PERL

Allan
Eduardo
Renan
Worlen

O que é PERL?

"Perl is a language for getting your job done"

Larry Wall

Introdução (1)

- **PERL** (***P****ractical* ***E****xtraction and* ***R****eport* ***L****anguage*) é uma linguagem de programação criada por *Larry Wall* em 1987 e que pode ser usada em uma grande variedade de tarefas :
  - Processamento de texto;
  - Desenvolvimento *Web*;
  - Administração de sistemas;
  - Acesso a Banco de Dados;

Introdução (2)

- **Características da PERL:**
  - *Linguagem simples;*
  - Suporte a várias plataformas (Unix, Mac OS, Windows, ...);
  - Modular e Orientadas a Objetos;
  - Desenvolvimento rápido;
  - Multiparadigma;

Desenvolvimento Rápido

- Muitos projetos de programa são de alto nível em vez de baixo nível. Isso significa que eles tendem a não envolver manipulações em nível de bits, chamadas diretas ao sistema operacional. Em vez disso, eles focalizam a leitura de arquivos, a reformatação de sísas e gravação em saída padrão. A Perl é vigorosa; um pequeno código Perl faz muito. Em termos de linguagem de programação, isso geralmente significa que o código será difícil de ler e penoso de escrever.

Compilada ou Interpretada?

- Bem, a Perl é um pouco especial a esse respeito. Ela é um compilador que pensa ser um interpretador. A Perl compila código de programa em código executável antes de executá-lo, portanto existe um estágio de otimização e o código executável trabalha mais rapidamente. Entretanto ela não grava esse código em um arquivo executável separado. Em vez disso, armazena-o na memória e, depois, executa-o.

  Isso significa que a Perl combina o ciclo de desenvolvimento rápido de uma linguagem interpretada com a execução eficiente de um código compilado. No entanto, as desvantagens correspondentes também estão lá: a necessidade de compilar o programa cada vez que ele é executado significa uma inicialização mais lenta do que uma linguagem puramente compilada e requer que os desenvolvedores distribuam o código-fonte para os usuários.

  Na prática, essa desvantagem não são muito limitadoras. A fase de compilação é extremamente rápida, portanto talvez, você nem note alguma demora entre a ativação do script Perl e o início da execução.

  Para finalizar, a Perl é compilada "nos bastidores" para uma execução rápida, mas você pode tratá-la como se fosse uma linguagem interpretada.

Primeiro Programa(2)

prog.pl

```
1  #!/usr/local/bin/perl
2
3  $x = 10;
4  $y = "chove";
5
6  print "$y sem parar há $x horas!\n";
7  #chove sem parar há 10 horas!
8
9  print '$y sem parar há $x horas!\n';
10 #$y sem parar há $x horas!\n
```

$ perl prog.pl

Variáveis (1)

No Perl 5 existem basicamente 3 estruturas. Escalares, arrays e hashes. A última também é conhecida em outras linguagens como dicionário, tabela de consulta ou array associativo.

Variáveis em Perl são sempre precedidas com um sinal chamado **selo**. Esse sinal é $ para escalares, @ para arrays, e % para hashes.

Variáveis (2)

Se declaramos uma variável, mas ainda não lhe atribuímos nada, então ela tem um valor chamado undef que é semelhante ao **NULL** do C.

Variáveis (3)

Escalares:

- $dna = “ATGCTTATTGTT”;
- $hits = 5;
- $evalue = 5e-120;
- $directory = `pwd`;

Primeiro Programa(1)

ola.pl

```perl
use 5.010;       #Declarando a versão mínima do Perl que o codigo exige
use strict;      #Ajuda na captura de erros e enganos comuns no codigo
use warnings;    # e em alguns casos prevenir que os realize. - compiler flags

say "Olá Mundo";
```

$ perl prog.pl

Variáveis (4)

Array:

```
@genes = ('BRCA1', 'NAT2' , 'MMP9' , 'MYC');
@cromossomos = (1,3,5,7,9,11,13,15,17,21,'X','Y');
```

- Os valores são acessados pelos seus índices:

```
print "$gene[0]";              # imprime BRCA1;
print "$cromossomos[3]";  # imprime 7;
```

- Tamanho do array

```
print "$#gene";                # imprime 3;
```

Variáveis (4)

Funções Úteis em Array:

- push @a, $a1, $a2; #...

adiciona os escalares dados ($a1,$a2...) no fim do array @a

- pop @a;

elimina o último escalar de @a, e retorna-o

- unshift @a, $a1, $a2; #...

como push, mas coloca os novos elementos no inicio de @a

- shift @a;

elimina o primeiro elemento de @a e retorna-o

- @a = reverse @a;

retorna o conteudo de @a revertido

- @a = sort @a;

retorna o conteudo de @a ordenado

Variáveis (5)

Hash:

```
%codons = (  'ATG'=>'M',
             'CTT'=>'L',
             'ATT'=>'I',
          );
```

- Acessando o valor de uma chave do hash
  print “$codons{'CTT'}\n”;   # imprime: L

Variáveis (5)

Funções Úteis em Hash:

- @a = keys %a;

devolve todas as chaves existentes em %a

- @b = values %a;

devolve todos os valores existentes em %a

- delete $a{$MY_KEY}

apaga $MY_KEY da hash %a

- each %a

devolve todos os pares KEY,VALUE iterativamente

Variáveis (6)

- Variáveis em Perl são globais por *default*
- *Não é necessário declarar variáveis, mas é possivel fazê-lo utililizando o operador my*
- *Variável introduzida com o my só é visível no escopo em que ela foi definida*

```
use warnings;
use strict;

my $var = 10;
```

Operadores (1)

A Perl usa todos os operadores aritméticos do C:

```
$a = 1 + 2;    # Soma 1 em 2 e armazena o resultado em $a
$a = 3 - 4;    # Subtrai 4 de 3 e armazena o resultado em $a
$a = 5 * 6;                 # Multiplica 5 por 6
$a = 7 / 8;                 # Divide 7 por 8 e obtém 0,875
$a = 9 ** 10;               # Eleva nove a décima potência
$a = 5 % 2;   # Armazena em $a o resto da divisão de 5 por 2
++$a;            # Incrementa $a e depois retorna
$a++;       # Retorna $a e depois incrementa
--$a;       # Decrementa $a e depois retorna
$a--;       # Retorna $a e depois decrementa
```

Operadores (2)

Para strings, estas são algumas das maneiras:

```
$a = $b . $c;     # Concatena $b com $c
$a = $b x $c;     # $b é repetida $c vezes
```

Para designar valores temos as seguintes formas:

```
$a = $b;     # Coloca em $a o conteúdo de $b
$a += $b;  # Soma o valor de $b ao valor de $a
$a -= $b;   # Subtrai o valor de $b do valor de $a
$a .= $b;   # Concatena $b a $a
```

Note que quando a Perl designa um valor com `$a = $b` ela faz uma cópia de `$b` e depois coloca em `$a`. Se depois você alterar o conteúdo de `$b` isto não altera `$a`.

Operadores (3)

- De comparação

comparação	númericas	em strings
igual	==	eq
diferente	!=	ne
menor	<	lt
maior	>	gt
menor or igual	<=	le
maior ou igual	>=	ge
comparação	<=>	cmp

Operadores (3)

- Em perl, falso (F) é qualquer expressão que tome o valor zero, e verdade (V) é qualquer expressão que tome um valor diferente de zero
- Todos os operadores de comparação, exceto <=> e cmp, são expressões que tomam o valor 1 em caso de verdade, e 0 em caso de falsidade
- Os operadores <=> e cmp, fazem uma comparação entre os dois operandos, e tomam o valor -1, 0 ou 1, conforme o primeiro operando seja menor, igual ou maior, que o segundo, respectivamente

Operadores (3)

Operador	**Tipo**
+ , - , * , / , % , ** , ++ , --	Aritmético
< , <= , == , >= , > , !=	Comparação Numérica
&& , || , ! , and , or , not	Lógicos
lt , gt , le , ge , eq , ne	Comparação de *String*
	Atribuição

RESPONDA RÁPIDO !

- Que número é esse?

  10000000000

- E agora, ficou melhor?

    10_000_000_000

Em Perl, é possível separar os números pelas casas dos milhares com o “_”.

Instruções if e unless

- ***If*** : A declaração será executada se a expressão lógica for verdadeira:

```
my $varA = 3;
my $varB = 5;
if ( $varA < $varB ) {
                    ....
}
```

- ***unless***: A declaração não será executada se a expressão for verdadeira:

```
my $varA = ‘A’;
unless ( $varA eq 'B' ) {
                    ....
}
```

Instruções Interativas

@genes = ('BRCA1', 'NAT2' , 'MMP9' , 'MYC','FOX2');

while *(expr) bloco*

```
my $i = 0;
while (  $i <= $#genes ){

        print “$i : $genes[$i]\n”;
        $i++;
}
```

foreach *var (lista) bloco*

```
foreach my $gene ( @genes ) {

        print “$gene\n”;
}
```

for*(expr;expr;expr) bloco*

```
for ( my $i =0 ; $i <= $#genes ; $i++ ) {

                print “$i : $genes[$i]\n”;
}
```

Formas especiais da instrução *do*

@genes = ('BRCA1', 'NAT2' , 'MMP9' , 'MYC', 'FOX2');

do *bloco* ***while*** *expr*;

```
my $i = 0;
do{
        print "$genes[$i]\n";
        $i++;
 }while ( $i <= $#genes );
```

do *bloco* ***until*** *expr*;

```
my $i = 0;
do{
        print "$genes[$i]\n";
        $i++;
 }until ( $i > $#genes );
```

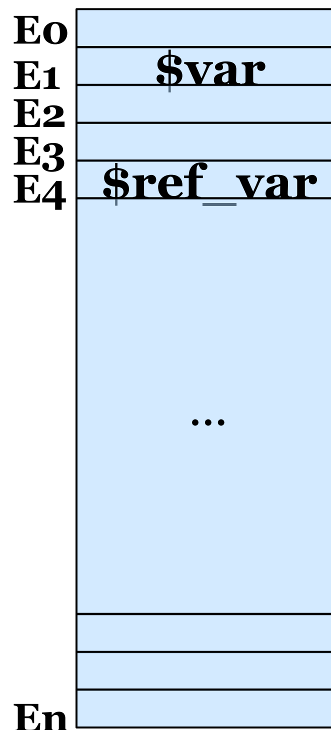
Referências (1)

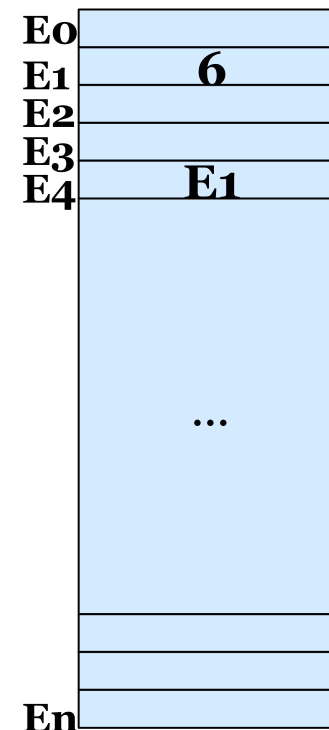
Referências são ponteiros para tipos de dados previamente definidos:

```
my $var = 6;
my $ref_var = \$var;
```

```
print $ref_var;  # imprime E1

print $$ref_var; # imprime 6
```





Referências (2)

Referência	Definição	Valor
Escalar	$ref_var = \$var;	$$ref_var
Array	$ref_array = \@genes;	$ref_array->[0]
Hash	$ref_hash = \%hash;	$ref_hash->{'ATG'}

Gerenciamento de Memória

- Perl garante que um pedaço de informação que não possui mais referências (não possui mais nenhum 'ponteiro') é deletado, assim ele impede *memory leaks*, e também garante que enquanto há referências na memória a uma informação, ela não é deletada (Ponteiros apontando para o nada)
- Coletor de Lixo:

  - Contador de referência

  - Marcar e Varrer (Extensivo, ao finalizar o programa)

Expressões Regulares (1)

- "Expressões Regulares", em inglês "Regular Expressions", e apelidado de REGEXP, é uma das ferramentas mais úteis em programação, e utilizada em Perl a muito tempo. Recentemente várias outras linguagens vêm introduzindo tal recurso, sendo a mais recente Java. Ou seja, em Perl REGEXP existe a mais de 10 anos, mas em java a apenas 2 anos, o que demonstra como REGEXP é um recurso importante, poderoso bem desenvolvido em Perl.
- Com REGEXP pode-se checar o formato de uma string, formata-la, substituir dados, capturar dados, ou até mesmo criar um parser.

Expressões Regulares (2)

Uma das maneiras mais simples de utilizar REGEXP é checando um formato:

```
my $var = "Contenho fulano no texto." ;
if ( $var =~ /fulano/ ) {
        print "A variável possui 'fulano'.\n" ;
}
```

Obs: =~(pertence), !~(não pertence);

Expressões Regulares (3)

Parâmetros:

Agora que sabe-se a declaração, parâmetros de controle

```
my $var = "Contenho FuLaNo no texto." ;
if ( $var =~ /fulano/i ) {
        print "A variável possui 'fulano'.\n" ;
}
```

Note o 'i' no fim do REGEXP, indicando “Not Case Sensitive" (não diferenciar maiúsculas e minúsculas).

Expressões Regulares (3)

Os demais parâmetros existentes:

g => global: todas as ocorrências.

s => single line: não parar em \n.

x => extend spaces. Extende/ignora espaço no seu REGEXP, permitindo melhor organização.

e => execute. Executa comandos quando usado em replace.

m => multiline: inverso de 's'.

http://sao-paulo.pm.org/artigo/2006/RegexPraticasTecnicasAvancadas

Sub-rotinas (1)

A sub-rotina agrupa uma sequência de declarações e funções que poderão ser reutilizadas pelo programa.

```
#!/usr/bin/perl

use strict;
use warnings;

my ($x,$y,$sum) = (5,9,0);

$sum= &sum ($x,$y);

print "$x + $x = $sum\n";

sub sum{

        my ($var1,$var2) = @_;

        my $sum =  $var1 + $var2;

        return $sum;
}
```

```
#!/usr/bin/perl

use strict;
use warnings;

my ($x,$y,$sum) = (5,9,0);

&sum ($x,$y,\$sum);

print "$x + $x = $sum\n";

sub sum{

        my ($var1,$var2,$rS_sum) = @_;

        $$rS_sum =  $var1 + $var2;

        return;
}
```

Sub-rotinas (2)

Passagem de parâmetros:

- Os parâmetros reais passados implicitamente para a sub-rotina, sempre por referência, são colocados no vetor local da função @_.
- Os parâmetros passados na linha de comando para o programa principal são colocados no vetor @*ARGV*.
- Caso não tenha ***return***, a sub-rotina retorna o valor da última expressão avaliada.

Manipulação de Arquivos (1)

```
1.  use strict;
2.  use warnings;
3.
4.  my $arquivo = 'report.txt';
5.  open(my $fh, '>', $arquivo) or die "Não foi possível abrir o arquivo '$arquivo' $!";
6.  print $fh "Meu primeiro relatório gerado pelo Perl\n";
7.  close $fh;
8.  print "pronto\n";
```

- A função **open** necessita de 3 parâmetros:
  O primeiro, $fh é uma variável escalar. O segundo parâmetro define a forma como o arquivo será aberto. Neste caso, utilizou-se o sinal de *maior que* (>) significando que o arquivo será aberto para escrita e *menor que* (<) seria de leitura. O terceiro parâmetro é o caminho do arquivo que será aberto.

Manipulação de Arquivos (2)

```
1. my $arquivo = 'caminho_correto_com_erro_de_digitação/report.txt'';
2. open(my $fh, '>', $arquivo) or die "Não foi possível abrir o '$arquivo' $!";
```

- $! - uma variável interna do Perl para imprimir a mensagem de erro proveniente do sistema operacional sobre a falha.
- die é a função que irá lançar uma exceção e então terminar o script.

```
Não foi possível abrir o arquivo 'caminho_correto_com_erro_de_digitação/report.txt' No such file or directory
...
```

Manipulação de Arquivos (3)

Lendo linhas:

```
use strict;
use warnings;

my $filename = $0;
open(my $fh, '<:encoding(UTF-8)', $filename)
  or die "Não foi possível abrir o arquivo '$filename' $!";

while (my $row = <$fh>) {
  chomp $row;
  print "$row\n";
}
print "pronto\n";
```

Toda vez que atingimos a condicional do laço while, primeiro será executado my $row = <$fh> *(readline),* a parte que irá ler a próxima linha do arquivo. Se existir alguma coisa nessa linha, a condicional será avaliada como VERDADEIRA. Até mesmo linhas vazias possuem caracteres especiais de nova linha, isso significa que ao ler essas linhas a variável $row irá conter o \n que irá ser avaliado como VERDADEIRO em contexto booleano.

Após ler a última, na próxima iteração do laço o operador *readline* (<$fh>) irá retornar undef que então é avaliado como FALSO. Sendo assim o laço while termina.

Tipos de Polimorfismo (1)

Coerção:

- Perl suporta coerção, por exemplo:

```
$var = @lista;            # $var pega o tamanho  do $var = @lista
$var = (@lista1, @lista2);      # $var pega o tamanho das 2 listas
$var = @lista1, @lista2;        # $var pega o tamanho da lista2
@lista = $var ;                 # há uma conversão implícita:
                                # $var  é entendido como $var[0]
                                # que seria @var com apenas 1
                                # elemento
```

Tipos de Polimorfismo (2)

Sobrecarga:

- Perl suporta sobrecarga, sendo que o tipo mais simples é o de verificação de quantidade de parâmetros de entrada, no @_

- Há também sobrecarga de operadores de uma classe, com o *overload*; utilizando ele, é possível determinar operações específicas para objetos de uma mesma classe.

Tipos de Polimorfismo (3)

Inclusão:

- Possui polimorfismo de inclusão, por exemplo, herança entre pacotes, quando usados como classes.
- Exemplo na parte de OO.

Serialização

- Existem 3 módulos que Perl oferece para serialização de objetos, Storable, FreezeThaw e Data::Dumper.
- Todos os 3 módulos produzem dados armazenados num escalar como saída, que podem ser escritos num arquivo ou armazenados num banco de dados qualquer. Também podem ser enviados a outro processo via pipe, socket ou conexão HTTP.
- Decodificar do outro lado também é simples.

Como Programar Orientado a Objetos (POO) (1)

- Em Perl, as classes são criadas através de *packages*, que é usada mais usualmente para criar módulos tradicionais.
  As classes são criadas como referencias para pacotes, são referencias 'abençoadas'.
  Depois de passar essas referências para a função ***bless***, é possível chamar funções por essas referências, ou seja, métodos.

- A função ***bless*** é responsável por dizer que uma certa referência pertence a um pacote.

Como Programar Orientado a Objetos (POO) (2)

```
# Arquivo da classe Pessoa, Pessoa.pm
package Pessoa;

use strict;
use warnings;

sub new{                                        # esse é o construtor da classe
        my $class = shift;                      # classe onde a referência será
                                                # abençoada
        my $self ={};
        $self->{NOME} = undef;                  # a referencia a ser abençoada,
        $self->{IDADE} = undef;                 # um hash nesse exemplo

        bless($self, $class);          # agora os métodos de classe podem ser
                                       # chamados pela referencia

        return $self;
}
```

Como Programar Orientado a Objetos (POO) (3)

```
# MÉTODOS DE ENCAPSULAMENTO DOS ATRIBUTOS; ainda no arquivo Pessoa.pm
sub nome{
		my $self = shift;
		$self->{NOME} = shift if @_;
		return $self->{NOME};
}

sub idade{
		my $self = shift;
		$self->{IDADE} = shift if @_;
		return $self->{IDADE};
}
# um metodo de exemplo
sub fale{
		my $self = shift;
		my $fala = shift II "Qualquer coisa";
		return $fala;
}

1;				#A classe pessoa precisa retornar um valor verdadeiro
```

Como Programar Orientado a Objetos (POO) (4)

```
# Arquivo PessoaMuda.pm
package PessoaMuda;
use Pessoa;              # "herdando" Pessoa

# Aqui estamos dizendo que PessoaMuda é uma Pessoa
@ISA = (Pessoa);

# Sobrescrevendo o método fale pra que retorne nada, afinal, é uma pessoa Muda.
sub fale{
        my $self = shift;
        return
}
1;
```

Como Programar Orientado a Objetos (POO) (5)

```
#!/usr/bin/env perl
# a linha acima é o shebang ou hashbang, que indica onde o compilador Perl está, e
# pode variar de sistema para sistema
# inicio da 'main'
use strict;
use warnings;

# importando as classes Pessoa e PessoaMuda
use Pessoa;
use PessoaMuda;

#Criando uma nova pessoa
my $p = Pessoa->new();
$p->nome('Joao');
print $p->nome();                              # Joao

print $p->fale()."\n";                         # Qualquer coisa
print $p->fale('Cade Maria?') . "\n";          # Cade Maria?
```

Como Programar Orientado a Objetos (POO) (6)

```
# Uma pessoa muda

my $pm = PessoaMuda->new();
$pm->nome(‘Red');
print $pm->nome().”\n”;                    # Red

$pm->idade(10000);
print $pm->idade();             # 10000

print $pm->fale('Alguma coisa');
# não imprime nada, por conta do método
# que foi sobrescrito
```

Tratamento de Exceções (1)

- O tratamento de exceções em Perl é feito usando a classe Error.pm
- Uma exceção geralmente carrega 3 informações importantes:
  - O tipo da exceção (determinado pela classe do objeto de exceção);
  - Onde a exceção ocorreu (na pilha);
  - O contexto de informação: mensagem de erro e outras informações do estado.

Tratamento de Exceções (2)

```
sub func1{
        try {
                func2();
        } catch IOException with {
                # Código de tratamento de exceção aqui
                print ‘deu erro’.”\n”;
        };
sub func2{ func3(); … }
sub func3{ processFile($FILE); … }

sub processFile{
        my $file = shift;
        if (!open(FH, $file)) {
                throw IOException (“Erro ao abrir arquivo <$file> - $!”) ;
        } else {
                # processa o arquivo
                return 1;
        }
}
```

Moose (1)

- Moose é uma extensão para orientação a objetos da linguagem de programação Perl.
- Ele traz recursos de linguagem orientada a objetos modernos para Perl 5, tornando a programação Perl orientado a objeto mais consistente e menos tedioso.

Moose (2)

- Moose é construído em cima de Class :: MOP
  (Meta Object Protocol – Protocolo de Meta-Objeto)
- Usando o MOP, Moose fornece introspecção completa para todas as classes Moose.

Metaclasses

- Perl tem metaclasse via *metaclass pragma e utilizando o Moose*.
- Moose::Meta::Class - The Moose metaclass

```
my $metaclass = Moose::Meta::Class->create( 'New::Class', roles => [...] );

my $metaclass = Moose::Meta::Class->create_anon_class(
    superclasses => ['Foo'],
    roles        => [qw/Some Roles Go Here/],
    cache        => 1,
);
```

Classes

- MOOSE permite que um programador ao criar classes:

  - A classe tem zero ou mais atributos.
  - A classe tem zero ou mais métodos.
  - A classe tem zero ou mais superclasses . A classe herda de sua superclasse(s).
  - Moose suporta herança múltipla.

Classes

- A classe tem zero ou mais métodos modificadores.

  Esses modificadores se aplicam a:

  - Seus próprios métodos
  - Métodos que são herdados de seus ancestrais
  - Métodos que são fornecidos pelos papéis.

- A classe faz de zero ou mais funções (também conhecidos como traits em outras linguagens de programação).
- A classe tem um construtor e um destrutor.
- A classe tem uma metaclasse.

Atributos

- Um atributo é uma propriedade da classe que o define.
- Um atributo sempre tem um nome, e pode ter outras características que o definem.
- Características de um atributo podem incluir uma flag de leitura / gravação, um tipo, os nomes de método de acesso e um valor padrão.

Funções (1)

- Funções em Moose são baseados em características. Eles realizam uma tarefa semelhante à mixins(classe que contém combinações de métodos de outra classe), mas são compostas horizontalmente em vez de herdada.
- Eles também são um pouco como as interfaces, mas ao contrário de interfaces, funções podem fornecer uma implementação padrão.

Funções (2)

- As funções podem ser aplicadas a casos individuais, bem como Classes.

  - A função tem zero ou mais atributos.

  - A função tem zero ou mais métodos.

  - A função tem zero ou mais métodos modificadores.

  - A função tem zero ou mais métodos necessários.

Extensões

- Há uma série de módulos de extensão de Moose no CPAN(Comprehensive Perl Archive Network (Rede de Repositórios Perl) que é um repositório de mais de 18.200 módulos de software escritos em linguagem de programação Perl, assim como suas respectivas documentações).
- Há 855 módulos em 266 distribuições no namespace MooseX. (Setembro de 2012)
- A maioria deles pode ser opcionalmente instalado com o Task :: módulo Moose.

Exemplo de Moose:

- classe *Point*
- subclasse *Point3D*

```
package Point;
use Moose;
use Carp;

has 'x' => (isa => 'Num', is => 'rw');
has 'y' => (isa => 'Num', is => 'rw');

sub clear {
    my $self = shift;
    $self->x(0);
    $self->y(0);
}

sub set_to {
    @_ == 3 or croak "Bad number of arguments";
    my $self = shift;
    my ($x, $y) = @_;
    $self->x($x);
    $self->y($y);
}

package Point3D;
use Moose;
use Carp;

extends 'Point';

has 'z' => (isa => 'Num', is => 'rw');

after 'clear' => sub {
    my $self = shift;
    $self->z(0);
};

sub set_to {
    @_ == 4 or croak "Bad number of arguments";
    my $self = shift;
    my ($x, $y, $z) = @_;
    $self->x($x);
    $self->y($y);
    $self->z($z);
}
```

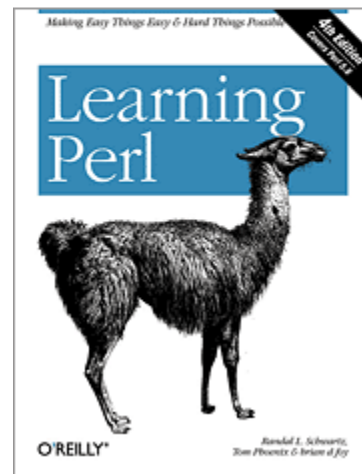
Avaliação da Linguagem

http://geekandpoke.typepad.com/geekandpoke/perl/

Bibliografia

Perl : Guia de Consulta Rápida
Decio Jr. - Ed. Novatec

Learning Perl (4th ed)
by Randal L. Schwartz, Tom Phoenix, brian d foy. Ed. O'Reilly



Programming Perl (2nd Edition)
by Larry Wall, Tom Christiansen, Randal L. Schwartz, Stephen Potter Ed. O'Reilly

Advanced Perl Programming (2th ed)
by Simon Cozens. Ed. O'Reilly

- http://www-cgi.cs.cmu.edu/cgi-bin/perl-man
- http://www.perl.org/docs.html
- http://www.perlmonks.org/