a package

Vanilla OO

- Perl's core object system is deliberately minimal
- it only has three rules
  1. A class is a package
  2. A method is a function

Vanilla OO

- Perl's core object system is deliberately minimal
- it only has three rules
  1. A class is a package
  2. A method is a function
  3. A (blessed) reference is an object

bless

- Associates a reference to a class name

bless

- Associates a reference to a class name
- A blessed reference now is a valid invocant and Perl will perform method dispatch

bless

- Associates a reference to a class name
- A blessed reference now is a valid invocant and Perl will perform method dispatch
- A constructor is the method that creates a blessed reference

Basic example

```
package Fish;
use Carp;

sub new {
  my ($class, %fish_attrs) = @_;

  croak "This fish needs a name"
    unless exists $fish_attrs{name};
  croak "This fish needs a diet"
    unless exists $fish_attrs{name};
  $fish_attrs{birth_year} = (localtime)[5] + 1900
    unless exists $fish_attrs{birth_year};
  bless \%fish_attrs, $class;
}

sub diet {
  my ($self, $diet) = @_;
  return $self->{diet} unless $diet;
  $self->{diet} = $diet;
}
sub name { return shift->{name} }
sub age { return (localtime)[5] + 1900 - shift->{birth_year} }

1;
```

Classes

- They're just packages : `package`

Methods

- They're just functions `sub`
- If want to override a parent method just declare the method in the child class using the same name, and call `SUPER::` to dispatch the parent

Override Example

```
sub overriden {
  my $self = shift;
  return $self->SUPER::overriden(@_);
}
```

Inheritance

- Perl uses a package global variable `@ISA` to keep track of inheritance
- The method dispatcher looks in each class's `@ISA` to find the names of its parents

Inheritance Example

```
package InjuredPlayer {
  @InjuredPlayer::ISA = qw/Player Hospital::Patient/;
}

# Better yet use the parent pragma
package InjuredPlayer {
  use parent qw/Player Hospital::Patient/;
}
```

Topic

Classes

- Define classes by naming them with package and use Moose within the package

Classes

- Define classes by naming them with package and use Moose within the package
- Define a property with has `perldoc Moose::Manual::Attributes`

Classes

- Define classes by naming them with package and use Moose within the package
- Define a property with has `perldoc Moose::Manual::Attributes`
- Define a method with sub `perldoc Moose::Manual`

Moose Class Example

```
package Car {
    use Moose;
    # Properties
    has painted_with => (       # paint goodies
        is      => 'ro',
        isa     => 'ArrayRef',
        default => sub { [qw/blue smurfs/] },
        lazy    => 1,
    );
    # Methods
    sub run {
      my $self = shift;
      $self->turn_on_engine();
      $self->fuel_engine()
    }
}
# Car class user
 use Car;
 my $flaming_car = Car->new( painted_with => [ qw/flames devils/ ]  );

 local $" = ' and ';
 say "This car was painted with @{ $flaming_car->painted_with }";
```

Roles

- Collection of behaviors and state

Roles

- Collection of behaviors and state
- Can't instantiate a role

Roles

- Collection of behaviors and state
- Can't instantiate a role
- Declared with `Moose::Role`

Roles

- Collection of behaviors and state
- Can't instantiate a role
- Declared with `Moose::Role`
- `requires` lists the required methods for its composing clases

Roles

- Collection of behaviors and state
- Can't instantiate a role
- Declared with `Moose::Role`
- `requires` lists the required methods for its composing clases
- `with` composes the Role into a class

Roles

- Collection of behaviors and state
- Can't instantiate a role
- Declared with `Moose::Role`
- `requires` lists the required methods for its composing clases
- `with` composes the Role into a class
- `DOES` will tell if the object "does" a role

Roles Example

```
package LivingBeing {
    use Moose::Role;
    requires qw/ name age diet /;
}
package CalculateAge::From::BirthYear {
    use Moose::Role;
    has 'birth_year',
        is => 'ro',
        isa => 'Int',
        default => sub { (localtime)[5] + 1900 };
    sub age {
        my $self = shift;
        my $year = (localtime)[5] + 1900

       return $year - $self->birth_year;
    }
}
package Cat {
    use Moose;
    has 'hame' => ( is => 'ro', isa => 'Str' );
    has 'diet' => ( is => 'rw', isa => 'Str' );
    with 'LivingBeing', 'CalculateAge::From::BirthYear';
}
my $kitty = Cat->new( diet => 'fish', birth_year => 2010, name => 'dude');
say $kitty->name, ' is alive!! ' if $kitty->DOES('LivingBeing');
say $kitty->name, ' is ', $kitty->age, ' years old.';
```

Same Example with sugar MooseX::Declare or

```
use MooseX::Declare;
role LivingBeing {
  requires qw/ name age diet /;
};
role CalculateAge::From::BirthYear {
  has 'birth_year' => (
    is      => 'ro',
    isa     => 'Int',
    default => sub { (localtime)[5] + 1900 }
   );
  method age {
    my $year = (localtime)[5] + 1900;
    return $year - $self->birth_year;
  }
};
class Cat with LivingBeing with CalculateAge::From::BirthYear {
    has 'hame' => ( is => 'ro', isa => 'Str' );
    has 'diet' => ( is => 'rw', isa => 'Str' );
};
my $kitty = Cat->new( diet => 'fish', birth_year => 2010, name => 'dude');
say "$kitty->name is alive!! " if $kitty->DOES('LivingBeing');
say "$kitty->name is $kitty->age years old.";
```

Inheritance

- Use a inheritance when one class truly extends another

Inheritance

- Use a inheritance when one class truly extends another
- Use a role when a class needs additional behavior, espacially when that behavior has a meaningful name

Inheritance

- Use a inheritance when one class truly extends another
- Use a role when a class needs additional behavior, espacially when that behavior has a meaningful name
- Inherit from an existing class by using `extends`, it takes a list of parent classes

Inheritance

- Use a inheritance when one class truly extends another
- Use a role when a class needs additional behavior, espacially when that behavior has a meaningful name
- Inherit from an existing class by using `extends`, it takes a list of parent classes
- use + to indicate that an attribute is modifying the attribute

Inheritance

- Use a inheritance when one class truly extends another
- Use a role when a class needs additional behavior, espacially when that behavior has a meaningful name
- Inherit from an existing class by using `extends`, it takes a list of parent classes
- use + to indicate that an attribute is modifying the attribute
- To override a method use `override`

Inheritance

- Use a inheritance when one class truly extends another
- Use a role when a class needs additional behavior, espacially when that behavior has a meaningful name
- Inherit from an existing class by using `extends`, it takes a list of parent classes
- use + to indicate that an attribute is modifying the attribute
- To override a method use `override`
- `isa` will tell if the invocant extends a named class

Inheritance Example

```
package LightSource {
  use Moose;
  has 'candle_power' => (
    is      =>'ro',
    isa     =>'Int',
    default => 1
   );

  has 'enabled' => (
    is      => 'ro',
    isa     => 'Bool',
    default => 0,
    writer  => '_set_enabled'
   );

  sub light {
    my $self = shift; $self->_set_enabled(1);
  }

  sub extinguish {
    my $self = shift; $self->_set_enabled(0);
  }
};
```

Inheritance Example

```
package SuperCandle {
  use Moose;
  extends 'LightSource';

  has '+candle_power' => ( default => 100 );
}
```

Inheritance Exmple

```
package LigthSource::Cranky {
  use Carp 'carp';
  use Moose;
  extends 'LigthSource';

  overide light => sub {
    my $self = shift;

    carp "Can't light a lit LightSource!"
      if $self->enabled;

    super();
  };

  override extinguish => sub {
    my $self = shift;

    carp "Can't extinguish unlit LightSource!"
      unless $self->enabled;

    super();
   };
}
```

Metaprogramming and Reflection

- Inspect a class via `meta`

Metaprogramming and Reflection

- Inspect a class via `meta`
- Once a module has been loaded, it's registered in `%INC`

Metaprogramming and Reflection

- Inspect a class via `meta`
- Once a module has been loaded, it's registered in `%INC`
- `Class::Load` does the task of properly checking this

Metaprogramming and Reflection

- Inspect a class via `meta`
- Once a module has been loaded, it's registered in `%INC`
- `Class::Load` does the task of properly checking this
- To check if a package exists ask `UNIVERSAL`
  `$pkg->can('can')`

Metaprogramming and Reflection

- Inspect a class via `meta`
- Once a module has been loaded, it's registered in `%INC`
- `Class::Load` does the task of properly checking this
- To check if a package exists ask `UNIVERSAL`
  `$pkg->can('can')`
- To check the version of a module `$module->VERSION()`

Metaprogramming and Reflection

- Inspect a class via `meta`
- Once a module has been loaded, it's registered in `%INC`
- `Class::Load` does the task of properly checking this
- To check if a package exists ask `UNIVERSAL` `$pkg->can('can')`
- To check the version of a module `$module->VERSION()`
- To check if a function exists in a package `$pkg->can( $func )`

meta Example

```
my $meta = LightSource->meta;

say 'LightSource instances have the attributes:';
say $_->name for $meta->get_all_attributes;

say 'LightSource instances support the methods:';
say $_->fully_qualified_name for $meta->get_all_methods;
```

Moose::Manual

Manual

- Look at the manual for tons of interesting features
- `https://metacpan.org/pod/Moose::Manual`

UNIVERSAL

- The `UNIVERSAL` package is the ancestor of all other packages, the ultimate parent

UNIVERSAL

- The `UNIVERSAL` package is the ancestor of all other packages, the ultimate parent
- Provides the following methods:

UNIVERSAL

- The `UNIVERSAL` package is the ancestor of all other packages, the ultimate parent
- Provides the following methods:
  1. VERSION() - Returns the value of the $VERSION package global

UNIVERSAL

- The `UNIVERSAL` package is the ancestor of all other packages, the ultimate parent
- Provides the following methods:
  1. VERSION() - Returns the value of the $VERSION package global
  2. DOES() - Supports the use of Roles in programs

UNIVERSAL

- The `UNIVERSAL` package is the ancestor of all other packages, the ultimate parent
- Provides the following methods:
  1. VERSION() - Returns the value of the $VERSION package global
  2. DOES() - Supports the use of Roles in programs
  3. can() - Returns the function reference if it's supported

UNIVERSAL

- The `UNIVERSAL` package is the ancestor of all other packages, the ultimate parent
- Provides the following methods:
  1. VERSION() - Returns the value of the $VERSION package global
  2. DOES() - Supports the use of Roles in programs
  3. can() - Returns the function reference if it's supported
  4. isa() - Returns true if its invocant derives from the named class

Topic

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths
- Another option for this task `Path::Class`

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths
- Another option for this task `Path::Class`
- `-X` test operators

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths
- Another option for this task `Path::Class`
- `-X` test operators
  - `-e` : File exists

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths
- Another option for this task `Path::Class`
- `-X` test operators
  - `-e` : File exists
  - `-f` : File is a plain file

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths
- Another option for this task `Path::Class`
- `-X` test operators
  - `-e` : File exists
  - `-f` : File is a plain file
  - `-d` : File is a directory

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths
- Another option for this task `Path::Class`
- `-X` test operators
  - `-e` : File exists
  - `-f` : File is a plain file
  - `-d` : File is a directory
  - `-r` : File allows read

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths
- Another option for this task `Path::Class`
- `-X` test operators
  - `-e` : File exists
  - `-f` : File is a plain file
  - `-d` : File is a directory
  - `-r` : File allows read
  - `-s` : File is Non-empty

Files

- Perl offers a Unix style view of the filesystem. Use `File::Spec` to protably manipulate file paths
- Another option for this task `Path::Class`
- `-X` test operators
  - `-e` : File exists
  - `-f` : File is a plain file
  - `-d` : File is a directory
  - `-r` : File allows read
  - `-s` : File is Non-empty
  - `perldoc -f -X` for more tests

Idioms: Schwartzian Transform

Idiom borrowed from Lisp

```
# Associate the names of workers and phone extensions
# PROBLEM: write a phone book, sorted by name;

my %extensions = (
  '000' => 'Freddie',
  '002' => 'Brian',
  '042' => 'John',
  '044' => 'Roger',
 );
# sort list by name alphabetically, need to sort by values?
my @sorted_names = sort values %extensions;

# map/transform data to preserve key/value information
my @pairs = map { [ $_, $extensions{$_} ] } keys %extensions;

# sort data using new representation
my @sorted_pairs = sort { $a->[1] cmp $b->[1] } @pairs;

# format the sorted data
my @formatted_exts = map { "$_->[1], ext. $_->[0]" } @sorted_pairs;

# print data
say for @formatted_exts;
```

Idioms: Schwartzian Transform

Schwartzian Transform cont...

```
# Schwartzian Transform chain all the previous steps
say for
  map  { "$_->[1], ext. $_->[0]" }
  sort { $a->[1] cmp $b->[1] }
  map  { [ $_ => $extensions{ $_ } ] }
  keys %extensions;
```

Idioms: File Slurping

File Slurp

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

# ...or
my $file; { local $/; $file = <$fh> };

# .. or
use File::Slurper;
my $content = read_text($filename);)
```

Throw/catch Exceptions

- To throw an exception, use `die` or `croak`
- To catch an exception, evaluate the code that can throw the exception withing an eval block, inspect the exception using `$@`

Throw/catch Example

```
local $@;

# catch the exception
my $fh = eval { open_log_file('some_file.log'} };

# analyze the exception
if (my $exception = $@) {
 # re-throw the exception if we can't handle it here
 die $exception unless $exception =~ /^Can't open logging/;
 $fh = log_to_syslog();
}
```

Pragmas

- Perl modules that influence the behavior of the language

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use
  2. `warnings` Always use

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use
  2. `warnings` Always use
  3. `utf8`

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use
  2. `warnings` Always use
  3. `utf8`
  4. `autodie`

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use
  2. `warnings` Always use
  3. `utf8`
  4. `autodie`
  5. `constant`

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use
  2. `warnings` Always use
  3. `utf8`
  4. `autodie`
  5. `constant`
  6. `vars`

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use
  2. `warnings` Always use
  3. `utf8`
  4. `autodie`
  5. `constant`
  6. `vars`
  7. `feature` use 5.18 or *use feature ':5.18'*

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use
  2. `warnings` Always use
  3. `utf8`
  4. `autodie`
  5. `constant`
  6. `vars`
  7. `feature` use 5.18 or *use feature ':5.18'*
  8. `experimental`

Pragmas

- Perl modules that influence the behavior of the language
- By convension, pragma names are written in lower case
- Using a pragma makes its behavior effective within a lexical scope
- To disable pragmas, it can be done in a lexical scpoe as well with no
- Some useful pragmas:
  1. `strict` Always use
  2. `warnings` Always use
  3. `utf8`
  4. `autodie`
  5. `constant`
  6. `vars`
  7. `feature` use 5.18 or *use feature ':5.18'*
  8. `experimental`
  9. `less` - Write your own lexical pragmas `perldoc perlpragma`

Taint Mode

- Sticky piece of metadata attached to all data which comes from the outside

Taint Mode

- Sticky piece of metadata attached to all data which comes from the outside
- Any data derived from tainted data is also tainted

`perldoc perlsec`

Taint Mode

- Sticky piece of metadata attached to all data which comes from the outside
- Any data derived from tainted data is also tainted

`perldoc perlsec`

- Launch your program with the `-T` command line argument to enable tainted mode

Taint Mode

- Sticky piece of metadata attached to all data which comes from the outside
- Any data derived from tainted data is also tainted
  `perldoc perlsec`
- Launch your program with the `-T` command line argument to enable tainted mode
- `Scalar::Util::tainted()` returns true if its argument is tainted

Taint Mode

- Sticky piece of metadata attached to all data which comes from the outside
- Any data derived from tainted data is also tainted `perldoc perlsec`
- Launch your program with the `-T` command line argument to enable tainted mode
- `Scalar::Util::tainted()` returns true if its argument is tainted
- To remove tain from data, extract known-good portions of the data with a regular expression capture

Taint Mode

- Sticky piece of metadata attached to all data which comes from the outside
- Any data derived from tainted data is also tainted `perldoc perlsec`
- Launch your program with the `-T` command line argument to enable tainted mode
- `Scalar::Util::tainted()` returns true if its argument is tainted
- To remove tain from data, extract known-good portions of the data with a regular expression capture
- `-t` flag enables taiin mode but reduces tain violations from excemptions to warnings

Taint mode example

```
# start code as $ perl -T tainted.pl

# tainted.pl
# ==========
my $number = <>;

die 'Number still tainted!'
  unless $number =~ /(\(/d{3}\) \d{3}-\d{4})/;

my $safe_number = $1;
```

One liners

http://www.catonmat.net/download/perl1line.txt

Topic

Test::More

- `ok()` The most basic assertion test function, takes two values

Test::More

- `ok()` The most basic assertion test function, takes two values
- `is()` Compares two values using Perl's eq operator. Aplies scalar context/

Test::More

- `ok()` The most basic assertion test function, takes two values
- `is()` Compares two values using Perl's eq operator. Aplies scalar context/
- `isnt()` Compares two values using ne

Test::More

- `ok()` The most basic assertion test function, takes two values
- `is()` Compares two values using Perl's eq operator. Aplies scalar context/
- `isnt()` Compares two values using ne
- `cmp_ok()` You can provide the comparision operator

Test::More

- `ok()` The most basic assertion test function, takes two values
- `is()` Compares two values using Perl's eq operator. Aplies scalar context/
- `isnt()` Compares two values using ne
- `cmp_ok()` You can provide the comparision operator
- `isa_ok()` Tests if a reference is of any type

Test::More

- `ok()` The most basic assertion test function, takes two values
- `is()` Compares two values using Perl's eq operator. Aplies scalar context/
- `isnt()` Compares two values using ne
- `cmp_ok()` You can provide the comparision operator
- `isa_ok()` Tests if a reference is of any type
- `can_ok()` Tests if an object provides functionality

Test::More

- `ok()` The most basic assertion test function, takes two values
- `is()` Compares two values using Perl's eq operator. Aplies scalar context/
- `isnt()` Compares two values using ne
- `cmp_ok()` You can provide the comparision operator
- `isa_ok()` Tests if a reference is of any type
- `can_ok()` Tests if an object provides functionality
- `is_deeply()` Compares two references to ensure their contents are equal

Test::More Example

```
Use Test::More tests => 1;  # Test plan
ok  1, 'the number one is a true value';

done_testing();
```

Test Anything Protocol

- The output from the tests are formatted in *Test Anything Protocol (TAP)*

Test Anything Protocol

- The output from the tests are formatted in *Test Anything Protocol (TAP)*
- `http://testanything.org`

Running Tests

- The program `prove` runs tests, interprets TAP and display relevant information

Running Tests

- The program `prove` runs tests, interprets TAP and display relevant information
- See `perldoc prove` for more options

Organizing tests

- CPAN distributions should include a `t/` directory containing test files `.t`

Organizing tests

- CPAN distributions should include a `t/` directory containing test files `.t`
- When building a distribution, the testing step runs all the tests

Organizing tests

- CPAN distributions should include a `t/` directory containing test files `.t`
- When building a distribution, the testing step runs all the tests
- Two common organization of tests are used:

Organizing tests

- CPAN distributions should include a `t/` directory containing test files `.t`
- When building a distribution, the testing step runs all the tests
- Two common organization of tests are used:
  1. Each `.t` file corresponds to a `.pm` file

Organizing tests

- CPAN distributions should include a `t/` directory containing test files `.t`
- When building a distribution, the testing step runs all the tests
- Two common organization of tests are used:
  1. Each `.t` file corresponds to a `.pm` file
  2. Each `.t` file corresponds to a logical feature

Organizing tests

- CPAN distributions should include a `t/` directory containing test files `.t`
- When building a distribution, the testing step runs all the tests
- Two common organization of tests are used:
  1. Each `.t` file corresponds to a `.pm` file
  2. Each `.t` file corresponds to a logical feature
- Hundered of testing modules available

Topic

Modules

- Package contained in its own file and loadable with `use` or `require`

Modules

- Package contained in its own file and loadable with `use` or `require`
- A module must be valid perl code

Modules

- Package contained in its own file and loadable with `use` or `require`
- A module must be valid perl code
- It must end with an expression that evaluates to true, so the parser knows that it has loaded successfully

Modules

- Package contained in its own file and loadable with `use` or `require`
- A module must be valid perl code
- It must end with an expression that evaluates to true, so the parser knows that it has loaded successfully
- When loading a module, Perl splits the pacakge name on `::` and turns the components of the package name into a file path.

Modules

- Package contained in its own file and loadable with `use` or `require`
- A module must be valid perl code
- It must end with an expression that evaluates to true, so the parser knows that it has loaded successfully
- When loading a module, Perl splits the pacakge name on `::` and turns the components of the package name into a file path.
- The search is made in every directory in `@INC`

Using and Importing

- With use perl loads a module from disk and calls import with any arguments provided
- The no builtin calls a module's unimport passing any arguments
- The call to import and unimport happens during compilation

Exporting

- The module `Exporter` is the standard way to export symbols from a module
- Relies on the presence of `@EXPORT_OK` and `@EXPORT`

Export example

```
package StrangeMonkey::Utilities;
use Exporter 'import';

# Will export these symbols upon request
our @EXPORT_OK = qw/round translate screech/;

# Will export these symbols by defaul
our @EXPORT = qw/dance sleep $variable/;

# Then on client code ...will import round and sleep
use StrangeMonkey::Utilities qw/round sleep/;
```

Topic

Distributions

- Collection of metadata and modules into a single, redistributable and installable unit

Distributions

- Collection of metadata and modules into a single, redistributable and installable unit
- The easiest way to configure, build, package, test and install Perl code is to follow the CPAN's conventions

Distributions

- Collection of metadata and modules into a single, redistributable and installable unit
- The easiest way to configure, build, package, test and install Perl code is to follow the CPAN's conventions
- A distribution built on these standards can be tested on several versions of Perl on several different hardware platforms

Attributes of a Distribution

- `Build.PL` or `Makefile.PL` : Drives configuration, build, test, bundle and install

Attributes of a Distribution

- `Build.PL` or `Makefile.PL` : Drives configuration, build, test, bundle and install
- `MANIFEST` : List of all files contained in the distribution

Attributes of a Distribution

- `Build.PL` or `Makefile.PL` : Drives configuration, build, test, bundle and install
- `MANIFEST` : List of all files contained in the distribution
- `META.yml` or `META.json` : Metadata about the distribution and dependencies

Attributes of a Distribution

- `Build.PL` or `Makefile.PL` : Drives configuration, build, test, bundle and install
- `MANIFEST` : List of all files contained in the distribution
- `META.yml` or `META.json` : Metadata about the distribution and dependencies
- `README` : Description of the distribution, copyright and licensing information

Attributes of a Distribution

- `Build.PL` or `Makefile.PL` : Drives configuration, build, test, bundle and install
- `MANIFEST` : List of all files contained in the distribution
- `META.yml` or `META.json` : Metadata about the distribution and dependencies
- `README` : Description of the distribution, copyright and licensing information
- `lib/` : Directory containing Perl modules

Attributes of a Distribution

- `Build.PL` or `Makefile.PL` : Drives configuration, build, test, bundle and install
- `MANIFEST` : List of all files contained in the distribution
- `META.yml` or `META.json` : Metadata about the distribution and dependencies
- `README` : Description of the distribution, copyright and licensing information
- `lib/` : Directory containing Perl modules
- `t/` : Directory containing test files

Attributes of a Distribution

- `Build.PL` or `Makefile.PL` : Drives configuration, build, test, bundle and install
- `MANIFEST` : List of all files contained in the distribution
- `META.yml` or `META.json` : Metadata about the distribution and dependencies
- `README` : Description of the distribution, copyright and licensing information
- `lib/` : Directory containing Perl modules
- `t/` : Directory containing test files
- `Changes` : Text Log of every significant change to the distribution

CPAN Tools for managing distributions

- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines

CPAN Tools for managing distributions

- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines
- `CPAN.pm` official CPAN client

CPAN Tools for managing distributions

- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines
- `CPAN.pm` official CPAN client
- `ExtUtils::MakeMaker` Package, build, test and install Perl distributions works with `Makefile.PL`

CPAN Tools for managing distributions

- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines
- `CPAN.pm` official CPAN client
- `ExtUtils::MakeMaker` Package, build, test and install Perl distributions works with `Makefile.PL`
- `App::cpanminus` configuration-free CPAN client

CPAN Tools for managing distributions

- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines
- `CPAN.pm` official CPAN client
- `ExtUtils::MakeMaker` Package, build, test and install Perl distributions works with `Makefile.PL`
- `App::cpanminus` configuration-free CPAN client
- `App::perlbrew` helps you to manage multiple installations of Perl.

CPAN Tools for managing distributions

- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines
- `CPAN.pm` official CPAN client
- `ExtUtils::MakeMaker` Package, build, test and install Perl distributions works with `Makefile.PL`
- `App::cpanminus` configuration-free CPAN client
- `App::perlbrew` helps you to manage multiple installations of Perl.
- `CPAN::Mini` Allows to create a private mirror of teh public CPAN

CPAN Tools for managing distributions

- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines
- `CPAN.pm` official CPAN client
- `ExtUtils::MakeMaker` Package, build, test and install Perl distributions works with `Makefile.PL`
- `App::cpanminus` configuration-free CPAN client
- `App::perlbrew` helps you to manage multiple installations of Perl.
- `CPAN::Mini` Allows to create a private mirror of teh public CPAN
- `Dist::Zilla` Automatest away common distribution tasks

CPAN Tools for managing distributions

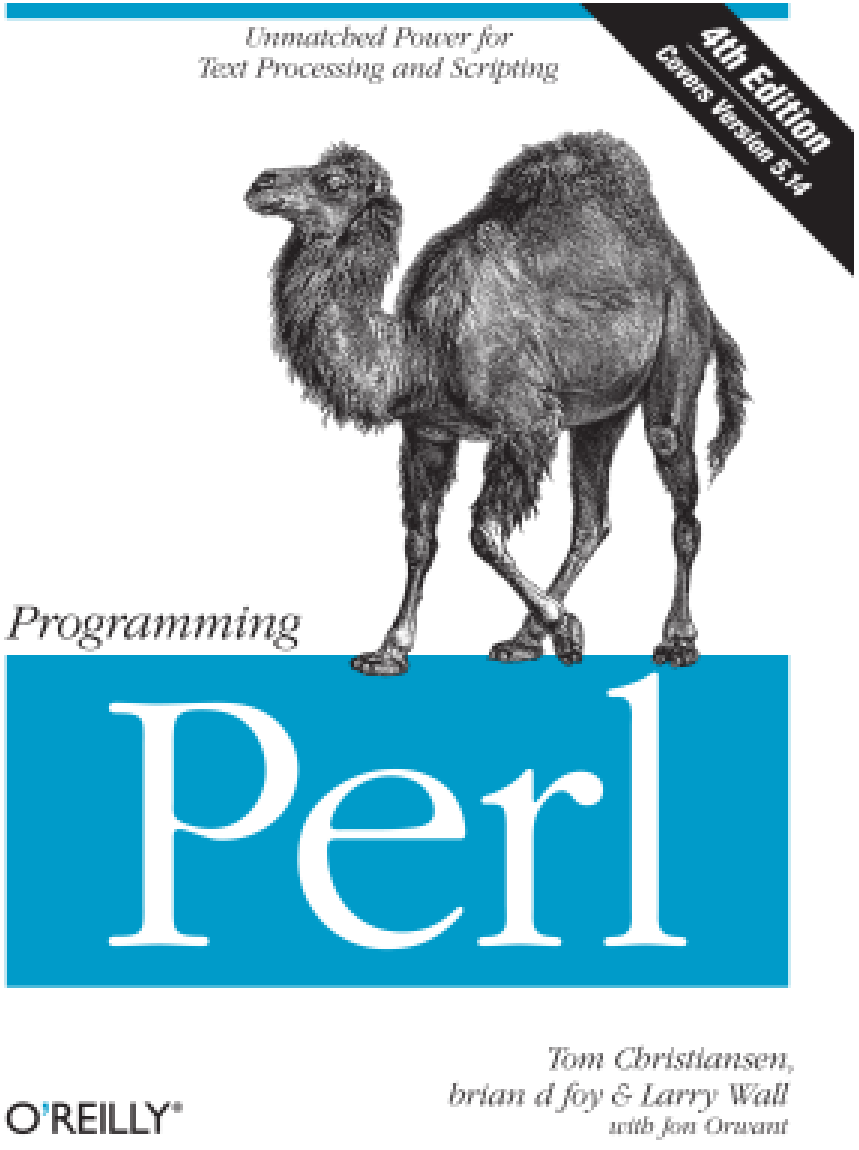
- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines
- `CPAN.pm` official CPAN client
- `ExtUtils::MakeMaker` Package, build, test and install Perl distributions works with `Makefile.PL`
- `App::cpanminus` configuration-free CPAN client
- `App::perlbrew` helps you to manage multiple installations of Perl.
- `CPAN::Mini` Allows to create a private mirror of teh public CPAN
- `Dist::Zilla` Automatest away common distribution tasks
- `Carton and Pinto` Manage and install code's dependencies

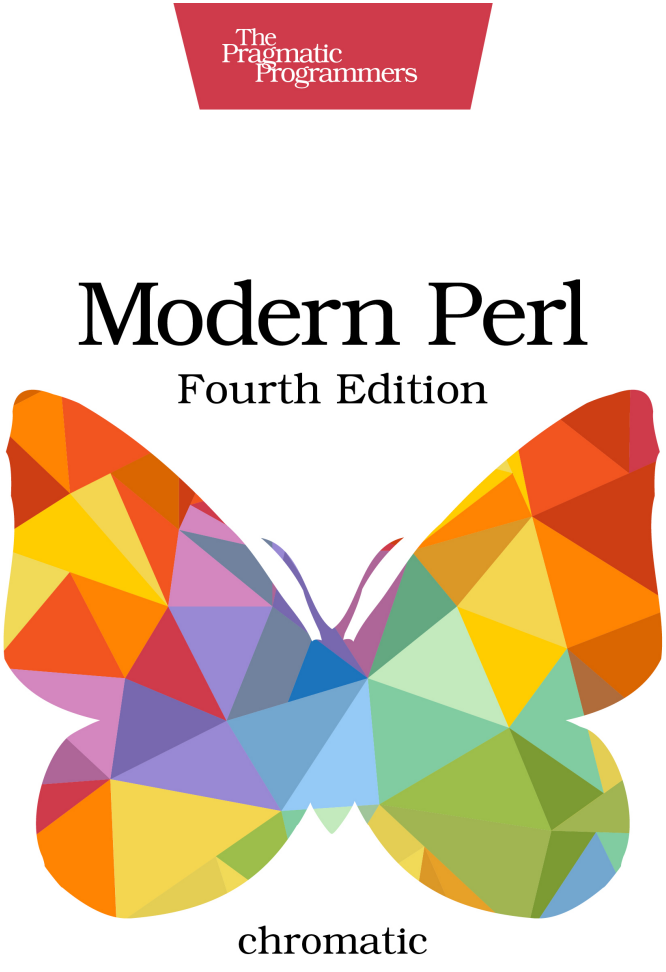
CPAN Tools for managing distributions

- `CPANTS` `http://cpants.perl.org` evaluates each uploaded distribution against packaging guidelines
- `CPAN.pm` official CPAN client
- `ExtUtils::MakeMaker` Package, build, test and install Perl distributions works with `Makefile.PL`
- `App::cpanminus` configuration-free CPAN client
- `App::perlbrew` helps you to manage multiple installations of Perl.
- `CPAN::Mini` Allows to create a private mirror of teh public CPAN
- `Dist::Zilla` Automatest away common distribution tasks
- `Carton and Pinto` Manage and install code's dependencies
- `Module::Build` alternative for `ExtUtils::MakeMaker` written in pure Perl

Topic

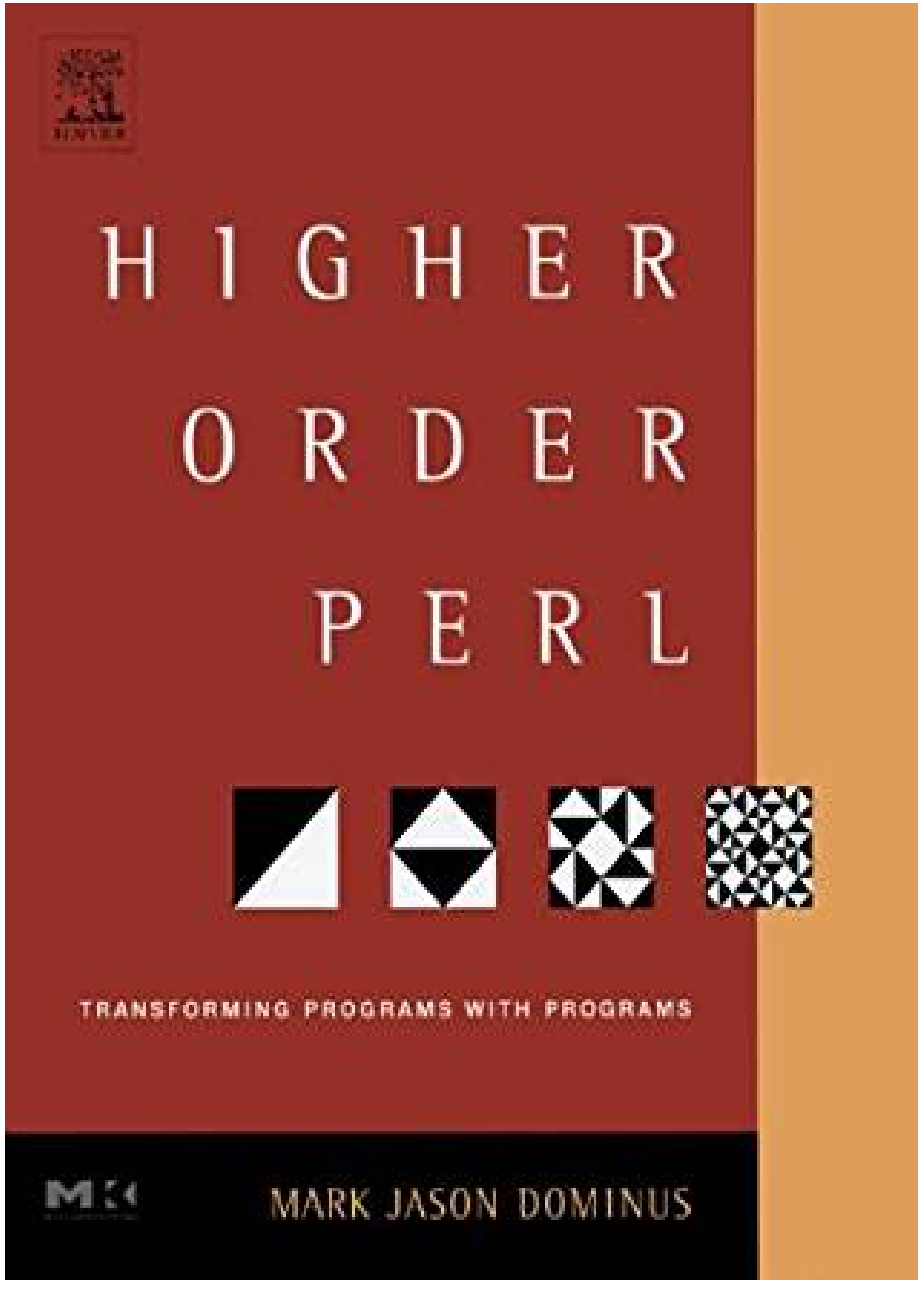
Camel book - http://shop.oreilly.com/



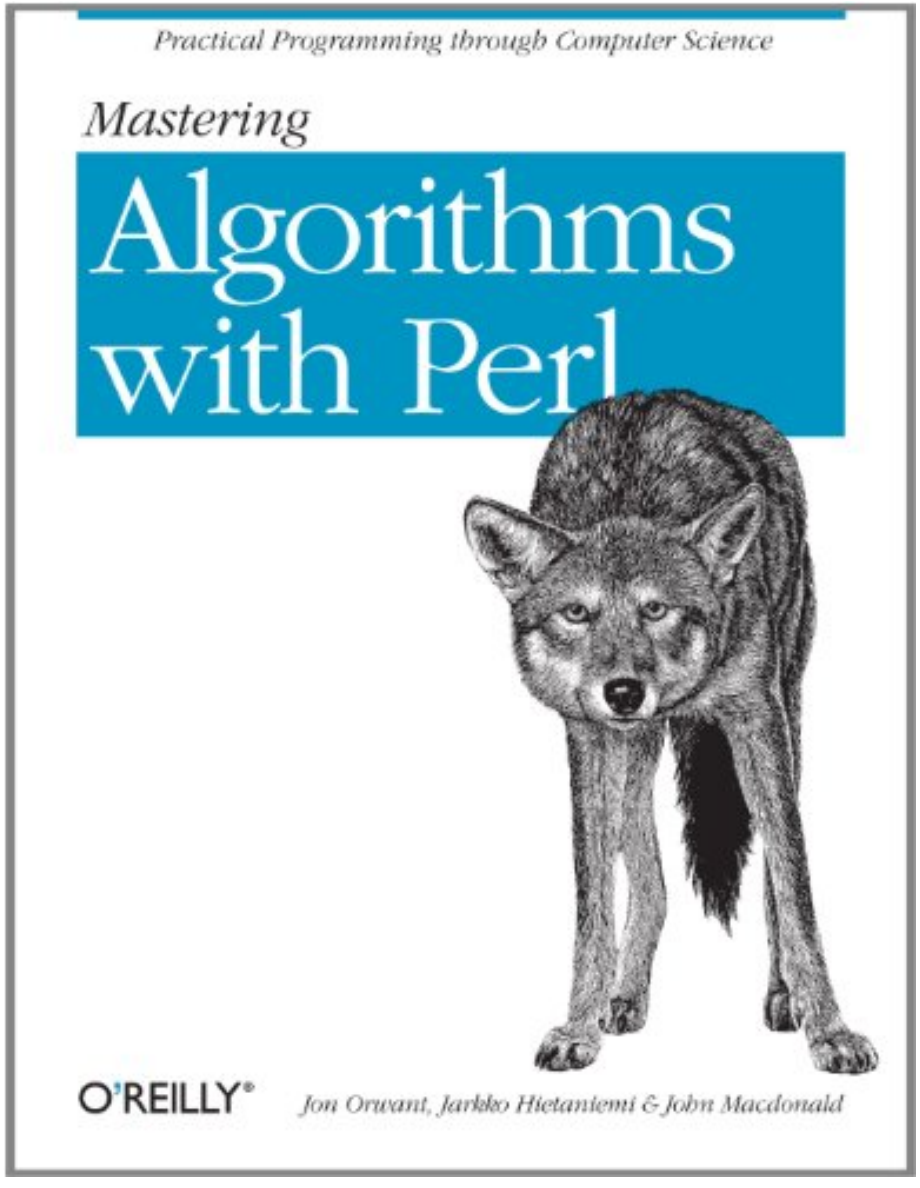

The Pragmatic Programmers
Modern Perl
Fourth Edition
chromatic
The Classic Reference, Updated for Perl 5.22

Perl Best Practices - http://shop.oreilly.

Higher Order Perl -



Mastering Algorithms with Perl - http://shop.



Topic

Database

DBI - http://dbi.perl.org/

DBIx::Class - http://www.dbix-class.org/about.html

Web programming

Plack - `http://plackperl.org/`

Web frameworkds

- Catalyst - http://www.catalystframework.org/
- Dancer - http://perldancer.org/
- Mojolicious - http://mojolicious.org/

Data Science and Math

PDL - http://pdl.perl.org/

Image Manipulation

Imager - https://metacpan.org/pod/Imager

Life Science

BioPerl - http://bioperl.org/

Object Oritentation Systems

```
http://moose.iinteractive.com/en/
```

Topic

Perl special blocks

```
print                "    PRINT: main running\n";
die                  "    DIE:   main dying\n";
die                  "DIE XXX /* NOTREACHED */";
END          { print "1st END:   done running"    }
CHECK        { print "1st CHECK: done compiling"  }
INIT         { print "1st INIT:  started running" }
END          { print "2nd END:   done running"    }
BEGIN        { print "1st BEGIN: still compiling" }
INIT         { print "2nd INIT:  started running" }
BEGIN        { print "2nd BEGIN: still compiling" }
CHECK        { print "2nd CHECK: done compiling"  }
END          { print "3rd END:   done running"    }
```

Topic

-Ofun

Acme namespace

`https://metacpan.org/search?size=20&q=Acme`

JAPH / Obfuscated Perl Contest

`https://en.wikipedia.org/wiki/Obfuscated_Perl_Contest`

The dromedary

__END__

Q/A

__END__