Perl Level1: Programming using natural language

Module MP85 – version n°2 – January 2014

Powertrain University

January 2014

Your contact

- Module MP85
- Author & Trainers
  - Benoit DELAMARE
    - VTI – PCE Cergy
    - 01 34 33 14 67
  - Alexandre de Jouvencel
    - VTI – PEL Cergy
    - 01 34 33 19 05

Objectives

- Discover PERL programming language, used for extracting, handling and reformatting data from one or several files.
- Understand and use Regular Expressions.
- Be able to write simple scripts in PERL language.

Summary

Perl Overview

Using a natural language

January 2014

Perl Overview – Introduction 1/2

- P.E.R.L Practical Extraction and Report Language,
- Created in 1986 by Larry Wall (system engineer) to manage a “News” system between two networks.
- It’s
  - A programming language
  - A free software
  - An Interpreted language:
    - No compilation
    - Less quick as compiled program
    - Each script requires Perl interpreter
- Why is it popular?
  - Portability: interpreter exist for most existing platform. (Unix, NT, Window, Mac, VMS, Amiga, Atari…)
  - Free cost, available in internet, with lot of library and utilities
  - Simplicity: few commands to do same thing as 500 lines for C program
  - Robustness: No memory allocation, no limited strings, stacks or variable names…

Perl Overview – Introduction 2/2

- Which use?
  - At beginning Perl was created to:
    - Manage Files (including managing several files in same time)
    - Manage Text (Search, substitution)
    - Manage Process (including throw networks)

    → Designed primarily to UNIX world
  - An now Perl is used to:
    - Generate and update HTML files (including to write CGI)
    - Universal access to database
    - File formats conversion

    → Not anymore linked to UNIX world
  - Perl is not designed to:
    - Write interactive **graphical user** interface
    - Scientific computing.

Perl overview – Perl a natural language 1/2

```
print "Hello World\n";
```

- Perl was created by a linguist, as a natural language:
  - Single things should remain
  - Complex things should not be impossible

    → Expect to evolve ➔ Open source language

- Perl motto is TIMTOWTDI BSCINABTE (pronounced "Tim Toady Bicarbonate"):

  “There's more than one way to do it”, “but sometimes consistency is not a bad thing”

- Language

  = Words and Sentences
  = Fragment into smaller semantic unit or combine into larger unit

  Meaning of each unit depends on syntactic, semantic and pragmatic context of unit location:
  “Pile” (dog) is a noun. But for this sentence, each pile (dog) word have difference sense

“Je pile face à une pile de piles à minuit pile, j’aurai besoin d’une pile!”
“If you dog a dog during dog days of summer, you’ll be dog tired dogcatcher!”

Perl overview – Perl a natural language 2/2

- In classical informatics languages, variables are classified by their type. In natural language first distinction is made between “singular” and “plural”:
  - Strings / Numbers are piece of singular data.
  - String or number lists are plural data.
- In Perl
  - `scalars` are singular variables
  - `Lists (arrays)` are plural variables

```
$Sentence = "Hello world\n";
print $Sentence;
```

Note: No variable declaration!!
Not initialized variables start to exist only when it is required!

Perl overview – Names

Variables types:

Type	Character	Sample	Is a name for:
Scalar	$	$cents	Individual value
List (Array)	@	@large	Value list, indexed by number
Hash	%	%interest	Value group, indexed by string
Sub-program	&	&comment	Callable code part
typeglob	*	*type	All that is called « type »

Perl overview – Singular

- According to context, variables are automatically interpreted as string, number or boolean:

```
$Camel = '123';
print $Camel + 1, "\n";
```

- The " character is used to force interpolation.

'\n' is different than **"\n"**

Perl overview – Exercice

- How these variables will be evaluated ?

```
$answer = 42;                          # An integer
$pi = 3.14159265;                      # A “real” number
$Engineer = 6.02e23;                   # A scientific notation
$animal = "Camel";                     # A string
$sign = "My $animal and me";           # A string with interpolation
$cost = ’that costs $100’;             # A string without interpolation
$why = $because;                       # An other variable
$x = $animal * $Engineer;              # An expression
$cwd = `pwd`;                          # A String generated by a command
$exit = system("vi $x");               # A command return code
$fido = new Camel "Fido";              # An object
```

Perl overview – Plural

List 1/2

- Array: ordered scalar list, indexed by scalar position in the list. It can contain string, number, reference or both.
- To assign:

```
@house = (‘bed’, ‘chair’, ‘table’);
```

- In the same time scalar can be assigned from list:

```
($tata, $toto, $titi) = @house;
```

- Exercise: Only using assignation swap two variable values

```
($alpha, $omega) = ($omega, $alpha);
```

Perl overview – Plural

List 2/2

- As in C, Perl list starts from zero.
- To access to list element using scalar context, use:

```
$house[0] = ‘bed’;
$house[1] = ‘chair’;
$house[2] = ‘table’;
```

- As array are ordered, several operations are existing, as **push** and **pop** (LIFO)

Perl overview – Plural

Hash 1/2

- Hash: unordered scalar set, indexed by string associated to each scalar.
- Very efficient and quick search system, whatever element number.
- Hash can be assigned by array, but each element par is interpreted as a key/value par:

```
%longday = (‘Sun’, ‘Sunday’, ‘Mon’, ‘Monday, ‘Tue’, ‘Tuesday);
```

Perl overview – Plural

Hash 2/2

- To make easier assignation, the following syntax is useful:

```
%longday= (
        ‘Sun’ => "Sunday",
        ‘Mon’ => "Monday",
        ‘Tue’ => "Tuesday"
);
```

- If Hash is used in list context all items are converted in Key/Values, in any order.

- To extract key list, use keys. To order key list, user sort function before:

```
@keylist = sort keys %longday;
```

- To use Hash table in scalar context, use following syntax:

```
$woman{"Adam"} = "Eve";
```

Debugger

Perl –d myScript.pl

January 2014

Debugger – Step by Step Commands

- s (EXPR)

  Command execute one step, he goes inside call of sub-program. If EXPR is specified and provides call to sub function, debugger goes inside the sub-program

- n (EXPR)

  Command execute one step, he doesn't go inside call of sub-program.

- <Enter>

  Debugger execute the last n or s command

- .

  Return the internal debugger pointer to the last executed line and displays this line

- r

  This command continues until the current sub-program is executed new time

Debugger – Breakpoint Commands

- **b**

  b Line

  b Condition

  b Line Condition

  b SubProgramName

  b SubProgramName Condition

  b Load FileName

  Break point only can be set on executable line

- **d**

  d Line

  Remove a breakpoint

- **D**

  Remove all breakpoints

- **c**

  c Line

  Continue the execution, setting a breakpoint active only one time at the specifying line

Debugger – Trace/Display Action Commands

- T

  Provide a rear trace of the call stack

- x

  x EXPR

  Evaluate the specified expression in a list context and displays result

- a

  a COMMAND

  a LINE

  a LINE COMMAND

  Set an action to do before executing the LINE (or current line if LINE is not specified) if command is not specified, action is removed from the LINE (or current line)

- A

  Remove all actions

Debugger – Special Commands

- q

  Quit the debugger

- R

  Resart the debugger, launching new session with exec. Most debugger configuration (breakpoint, action, history are recorded)

First Perl Program

January 2014

Perl First Perl Program - Subject

- We have a note set for each student class member. We need a combined list, providing notes list and average, for each student.

- We have a text file, named « notes », providing:

```
Thuc 25
Benoit 12
Alex 54
Thuc 6
Alex 78
…
```

Perl – First Perl Program

```
# Open notes file
open (NOTES, "notes") or die "Can't open notes files: $!\n";

# Browse opened file, and store detected par student / note
while ($line = <NOTES>) {
    $student, $note) = split(/ /, $line);
    $notes{$student} .= $note . " ";
}

# Organize student notes, and compute the total
foreach $student (sort keys %notes) {
    $score = 0;
    $total = 0;
    @notes = split(/ /, $notes{$student});
    foreach $note (@notes) {
          $total += $note;
          $score++;
    }

    # Evaluate student average
    $average = $total / $score;

    # Display student combined notes and average
    print "$student: $notes{$student}\tAverage: $average\n";
}
```

Perl First Perl Program – File Handles 1/2

- File Handle is like C/C++ “stream”, used to reference file, periphery, socket or pipe
- Predefined handle:
  - STDIN
  - STDOUT
  - STDERR
- Open it using open instruction:

open (NOTES, "filename");	**# Read an existed file**
open (NOTES, "<filename");	**# (Same thing, but explicit syntax)**
open (NOTES, ">filename");	**# Create file in write mode**
open (NOTES, ">>filename");	**# Append in existing file**
open (NOTES, "| command_output");	**# Create an output filter**
open (NOTES, "command_input |");	**# Create an input filter**

- Read handle line, using *diamond* operator: <HANDLE>

Perl First Perl Program – File Handles 2/2

- Exercise: Ask for a number and display get number:

```
print STDOUT "Type a number: ";              # Ask for a number
$number = <STDIN>;                           # Get specified number
print STDOUT = "Number is $number\n";        # Display the number
```

- chop instruction, removes (and returns) last character

- chomp instruction, removes last record markers (and returns number of removing characters)

```
print STDOUT "Type a number: ";              # Ask for a number
chomp($number = <STDIN>);                    # Get specified number
print STDOUT = "Number is $number\n";        # Display the number
```

Perl First Perl Program – Operators

- Most of Perl operator are borrowed from C, except some string specific ones:
  - Arithmetic operator : Same + - * / % **
  - String operators: .x
  - String concatenation: automatic during interpolation
  - Assignation: = as in C, and more
  - Auto increment and decrement: $a++, ++$a, $b--, - -$b
  - Logical operators: && and || or ! Not
  - Comparison operators:

Comparison	Numerial	String
Equal	==	Eq
Not equal	!=	Ne
Less than	<	Lt
Higher than	>	Gt
Less or equal to	<=	Le
Comparison	<=>	cmp

  - Test file operators:

Test	Name
-e $a	Exist
-r $a	Read mode
-w $a	Write mode
-d $a	Folder
-f $a	file
-T $a	Text file

Perl First Perl Program – Control Structure

- Boolean interpolation rules:
  - All strings are true, except “” and “0”
  - All numbers are true, except 0
  - All references are true
  - All undefined value are false.

What are interpreted “boolean” values of following expression:

```
0          # Becomes "0" string so false
1          # Becomes "1" string so true
10 – 10    # 10 – 10 is 0, becomes "0" string so false
0.00       # Becomes 0 and "0" string so false
"0"        # "0" string so false
""         # Empty /null  string so false
"0.00"     # "0.00", not empty and not really equal to "0". so true
"0.00" + 0 # Number 0 (cast by  +), so false
\$a        # A reference to $a, so true even if $a is false
undef()    # function that returns undefined string, so false
```

Perl First Perl Program – Loop Instruction

- while and until: work as the same manner if and unless instruction
- for : as in C.
- foreach: to execute same code for each element of a known scalar set (as an array)

```
foreach $user (@users) {
        print "$user is a good guy\n" if (-f $home{$user});
}
```

- next operator, to jump to next element in the loop
- Last jump to end of bloc, as specified condition returns false
  - It is possible to notify nested loop, using a label to target which is the associated loop linked to the Next or last instruction

Regular Expressions

RE or RegExps

January 2014

Regular Expressions – Overview 1/2

- Regular expression is writing a concise set of strings.
- Regexps are used by most UNIX text processors, using *grep*, *findstr*, *sed* and *awk*

Regular Expressions – Overview 2/2

- Perl use Regexps
  - In conditional expressions, to check is a string matches with a given pattern. When something like **/jigger/** is detected, it's a *correspondence search operator*
  - Now, if we can found pattern in a string, we can replace them by other things: **s/jigger/thingy/** requests to substitute "**jigger**" by "**thingy**". It's a *substitution operator*
  - Finally, pattern can specify place where something is, but also place where something is not.
- Simplest use of RE is to search literal expression:

```
# Find and print all http links from an HTML file
while ($line = <FILE>) {
  if ($line =~ /http:/) {
      print $line;
  }
}
```

How to manage ftp and mailto link?
And how to manage new link type?

Regular Expressions – Character Classes

- As RE describes a set of strings, we can only describe what we look for: several alphabetic characters, following by colon : /[a-zA-Z]+:/
  - where brackets provide a *character class*. a-z and A-Z represent all alphabetic characters, (**-** is the interval character between first and last (included) specified characters)
  - + is a quantifier, meaning one or several characters
- As several character classes are frequently used, Perl defines special characters to replace them:

Name	ASCII definition	Character
All characters (expect new line)	[^\n]	.
Whitespace	[\t\n\r\f]	\s
Not a Whitespace	[^ \t\n\r\f]	\S
Word character	[a-zA-Z_0-9]	\w
Not a word character	[^a-zA-Z_0-9]	\W
Digit	[0-9]	\d
Not a digit	[^0-9]	\D

Regular Expressions – Quantifiers

- Characters and Character classes search for unique characters. To search several characters, we must specify a *quantifier*:

- Quantifier general form specify minimum and maximum number of expected characters: to specify a US phone number between 7 to 11 digits: /\d{7,11}/
  - **\d{8} means 8 digits are expected**
  - **\d{7,} means 7 digits at least are expected**
  - **To specify that maximum of 11 digits are expected, you must use the \d{0,11} syntax.**

- As several quantifiers are frequently used, Perl defines special characters to replace them:

Name	Usual quantifier	Character
At least one element 0 occurrence or more element 0 or one element	{1, } {0, } {0,1}	+ * ?

Regular Expressions – Boundary & BackReferences

- Each time we look for a pattern, Perl try until it finds one. An *anchor* can delimit range where search is performing. Anchor doesn't provide a character, but represent a boundary

Name	Character
Word boundary	**\b**
Not a Word boundary	**\B**
Beginning of a string	**^**
End of a string	**$**

- Parentheses are used to group characters before a quantifier, but they can be used to remember to found part, for future use. /\d+/ and /(\d+)/ will search both as much as possible number.

- Using recorded reference depends on the context: in the same RE, use an antislash following by an integer representing the set of parentheses, determined by counting left parentheses from the left of the pattern

- To use a found reference outside the RE, use special variable, $x as it's a normal scalar variable

Regular Expressions – Modifiers

- RE can be completed by modifiers, in order to change pattern detection compartment:

Modifiers	Means
/i	Ignore the character case
/m	^and $ are not string boundary but become line boundary (near a \n)
/s	Allow to . to detect new lines and ignire obsolete $* varaiable
/x	Ignore most blank and allow to use comment inside the pattern
/o	Compile the pattern only one time
/g	Global search, to detect all occurrences
/cg	Allow to continue to search after one fail in /g
/e	Evaluate the right side of a substitution pattern as an expression

- Exercice:
  - From the string $String="password=xyzzy verbose=9 score=0"; create an hash table %hash = (password =>xyzz, verbose => 9, score => 0); using RE

```
%hash = $String =~ /(\w+)=(\w+)/g;
```

Regular Expressions – Quick Reference Guide

Adobe Acrobat Document

Special Names

January 2014

Special Names 1/3

- **$*Number*** Contain text found by the set of parentheses associated to the last pattern search
- **$_** Default variable (for search) feature
- **@_** Default list (for split feature)
- **@ARGV** In a sub program, this array contains the argument parameter list
- **%ENV** Hash table providing current environment variables

Special Names 3/3

- **$+** Returns result of the last parentheses search, in the current active context
- **$&** Returns string found during the last pattern search, in the current active context
- **$’** String that follows all what were found by the last successful pattern search, in the current active context
- **$`** String that is before all what were found by the last successful pattern search, in the current active context

Special Names 2/3

- **$!** In a numeric context, returns the last system call error. In a string context, returns the associated error message
- **STDERR** Special file handle for standard error output
- **STDIN** Special file handle for standard input
- **STDOUT** Special file handle for standard output

Sub-Programs

January 2014

Sub-Programs – Declaration and Definition

- To declare and define a sub program, use one a the following syntax:
  - **sub NAME BLOCK**
  - **sub NAME PROTO BLOCK**
  - **sub NAME ATTRS BLOCK**
  - **sub NAME PROTO ATTRS BLOCK**
- To create an anonymous sub program, only forget the NAME. Be careful to have a mean to call the sub function. (save the return value):
  - **$refsub = sub BLOCK;**
- To import sub-program defining in another module:
  - **use MODULE qw(NAME1 NAME2 NAME3…);**

Sub-Programs – Call

- To call sub-programs directly, use:
  - **NAME (LIST)**
  - **NAME LIST**
  - **&NAME** *Pass the current @_ to the sub-program*
- To call sub-program using indirect method:
  - **&$refsub(LIST)**
  - **$refsub→(LIST)**
  - **&$refsub** *Pass the current @_ to the sub-program*

Sub-Programs – Variable Scope

- A variable doesn’t require to be declared and initialized before to be used.
  - But it is recommended to declare them, in order to allow to compiler to check if a used variable is declared
- Two variable types: *global* (to the package) or *lexical* (local to a block)
  - A not declared variable is global
  - *my* is used to declare a lexical variable, only be used in associated block
  - *local* is used to declare a lexical variable, be used in associated block and all child blocks

Sub-Programs – Parameters 1/2

- All arguments are stored in the array @_ laid flat and pass as a parameter. To use it, only copy all @_ values into a list of *my*.

```
sub set_env_perhpas {
          my ($key, $value) = @_;
          $ENV{$key} = $value unless $ENV{$key};
}
```

- Parameters don't need to be named, you can directly use @_ content:

```
sub max {
          my $max = shift (@_);
          for my $element (@_) {
               $max = $element if $max < $element;
          }
         return $max;
}

$BestDay = max($Mon, $Tue, $Wed, $Thu, $Fri);
```

Sub-Programs – Parameters 2/2

- Example where arguments are not named in order to modify them:

```
upperLetter_here($v1, $v2);
sub upperLetter_here {
        for (@_) {tr/a-z/A-Z/}
}
```

- Attention, this syntax will raise an exception if you use a scalar data to call this function `upperLetter_here(Philip);`

- To be sure it's better to work on a parameter copy, and return a result:

```
($v3, $v4 ) = upperLetter($v1, $v2);
sub upperLetter {
  my @parms = @_;
  for (@parms) {tr/a-z/A-Z/}

   #check if it was called in list context
   return wantarray ? @params : $parms[0];
}
```

Sub-Programs – Prototypes

- Prototype is used to create automatic model of call context. Perl will complete automatically missing \ or update the call context in function of the prototype
  - **Sub mypush(\@@) Perl is waiting for an array reference following by a list.**

This function will be called myphush LISTE, LISTE, and Perl will automatically complete the first list to obtain its reference

  - **Other samples:**

Declare as	Called by
`sub mylink($$)`	`mylink $old, $new`
`sub myreverse (@)`	`myreverse $a, $b, $c`
`sub myjoin($@)`	`myjoin ‘‘:’’, $a, $b, c`
`sub mypop(\@)`	`mypop @array`
`sub mysplice(\@$$@)`	`mysplice @array, @array, 0, @pushme`
`sub mykey(\%)`	`mykey %{$hashref}`
`sub mypipe(**)`	`mypipe READHANDLE, WRITEHANDLE`
`sub myindex($$;$)`	`myindex $getstring, ‘‘substr’’`
	`myindex $getstring, ‘‘substr’’, $start`
`sub mywrite(*$;$$)`	`mywrite OUTF, $buf`
	`mywrite OUTF, $buf, $length($buf)-$off, $off`
`sub myopen(*;$@)`	`myopen HANDLE`
	`myopen HANDLE, $name`
	`myopen HANDLE, $name, ‘‘-|’’, @cmd`
`sub mygrep($@)`	`mygrep {/toto/} $a, $b, $c`
`sub myrand($)`	`myrand 42`
`sub mytime()`	`mytime`

Usual Functions

January 2014

Usual Functions – Attention !!

- This section doesn't provide all useful Perl functions. It provides only the most interesting "automatic" Perl functions… in my opinion.
- To have a real overview about Perl internal functions, please refer to "Functions" chapter of the "*Programming in Perl*" book
- Others interesting functions are providing by additional modules. To have an overview to them, pleas refer to "Standard Modules" chapter of the "*Programming in Perl*" book

Usual Functions – 1/4

- Scalar management
  - **chomp / chop**
  - **lc / lcfirst / uc / ucfirst**
  - **length**
  - **reverse**
  - **substr**
- Regular Expressions and Pattern Search
  - **m//**
  - **s///**
  - **split**

Usual Functions – 2/4

- Array / List / Hash management
  - **pop /push**
  - **shift /unshift**
  - **splice**
  - **join**
  - **sort**
  - **delete**
  - **each**
  - **exists**
  - **keys**
  - **values**

Usual Functions – 3/4

- Input/Output
  - **close / closedir**
  - **die**
  - **eof**
  - **mkdir**
  - **open /opendir**
  - **getc**
  - **print / printf**
  - **read / readdir**
- Files and Folders
  - **chdir**
  - **chmod**
  - **glob**
  - **rename**
  - **rmdir**
  - **stat**
  - **unlink**

Usual Functions – 4/4

- Miscellaneous
  - **defined**
  - **exit**
  - **wantarray**
  - **undef**
  - **gmtime / localtime / time**
  - **scalar**
  - **system and `` (to be prefered to exec)**
- And all mathematics command …

Best Practices

January 2014

Best Practices – Use compiler automatic check

- When you know that you program will grows-up, don't forget to use:
  - **use warnings;**

    This pragma helps you to detect incorrect use (as variables that are used only one time, declaration that hides other declaration, use of undefined value…)

  - **use strict;**

    This pragma allows
    - a check on global variable management (all used variable must be declared before)
    - A check on reference, (you can't use symbolic reference, only real created reference
    - A check on raw strings that must be encapsulated by double quotes.

Best Practices – Common Errors

- Using a comma (,) behind a file handle in print feature:
- Use == instead or eq or != instead of ne
- Forget to record $1, $2…from regular expression
- Don’t forget that sub-program parameters in @ are the real value (reference). Think to create temporary variable
- Write @truc[0] instead of $truc[0]
- Forget parentheses with a list operator, as my

Best Practices – Advices

- Be careful to operation that can have a different comportment in function of context
- <HF> is not a file handle, it's a feature to do an operation line by line from a File Handle
- Read data from <> operator are stored into $_ only if the read file is the only condition of a while loop
- Don't use = instead of =~
- Use my to declare when you can. local create a temporary global variable!!
- next and last don't work in do {} while syntax
- Use the /x extension with RE, to format it

Best Practices – Coding Rules 1/2

- The block closing accolade must be on the same column of the associated keyword which starts the block
- A short block can be written on one line, including accolade (in this case, forget the final semi colon)
- It’s better to frame operator with space
- Insert empty line between to different code block
- No space between function name and associated parentheses
- No space before a semi colon
- Space after comma
- Divide long line after operator
- Line-up corresponding items
- Prefer using parentheses, event if you can omit them
- Don’t be afraid using loop label, in order to create loop break on several level
- $ALL_CAPITAL: constant
- $Few_Capital: for global variable
- $no_capital: for my or local variable

Best Practices – Coding Rules 2/2

- All error messages should be printed in STDERR (use die), specifying program, function call; argument, and most important, associated system error message (if it's a system call which failed)
- Use the pronoum $_, when you can: `for (@line){ $_ .= \n }`
- Use for to set the $_ pronoum:

```
for ($episode){
   s/fred/barney/g;
   s/wilma/betty/g;
   s/pebbles/bambam/g;
}
```

- Use || to set default values:

```
sub bark{
   my $dog = shift;
   my $quality= shift || "barking";
   my $quantity = shift || "all_time";
   …
}
```

- Do substitution or transduction "en passant":

```
($new = $old) =~ s/bad/good/g;
for (@new = @old) =~ s/bad/good/g;
```

- Use negative index to access to end of array

Samples

And Usual Technics

January 2014

Samples - Get Module Version

- From specified list_mod file, extract specified module version, and set the ‘VERSION_TAG’ environment variable with detected version
- An option create a SetVers.bat file that can be used to set the environment variable
- Program must return:
  - **0 if version was not found**
  - **1 if version was found**
  - **-1 an error occurred**

One Solution

Samples - Patch Calib

- To test memory optimization Ebx need to associate the volatile attribute to all its automatic generated calibration data.
  - Create a script that update all calibration data in C75 and H files of a specified folder

File to Update

One Solution

Samples - File Renamer for CdRom Process

- Rename all folder documents reference name to short reference name (recursively)
  - **Document can be Word (.doc) or Acrobat (.pdf) files**
  - **Managed original reference names are:**
    - **V0X NT XX XXXXX [X[X[X]]]**
    - **PON EEM NT XX XXXX [X[X[X]]]**
    - **PON UXX NT XX XXXX [X[X[X]]]**
  - **Item separator can be a space, a dot(.) or an underscore(_)**
  - **New reference name will be NTXXXXXX[X]**

Solution

Samples - Command Line Manager

- 1. Create a case sensitive command line manager:
  - **Using argument as:** `-v Arg [-D ARG] -o ARG [–x]`
- 2. Create a non case sensitive command line manager:
  - **Using argument as:** `-Folder:Arg -Action:[Add|Remove] [-Verbose]`
- 3. Create a non case sensitive command line manager:
  - **Using argument as:** `Action1 /Act1Param [/Act1Opt:ARG] Action2 /Act2Param:ARG [/Act2Option]`


Exercice 1


Exercice 2


Exercice 3

To Continue…

Object-Oriented programming and Database management

January 2014

To Continue… – Object-Oriented Programing 1/2

- “Object” is a ‘reference’, a “method” and ‘procedure’, and a “class”, a ‘package’.
  - We associate to an object, data and methods, to make easier use of this object
- Create a perl file (with .pm extension), containing object to define
  - Two particular methods:
    - ***new***, constructor automatically called during object creation
    - ***DESTROY***, destructor automatically called during object destruction
- *bless* function allow to make object is visible from outside.

To Continue… – Object-Oriented Programing 2/2

```
#!/bin/perl
use strict;
package myclasse;   # object name
sub new {           # Constructor
my ($classe, ...) = @_;                 # Class is always the first parameter
...
return bless référence, $class;         # Variable is not returned, it's its reference!
}
sub DESTROY {       # destructor
my ($objet) = @_;                       # Object is always the first parameter
...
}
sub method1 {       # To define other methods
my ($objet, ...) = @_;                  # Object is always the first parameter
...
}
```

```
use myclass;
my $my_object = new myclasse(parameters); # constructor call

$my_object-> method1(parameters);          # method call
exit;                                      # End of program, object destructor call
```

To Continue… – Database Management

- Perl is able to integrate SQL request

- From Perl 5, we access to any database using a unique methodology: using DBI (DataBase Interface) module

```
#!/bin/perl
use DBI;              # To specify that we want access to database

my $dbh;

# connexion to 'test' mysql database
$dbh = DBI->connect("dbi:mysql:test", 'login', 'pwd');

$dbh->disconnect;    # To disconnect from database

$dbh->do("request") or die "Request problem : $DBI::errstr";        # To execute a SQL request
```

- To execute a request:
  - Request preparation, using *prepare*
  - Execution, using *execute*
  - Browse each returned line in a loop,using *fetchrow_array*
  - End of request, using *finish*

Automotive technology, naturally