Perl in Plain English

| PiPE

Sample code:

A dog is a type of object.

It has a string called a name.

By default, a dog's name is "Fido".

A dog must have an int called its age.

It may have a human called its owner.

It has a string called its color. Its color cannot change. A dog's color is usually "black".

A dog can run_fast, play_fetch, and roll_over.

To use a dog as a string, say its name.

A dog cannot be used as a number.

What it means:

```
A NAME is a type of TYPE                # define an object
It has a TYPE called its ATTRIB         # declare an attribute
NAME's ATTRIB                           # access an attribute
Its ATTRIB cannot change                # explicit read-only
NAME must have (a/an) ATTRIB            # make an attribute required
A NAME can SUB                          # declare a prototype
By default, ...                         # default attribute state
To use a NAME as a TYPE, EXPR           # coercion
A NAME cannot be used as a TYPE         # explicitly prevented coercion
VAR is a TYPE                           # new variables are lexically scoped
                                        #    by default
… is a global TYPE                      # 'our' becomes 'global'
… is a local TYPE                       # 'local' is still just 'local'

... is a …  #implies a copy        ... is the …  #implies a reference
```

Some built in functions and operators:

```
… plus …                          # +
… gets …                          # =
… joined to …                     # .
… INT to INT …                    # ..
... shifted from…                 # shift
... gets …                        # push
... popped from…                  # pop
… selected by EXPR                # grep
… mapped by EXPR                  # map
Print STR                         # print
Pronounce STR                     # printf
Say STR                           # say
… (sorted) (LIST/HASH)            # sort

    Sort (LIST/HASH)  #like (LIST = sort LIST)

NOTHING  #bareword constant for the empty string

# Any number or operator may be expressed as a symbol or
# fully written out
```

What about control structures?

```
# control structures and blocks are opened by ':' and closed by '.'
# lines are separate by a ';'

Do: ... .                    # basic 'do' block
LABEL does: ... .            # labeled block

If BOOL: ... .
    Otherwise if BOOL: ... .
    Otherwise: ... .
EXPR Unless BOOL.            # natural language syntax
... while ...
... until ...

For each item in LIST: ... .
    or
For each VAR in LIST: ... .

Be done with LABEL.          # jumps out of all loops up to LABEL
Stop this.                   # last
Redo this.                   # redo
Do the next one.             # next
```

Some other syntax bits:

Lists - comma separated, final item is preceded by 'and' - (one, two, and three)

Type Casting: single - '... is (a/an) …' list - '... are …'
ex. *'foo* is a *scalar'* *'bar* is an *array of ints*' '*bits*, *bots*, and *bats* are *hashes*'
uses coercions

(... gets …) is an assignment, (… equals …) is a comparison, (... is …) becomes context based:
ex. (NAME's ATTRIB is …) or (**If** NAME's ATTRIB is …)
for arrays and hashes, (... has ...) is used for context-based assignment/comparison

Assignment: scalar/scalar, array/array, hash/hash - '... is …' or '... is the same as ….'
scalar/primitive - '... gets ...' array/primitive - '... gets ...' hash/primitive - '... gets ...'
array/list - '... gets each of …' hash/list - '... gets pairs of ...'
array/list or hash/list - '... gets ...'
list/list - '... have each of …'
create a reference - '... is the ...'

Comparators: equality - '... is equal to ...' or '... equals ...' # can still say '=='
precedence - '... is less than ...' or '... is greater than …' # or '>'
'... is greater than or equal to … ' # or '>='
comparison - '... compared to ...' or '... compared with …' # etc.

Subscripts: array - 'the item at INDEX in ARRAY' or 'INDEX in ARRAY' # or: the [first] item in ...
hash - 'the value of KEY in HASH' or 'KEY in HASH'
ex. of multi-level indexing - Scalar_Foo gets 3 in 5 in 2 in Array_Bar

Subroutines and other details:

```
mySub is a subroutine.                                  # prototype
mySub does:                                             # define with 'SUB does: ...'
        Say 'subroutines can be prototyped';
        Say NOTHING.

myMultiline does:                                       # no prototype
        say "Use semicolons to delimit lines";          # semicolons end lines in blocks
        say "then you can have lots of whitespace";

        say "even multiple paragraphs!".                # block ends with a period

mySecondExample does the following:
        An exampleVar gets 3;
        Say "Double-quoted strings can interpolate variables:";
        Print "[TAB]For example, 1 + 2 = [exampleVar].[NEWLINE]";  #syntax from Inform7
                #some barewords interpolate to system values
        Say NOTHING;
        Say "Another trick, [a local exampleVar that gets 'the elusive inline assignment']!";
        Say "What is exampleVar now? Did you guess [exampleVar]?".     #still 3
```

Subroutine examples:

IOExamples does the following:

A local array is called list_of_lines;
another_list is an array;
There is a string called ‘previousLine’;

For each 'bar' in STDIN:

list_of_lines has bar; #’it’ is the same as $_
another_list has a copy of the item; #BAD! Same as above, but messier

previousLine is the bar; #BAD! makes an alias to ‘bar’, which will disappear
previousLine gets bar;

If previousLine equals the item, say “you have the same line twice!” and stop this;

Do the next one. # next in the loop

Returns list_of_lines.

Subroutine examples:

#Special ‘defined fields’ can be accessed from within the subroutine. They can be defined anywhere in the
code after a subroutine declaration.
ArgsSub is a subroutine. It has a string called ‘SecondArg’, sometimes a scalar called ‘ThirdArg’, and must have a string called ‘FirstArg’. It may have an array called ‘extra_args’.

#This is tricky: ‘sometimes’ implies low priority -> in other words, ‘SecondArg’ only gets filled after #‘FirstArg’ and ‘ThirdArg’ are full. Finally, any other arguments get dumped into ‘extra_args’.

#Hierarchy of Needs:
… never has TYPE - causes other fields to ignore values of TYPE
… must have - throws error if not filled
… has
… usually has
… sometimes has
… may have

#Arguments are shifted from @_ into each defined field, first by the weight of the field’s need, then
#left-to-right along the list as assembled at compile time (top to bottom of the program).

Do ArgsSub with “I’m first!”, “Maybe I’m second”, “I’m definitely third”, and “I’ll just go to waste”.

#To avoid confusion, list defined fields by weight of need
BetterArgsSub is a subroutine. It must have a string called ‘most_urgent’, has a string called ‘less_urgent’, and sometimes has a string called ‘ambivalent’.

Subroutine examples:

#If you want to prevent a certain TYPE value from being used, define a ‘never’ field of that TYPE

pickyConcat is a subroutine that never has an array, must have a scalar called ‘left’, and must have a scalar called ‘right’. It returns “[left][right]”.

#Special case for one-line routines -> ‘SUB returns … ’

#So, to enforce a strict usage of certain TYPEs with a subroutine, use only defined fields of those types,
and do not access @_ (which still carries excess parameters from the call) while inside the sub

Do pickyStringConcat with too, many, params, and an array that has (2, 3, and 4).

#Does nothing with the third argument, and ignores the last

Finally, pragmas, packages, and subroutines:

Using **PRAGMA**,	# use PRAGMA
Let's talk about **PACKAGE**.	# switch to package
In Plain English,	# interpret as PiPE
In Perl,	# interpret as traditional Perl

So, let's see it all together.

```
Use Language::PiPE;          # someday soon!
```

Let’s talk about Example.

In Plain English,
Print “Maximum: ” to STDOUT. A scalar called max reads from STDIN. An array called list gets each of 2 to max.
While next is a scalar that gets shifted from list:
    Say next to STDOUT;
    list is selected by the topic modulo next, in list.

In Perl,

```
print STDOUT 'maximum:';
my $maxim = <STDIN>;
my (@list) = (2..$maxim);
while ($next = shift @list)
{
    print STDOUT $next, "\n";
    @list = grep {$_ % $next} @list;
}
```