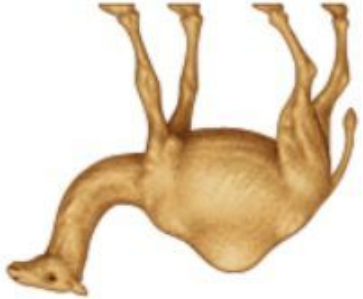
Beatriz Ogioni
Lucio Sandrini
Heitor Schulz
Tiago Silva
Matheus Hemerly

PERL



Antes de seu lançamento, **todo o processamento de texto** em sistemas baseados em Unix era feito com uma porção de ferramentas!

A idéia de seu criador foi **juntar as principais vantagens** de todas essas linguagens: expressões regulares do 'sed'; a identificação de padrões de AWK; a profundidade de C; além da sintaxe baseada tanto em C quanto em Shell Script.

Histórico

“Tornar fáceis as coisas fáceis e possíveis as coisas difíceis”

- PERL é acrônimo de Practical Extraction and Report Language;
- Foi desenvolvida por Larry Wall, em 1984;
- Sua primeira versão era uma linguagem **intuitiva, facilmente codificada** para a digitalização, extração e impressão de informações de ficheiros de texto;
- **Linguagem de script** que poderia lidar com muitas tarefas de gestão de sistema;
- O código fonte se tornou **aberto e colaborativo** a partir de 1987.

Em 1992, ela estava em sua 4ª versão e se tornou uma **linguagem padrão para Unix**.

Então, **suas limitações começaram a aparecer**:

Ela era **ótima** para **códigos pequenos e poderosos.** → Mas **péssima** para **programas maiores**!

PERL 5 **foi uma grande reorganização** que consertou muitas das limitações da linguagem, tornando-a mais poderosa, mais legível e mais extensível.

De uma **ferramenta de processamento de textos**.

Linguagem de programação de **propósito geral** com seu **próprio ambiente de desenvolvimento de software completo**

PERL 6 **foi uma tentativa de reestruturar a linguagem** para expandir ainda mais seus horizontes com adições interessantes como o paradigma de orientação a objetos. Tentativa que **não teve tanto sucesso**, e estas são algumas das explicações encontradas:

- Desenvolvimento lento
- Falta de adesão
- Surgimento de poderosas linguagens de script (ex.: Python e Ruby)
- Críticas quanto à performance

Comunidade

Um dos sites brasileiro de maior relevância da comunidade é o São Paulo Perl Mongers, porém ele não é atualizando desde 2015.

Existem **vários grupos de utilizadores online** que conectam os desenvolvedores e utilizadores de PERL.

Um detalhe muito importante é que **as pessoas fornecem, gratuitamente, o código-fonte** para seus programas.

Esse detalhe **facilita o aprendizado** ao PERL por meio de **exemplos** e você também pode fazer o **download** e **modificar milhares de scripts para seu próprio uso**.

Portabilidade

No ambiente atual do Linux e também Mac, o Perl é uma ferramenta padrão do Unix e verdadeiramente indispensável.

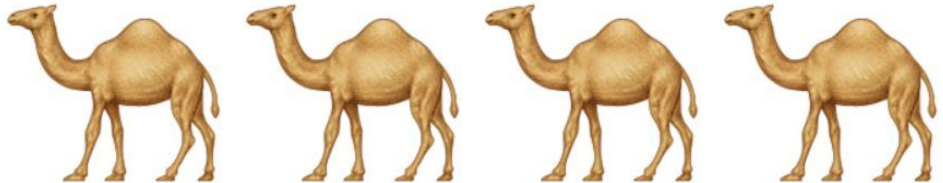
As tarefas podem ser significativamente mais curtas, mais fáceis ou mais extensíveis quando resolvidas com o Perl do que as ferramentas tradicionais, sendo que, muitas das de sistema, scripts e programas maiores são rotineiramente escritos em Perl.

- Perl é instalado por padrão nos sistemas operacionais Linux e Mac.
- No Windows é necessário fazer a instalação do interpretador Perl.

Aplicações

- Desenvolvimento WEB;
- Administração de sistemas;
- Acesso a banco de dados;
- Processamento de textos
- Ferramentas de sistema (principalmente Linux);
- Scripts.

Características Gerais



- Sintaxe Simples;
- Multiparadigma: Funcional, Estrutural e OO;
- Tipagem fraca e dinâmica;
- Case sensitive;
- Alto nível;
- Gerenciamento de memória interno da linguagem;
- Suporta Unicode;
- É possível "compilar" o código Perl para bytecode executável e para o código C.

Sintaxe

Bom saber:

Você pode ou não usar parênteses para argumentos de funções. Eles são necessários apenas quando precisa-se deixar clara a precedência. :D

Comandos são finalizados com ponto e vírgula:

```
print ("hello world");
```

Espaços em branco são irrelevantes, exceto dentro de strings com aspas:

```
#isso faz com que o texto seja impresso com uma quebra de linha
print "hello
 world";
```

Aspas simples e Aspas duplas podem ser usadas:

```
print "hello world \n";     # hello world
print 'hello world \n';      # hello world \n
```

Sintaxe

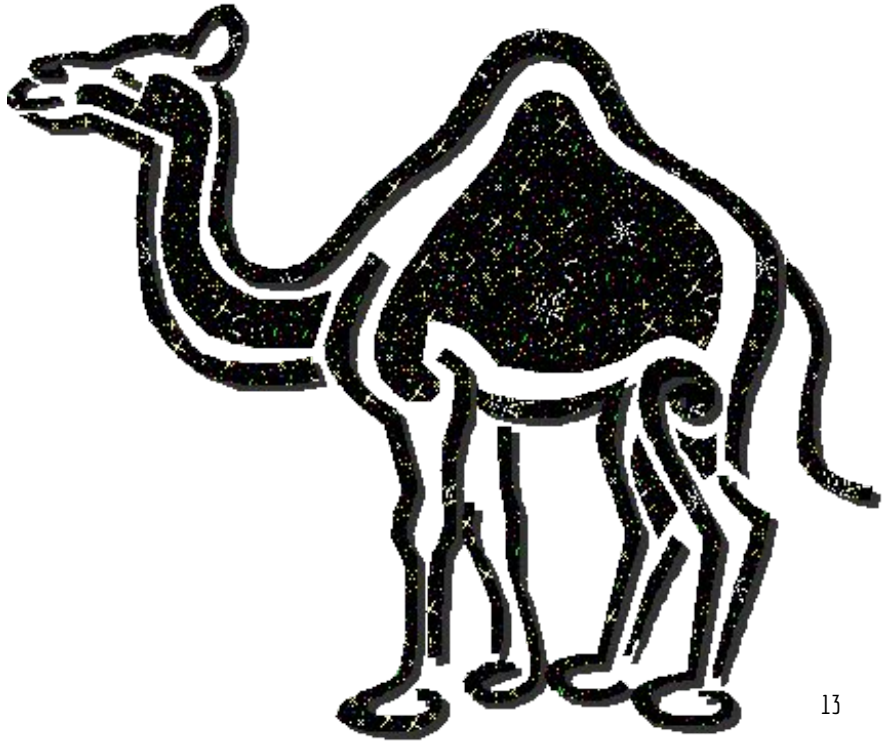
A primeira linha do programa deve começar com a seguinte **shebang:**

`#!/usr/bin/perl`	Isso indica o caminho para o compilador-interpretador.

Para maior segurança, adicione:

`use strict;` `use warnings;`	Strict fará com que seu programa seja imediatamente interrompido caso encontre erros durante a compilação

Tipos de Dados



Tipos de dados

- Tipagem Fraca e Dinâmica;
- Case sensitive;
- Não possui booleanos;
- Existem dois tipos de dados:

- **Strings:** podem ser, ou não, interpoláveis;
- **Numéricos:** Podem ser inteiros ou ponto flutuante de precisão dupla.

Null : - 0
- *undef*
- string ""
- string "0"

Exemplos:

```perl
#!/usr/bin/perl

$str = "PERL";
$inteiro = 13;
$naoInteiro = 1.3;
$negativo = -10;
$octal = 1313;
$hexa = 0xbb;
$scinote = 12.31E4;
```

Variáveis

Perl tem três tipos principais de variáveis: scalars ($), arrays (@) e hashes (%).

Scalars:

- Variável simples;
- Suporta inteiro, float e string;

Entrada:

```
#!/usr/bin/perl

$str = "PERL";
$inteiro = 13;
$naoInteiro = 1.3;

print "Eu sou $str, tenho $inteiro anos
    e $naoInteiro reais\n";

```

Saída:

```
Eu sou PERL, tenho 13 anos
    e 1.3 reais
```

String multi-linhas:

Entrada:

```
1  #!/usr/bin/perl
2
3  print << 'FIM'
4  Olá
5  Meu
6  Nome é
7  PEEEEERRRRRLLLLL
8  e eu printo esquisito
9  FIM
```

Saída:

```
Olá
Meu
Nome é
PEEEEERRRRRLLLLL
e eu printo esquisito
```

Variáveis

Perl tem três tipos principais de variáveis: scalars ($), arrays (@) e hashes (%).

Array:

- É uma lista de valores;
- Valores acessíveis através do índice;
- É permitido o uso de range operator (..) na criação de arrays;
- Para saber o tamanho do array é só o igualar a uma variável escalar;

Exemplo:

Entrada:

```perl
#!/usr/bin/perl

@array = ('camelo', 'lhama', 'alpaca');
$tam = @array;

print "eu sou $array[0] ou $array[1] ou $array[2]\n";
print "eu posso ser $tam coisas\n";
```

Saída:

```
eu sou camelo ou lhama ou alpaca
eu posso ser 3 coisas
```

Exemplo:

Entrada:

```perl
#!/usr/bin/perl

@array = (a..z);
$tam = @array;

print "@array\n";
print "meu tamanho é $tam\n";
```

Saída:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
meu tamanho é 26
```

Variáveis

Perl tem três tipos principais de variáveis: scalars ($), arrays (@) e hashes (%).

Funções de array:

- **Push:** Insere elemento no array;
- **Pop:** Retira e retorna o último elemento do array;
- **Shift:** Retira e retorna o primeiro elemento do array;
- **Unshift:** Insere um elemento no início do array;
- **Sort:** Retorna o array ordenado;
- **Split:** Transforma string em array a partir de um delimitador;
- **Join:** Transfoma array em string;
- **Reverse:** Retorna array invertido.

Exemplo de algumas funções:

Entrada:

```perl
#!/usr/bin/perl

@array = (a, c, z, n);

push (@array, b);
print "@array\n";

print (pop (@array));
print "\n";
print "@array\n";

print (shift (@array));
print "\n";
print "@array\n";

unshift (@array, b);
print "@array\n";

```

Saída:

```
a c z n b
b
a c z n
a
c z n
b c z n
```

Variáveis

Perl tem três tipos principais de variáveis: scalars ($), arrays (@) e hashes (%).

Hashes:

- Grupos não ordenados de pares chaves-valor;
- As chaves são strings únicas;
- Os valores podem ser de qualquer tipo ($, @ e %);
- Usa-se => para substituir a vírgula entre as chaves para uma leitura mais agradável;

```
#!/usr/bin/perl

%dinheiros = ('Bia' => 1000, 'Lucio' => 0, 'Heitor' => 50);
%dinheiros = ('Bia', 1000, 'Lucio', 0, 'Heitor', 50);
%dinheiros = (-Bia => 1000, -Lucio => 0, -Heitor => 50);
```

Exemplo:

Entrada:

```perl
#!/usr/bin/perl

%dinheiros = ('Bia' => 1000, 'Lucio' => 0, 'Heitor' => 50);
#%dinheiros = ('Bia', 1000, 'Lucio', 0, 'Heitor', 50);
#%dinheiros = (-Bia => 1000, -Lucio => 0, -Heitor => 50);

print %dinheiros;
print "\n";
print "$dinheiros{'Bia'}\n";
```

Saída:

```
Bia1000Lucio0Heitor50
1000
```

Variáveis

Perl tem três tipos principais de variáveis: scalars ($), arrays (@) e hashes (%).

Funções de hash:

- **Keys:** Retorna um array com as chaves;
- **Values:** Retorna um array com os valores;
- **Exists:** Verifica se uma chave existe.
- **Delete:** Remove um elemento da hash.

Exemplo de algumas funções:

Entrada:

```perl
#!/usr/bin/perl

%dinheiros = ('Bia' => 1000, 'Lucio' => 0, 'Heitor' => 50);

@keys = keys %dinheiros;
print "@keys\n";

@values = values %dinheiros;
print "@values\n";

if(exists $dinheiros{'Bia'}){
  delete $dinheiros{'Lucio'};
}

print %dinheiros;
print "\n";
```

Saída:

```
Lucio Bia Heitor
0 1000 50
Bia1000Heitor50
```

Array de arrays:

```perl
#!/usr/bin/perl

@arrayDeArray = (
                  ["OLA", "sou", "PERL"],
                  ["OLA", "sou", "Bia"],
                  ["ela", "é", "doidinha"],
);

print "$arrayDeArray[1][0], $arrayDeArray[0][2]
    é $arrayDeArray[2][2]\n";

```

```
OLA, PERL
    é doidinha
```

Hash de array:

```perl
#!/usr/bin/perl

%arrayDeArray = (
                  Frase1 => ["OLA", "sou", "PERL"],
                  Frase2 => ["OLA", "sou", "Bia"],
                  Frase3 => ["ela", "é", "doidinha"],
);

print "$arrayDeArray{Frase1}[2]: é isso ai\n";

```

```
PERL: é isso ai
```

Variáveis especiais

Algumas delas são:

- $_: Variável padrão;
- @_ : Array com argumentos para sub-rotinas;
- @ARGV: Array com argumentos passados por linha de comando;
- %ENV: Variáveis de ambiente.

```
$!                  $-                  $?
$"                  $`                  $DEBUGGING
$^S                 $.                  $@                  %ENV
$LIST_SEPARATOR     $a                  $[                  $^F
$#                  $PERL_VERSION       $\                  $^H
$^T                 $/                  $]                  %OVERLOAD
$$                  $ACCUMULATOR        $ERRNO              $^I
$%                  $0                  $^                  $^L
$&                  $ARG                $EUID               @+
$^UTF8LOCALE        $PID                $UID                $^M
$'                  $:                  $^A                 @-
$^V                 $ARGV               $WARNING            $^N
$(                  $;                  $^C                 @_
$^W                 $b                  $|                  $^O
$)                  $<                  $~                  @ARGV
$*                  $BASETIME           $^D                 $^OPEN
$+                  $PROCESS_ID         %!                  $^P
$^X                 $=                  $^E
$,                  $>                  %^H
$_                  $COMPILING          $^ENCODING
```

Fonte: *https://learn.perl.org/docs/keywords.html*

Referência

Um escalar pode armazenar um ponteiro para um escalar, array ou hash.

- Para referenciar algum tipo é preciso adicionar o caractere **** antecedente à referência;
- Para desreferenciar, basta adicionar o marcador do tipo referenciado(**$**, **@** ou **%**);

```
1  #!/usr/bin/perl
2
3  $a = "teste";
4  $b = \$a;
5
6  print "$a\n";
7  print "$b\n";
8  print "$$b\n";
```

```
teste
SCALAR(0x26a29c0)
teste
```

Variáveis anônimas:

Entrada:

```perl
#!/usr/bin/perl

$array = [
            {Bia => "3333-3333"},
            {Lucio => "6666-6666"},
            {Tiago => "9999-9999"}];

print $array . "\n";
print $array->[0] ."\n";
print $array->[0]->{Bia} . "\n";
```

Saída:

```
ARRAY(0x7ffcde02ebb8)
HASH(0x7ffcde002ee8)
3333-3333
```

Escopo de amarração

As variáveis por padrão tem escopo global;

```
1  #!/usr/bin/perl
2
3  $global = "O de fora :P\n";
4
5  if (1){
6      $local = $global = "O de dentro :D\n";
7  }
8
9  print $global . $local;
```

```
O de dentro :D
O de dentro :D
```

Escopo de amarração

Porém, tem como definir elas localmente utilizando o operador **my.**

```perl
#!/usr/bin/perl

$global = "O de fora :P\n";

if (1){
    my $global = "O de dentro :D\n";
    print $global;
}

print $global;
```

```
O de dentro :D
O de fora :P
```

Constantes

As constantes são declaradas com use constant =›.

Serve apenas para escalares ($) e referências.

```perl
#!/usr/bin/perl

use constant Dollar => 3.15;
$dinEmReais = 100000;

$dinEmDollars = $dinEmReais * Dollar;

print $dinEmDollars . "\n";
```

```
315000
```

ReadOnly

Utilizado por meio de um pacote específico: Readonly.

Pode ser usado em qualquer tipo de variável, e pode ser restrito ao escopo.

```
3  use Readonly;
4
5  Readonly my $a => 10;
6  if(1){
7      Readonly my @b => (10, 20, 30);
8      print "@b\n";
9  }
10 print "$a\n";
11 print "@b\n";
```

```
10 20 30
10
```

Operadores

Operadores aritméticos, de comparação, lógicos, atribuição, bit a bit entre outros.

Operadores Aritméticos

Operador	Descrição	Exemplo
+	Adição	$a + $b
-	Subtração	$a - $b
*	Multiplicação	$a * $b
/	Divisão	$a / $b
%	Resto de divisão	$a % $b
**	Exponenciação	$a ** $b
++	Incremento	$a++
--	Decremento	$a--

Operadores de Atribuição

Operador	Exemplo
=	$a = 10; #$a recebe 10.
+=	$a += $b; #Equivale a $a = $a + $b;
-=	$a -= $b; #Equivale a $a = $a - $b;
*=	$a *= $b; #Equivale a $a = $a * $b;
/=	$a /= $b; #Equivale a $a = $a / $b;
%=	$a %= $b; #Equivale a $a = $a % $b;
**=	$a **= $b; #Equivale a $a = $a ** $b;

Operadores de Lógicos e Bit-a-Bit

Operador	Descrição
&& , and	“E” Lógico.
|| , or	“OU” Lógico.
! , not	“NÃO” Lógico.

Importante!
Nos operadores AND e OR pode acontecer **curto-circuito!**

Operador	Descrição
&	“E” binário
|	“Ou” binário.
^	“Ou” exclusivo.
~	Complemento de 1.
<<	Shift à esquerda.
>>	Shift à direita.

Operadores de Comparação:

Operador	Descrição
== , eq	“Igual"
!= , ne	“Não Igual"
<=> , cmp	Comparação Relacional (-1, 0, 1)
< , lt	"Menor que"
> , gt	"Maior que"
<= , le	"Menor ou Igual a"
>=, ge	"Maior ou Igual a"
~~	"Comparação Inteligente"

Operador range (..) e ternário:

```
@a = (1..5);
print "@a".$/;
```

```
1 2 3 4 5
```

```
$a = 20;
$b = 10;

$c = ($a > $b) ? "Maior" : "Menor";
print $c.$/;
```

```
Maior
```

Condicionais

Perl possui a maioria das construções condicionais e de iteração, exceto pela construção case/switch - esse pode ser "adquirido" por meio de um módulo em Perl 5.8;

if:

```
$parametro = 0;

if ($parametro == 0){
    print "entrou no if! \n";
} elsif ($parametro == 1){
    print "entrou no elsif! \n";
} else {
    print "não entrou em nada :( \n";
}
```

unless:

```
$idade = 17;

print "Entrada ";
print "nao " unless ($idade >= 18);
print "permitida\n";
```

Iterativos

while:

```
$horaDeContar = 0;

while($horaDeContar <= 5){
  print "To no numero $horaDeContar\n";
  $horaDeContar++;
}

print "ACABEIIIII" . "\n";
```

```
To no numero 0
To no numero 1
To no numero 2
To no numero 3
To no numero 4
To no numero 5
ACABEIIIII
```

until:

```
$horaDeContar = 0;

until($horaDeContar >= 5){
  print "To no numero $horaDeContar\n";
  $horaDeContar++;
}

print "ACABEIIIII" . "\n";
```

```
To no numero 0
To no numero 1
To no numero 2
To no numero 3
To no numero 4
ACABEIIIII
```

Iterativos

until:

```
print "hora de contarrr!",$/;
$cont = 0;

print "$cont " until $cont++ > 5;
print "\nC'est fini\n";
```

```
hora de contarrr!
1 2 3 4 5 6
C'est fini
```

Iterativos

for:

```
for($i = 0; $i <= 10; $i++){
      print "Rodando muitas vezes" . "\n";
      print "To na vez $i\n";
}

print "PEEERRRLLL\n";
```

```
@array = (1, 2, 3, 4, 5, 6);
print "Agora vou imprimir o array" . "\n";

for(@array){
      print $_ ."\n";
}

print "PEEERRRLLL\n";
```

```
Agora vou imprimir o array
1
2
3
4
5
6
PEEERRRLLL
```

Iterativos

foreach:

```
foreach $a (1..5){
   print $a . " ";
}
```

do-while:

```
$a = 0;

do{
    print $a,$/;
    $a++;
} while ($a <= 5);
```

```
0
1
2
3
4
5
```

Operações de I/O

Função open:

```
open(my $fh, ">", $fileName); # Abertura para Sobrescrita

open(my $fh, ">>", $fileName); # Abertura para Escrita (append)

open(my $fh, "<", $fileName); # Abertura para Leitura
```

```
open(my $fh, "<", $fileName)
    or die "Não foi possível abrir o arquivo";
```

Operações de I/O

Função print:

- Arquivos:

```
print $fh "Escrevendo alguma coisa nesse arquivo\n";
```

- Log:

```
print "Escrevendo alguma coisa na tela\n";
```

Operações de I/O

- Leitura linha por linha:

```
while (my $linha = <$fh>) {
  chomp $linha; #elimina newline
  print "$linha\n";
}
```

- Leitura do arquivo inteiro:

```
chomp(my @linhas = <$fh>);
```

- Fechar arquivo:

```
close $fh
    or warn "Não foi possível fechar arquivo\n";
```

Expressões regulares

Elas são usadas para checar o formato de uma string, formatá-la, substituir e capturar dados, entre outros.

Geralmente, elas são designadas entre barras / e seu reconhecimento pode ser feito através do operador lógico "=~" ou "!~";

Os três operadores mais comuns são:

- m//: Correspondência;
- s///: Substituição;
- tr///: Transliteração;

Sendo que cada um tem seus modificadores.

Expressões regulares

Correspondência:

```
$frase = "preciso achar coisas aqui";

if($frase =~ m/achar/i){
    print "Encontrado: $&".$/;
}
```

```
Encontrado: achar
```

Modificador	Descrição
i	Case insensitive
m	Parar em caso de \n
o	Avalia a expressão somente uma vez
s	Não parar em caso de \n
x	Permite espaço na expressão
g	Todas as correspondências

Expressões regulares

Substituição:

```
$frase = "preciso achar coisas aqui";

$frase =~ s/achar/descobrir/g;
print $frase.$/;
```

```
preciso descobrir coisas aqui
```

Modificador	Descrição
i	Case insensitive
m	Parar em caso de \n
o	Avalia a expressão somente uma vez
s	Não parar em caso de \n
x	Permite espaço na expressão
g	Todas as correspondências
e	Execução de comandos

Expressões regulares

Transliteração:

```
$frase = "preciso achar coisas aqui";

$frase =~ tr/a-c//cd;
print $frase.$/;
```

cacacaa

```
$frase = "preciso achar coisas aqui";

$frase =~ tr/a-c//d;
print $frase.$/;
```

preiso hr oiss qui

Modificador	Descrição
c	Complemento
d	Exclui caracteres encontrados mas não substituídos
s	Elimina repetições

Desvio incondicional

Escape:

- Last: sai do ciclo **mais interno;**
- Next: salta para a próxima iteração do ciclo **mais internos**;
- Redo: salta para o ciclo **mais interno,** sem reavaliar a condição;

Desvio irrestrito:

- Goto: Avança para a próxima instrução determinada pela marca (label).

Exemplos:

last:

```
@vet = (1, 2, 3, 4);

for $i (@vet){
  print $i."\n";

  if($i == 3){
      last;
  }
}
```

```
1
2
3
```

redo:

```
@vet = (1, 2, 3, 4);

FLAG: for $i (@vet){
        print $i."\n";
        if($i == 2){
          $i = "teste";
          redo FLAG;
        }
      }
```

```
1
2
teste
3
4
```

next:

```
@vet = (1, 2, 3, 4);

for $i (@vet){
  if($i == 2){
      next;
  }
  print $i."\n";
}
```

```
1
3
4
```

goto:

```
$i = "bia";

goto LABEL;

$i = "batata";

LABEL: print $i . $/;
```

```
bia
```

Coletor de lixo

Possui coletor de lixo!

Estratégia:

- Contagem referência;
- A linguagem usa contadores de referência para determinar quando as variáveis de um bloco estão prontas para serem coletadas. Isso garante que a informação referenciada não seja deletada.

Modularização

Sub-rotinas (funções):

- São declaradas com a palavra chave **sub**;

```
sub perl{
    print "e ai PEEERRRLL" . $/;
}

print "fora da funcao" . $/;
perl;
```

```
fora da funcao
e ai PEEERRRLL
```

Passagem de argumentos:

A passagem é feita por referência é posicional, por meio do array **@_**.

```
3  my @array = ('oi', 'ola', 'tchau');
4  print "@array\n";
5  modifica(@array);
6  print "@array\n";
7
8  sub modifica {
9    my $aux = @_[0];
10   $_[0] = $_[2];
11   $_[2] = $aux;
12 }
```

```
oi ola tchau
tchau ola oi
```

Passagem de argumentos:

Permite quantidade variável de argumentos.

```
3  Nargumentos(1,2,3);
4  Nargumentos('oi', 'hi');
5
6  sub Nargumentos {
7    my $n = @_;
8    print "Essa funcao recebeu $n argumentos\n";
9  }
```

```
Essa funcao recebeu 3 argumentos
Essa funcao recebeu 2 argumentos
```

Passagem de argumentos de tipos diferentes:

Em caso de argumentos do tipo **@** e **$**, por exemplo, é necessário utilizar referência:

```
3  my @array = ('oi', 'ola', 'tchau');
4  imprimeN(\@array, 1);
5
6  sub imprimeN {
7    my @a = @{$_[0]};
8    my $n = $_[1];
9
10   print "$a[$n]\n";
11 }
```

```
ola
```

Retorno de sub-rotinas:

Retornam qualquer tipo de variável, ou nada:

```
3  my $str = funcao1();
4  print $str;
5  funcao2();
6
7  sub funcao1 {
8    return "Eu tenho retorno\n";
9  }
10
11 sub funcao2 {
12   print "Nao tenho retorno\n";
13   return;
14 }
```

```
Eu tenho retorno
Nao tenho retorno
```

Retorno de sub-rotinas:

Não é necessário usar **return** para retornar variáveis:

```
4  print funcao1();
5  print funcao2();
6
7  sub funcao1 {
8    return "Eu tenho retorno\n";
9  }
10
11 sub funcao2 {
12   my $s = "Tambem tenho retorno\n";
13 }
```

Retornam a última expressão avaliada caso não haja um **return**.

```
Eu tenho retorno
Tambem tenho retorno
```

Orientação a Objetos

- A orientação a objetos nativa é **“hash-based”**;
- Cada classe é representada por um pacote (**package**);
- Utiliza a função **bless** para criar um objeto da classe.

Orientação a Objetos:

Definição da classe, atributos e o construtor:

```perl
package Produto;

sub new{
    my $class = shift;
    my $self = {
        nome => shift,
        preco => shift
    };
    bless($self, $class);
    return $self;
}
```

Orientação a Objetos:

Métodos:

```perl
sub getNome{
    my $self = shift;
    return $self->{nome};
}
sub setPreco{
    my ($self, $preco) = @_;
    $self->{preco} = $preco;
    return $self;
}
sub ehCaro{
    my $self = shift;
    if($self->{preco} > 50){
      return 1
    }
    return 0;
}
```

Orientação a Objetos:

- Permite herança simples e multipla.
- As classes herdadas ficam no array **@ISA**.
- Chama métodos da superclasse com a palavra **SUPER**.

```perl
package Carro;
use Produto;
@ISA = qw(Produto); #heranca

sub new{
    my $class = shift;
    my $self = $class->SUPER::new($_[1], $_[2]);
    $self->{marca} = shift; #novo atributo
    bless ($self, $class);
    return $self;
}

sub ehCaro{
    my $self = shift;
    if($self->{preco} > 50000){
      return 1
    }
    return 0;
}
```

Orientação a Objetos:

```
 8  use Produto;
 9  use Carro;
10
11  my $p = Produto->new("copo", 50.50);
12  my $c = Carro->new("carro", 20600, "Fiat");
13
14  print "Copo caro\n" if ($p->ehCaro());
15  print "Carro barato\n" if (!$c->ehCaro());
```

```
Copo caro
Carro barato
```

Orientação a Objetos: **Moose**

- A forma nativa de de OO que Perl oferece pode ser um pouco confusa e dificil de utilizar.

- Existe um package chamado **Moose** muito recomendado para tornar a OO em Perl mais intuitiva.

- Forma recomendada de Orientação a Objetos pela **perl.org**.

Orientação a Objetos: **Moose**

```perl
package Produto;
use Moose;

has 'nome' => (is => 'rw', isa => 'Str');
has 'preco' => (is => 'rw', isa => 'Num');

sub ehCaro{
    my $self = shift;
    if($self->preco > 50){
      return 1
    }
    return 0;
}
```

```perl
package Carro;
use Moose;
extends 'Produto';

has 'marca' => (is => 'rw', isa => 'Str');

sub ehCaro{
    my $self = shift;
    if($self->preco > 50000){
      return 1
    }
    return 0;
}
```

Orientação a Objetos: **Moose**

- **Is** : define tipo de acesso do atributo e cria os métodos de acesso **readers** e **writers** (tipo get e set de java), pode ser:

    - **rw** : leitura e escrita (cria reader e writer);
    - **ro** : apenas leitura (cria apenas reader);
    - **bare** : não cria nenhum dos métodos de acesso;

- **isa** : define o tipo do atributo, mas não é obrigatório.

Orientação a Objetos: **Moose**

```
 8  use Produto;
 9  use Carro;
10
11  my $p = Produto->new(nome => "copo",
12                       preco => 50.50);
13  my $c = Carro->new(nome => "carro",
14                     preco => 20600,
15                     marca => "Fiat");
16  print "Copo caro\n" if ($p->ehCaro());
17  print "Carro barato\n" if (!$c->ehCaro());
```

Orientação a Objetos: **Moose**

- Permite que uma classe possua referência para arrays e hashes e as trate com suas funções nativas.

```
has 'produtos' => (
    traits  => ['Array'],
    is      => 'ro',
    isa     => 'ArrayRef[Produto]',
    default => sub { [] },
    handles => {
        all_produtos    => 'elements',
        add_produto     => 'push',
        map_produtos    => 'map',
        filter_produtos => 'grep',
        find_produto    => 'first',
        get_produto     => 'get',
        join_produtos   => 'join',
        count_produtos  => 'count',
        has_produtos    => 'count',
        has_no_produtos => 'is_empty',
        sorted_produtos => 'sort',
    },
);
```

Polimorfismo

Perl possui todos os tipos de polimorfismo:

- Coerção;
- Sobrecarga;
- Paramétrico;
- Inclusão;

AD-HOC

- Coerção:

```
1  $num = 2;
2  $str = "9";
3
4  print $num + $str; # Converte $str para inteiro.
5
6  print "\n" . $num . $str; # Converte $num para string.
```

```
11
29
```

AD-HOC

- Sobrecarga:

```
2  package Coisa;
3  use overload
4        "-" => "menos",
5        "+" => "mais";
6
7  sub menos {
8    #operacao
9  }
10 sub mais {
11   #operacao
12 }
```

Universal

- Paramétrico:
  - Automaticamente realizada pela linguagem, já que as sub-rotinas são genéricas, pois não e possível definir o tipo dos parâmetros e do seu retorno;
  - A função **mult3** funciona tanto para string quanto para inteiro, nesse exemplo;

```
1  my $num = 3;
2  my $str = '4';
3
4  mult3($num);
5  mult3($str);
6
7  sub mult3 {
8    my $a = shift;
9    my $m = $a * 3;
10   print "$m\n";
11 }
```

```
9
12
```

Universal

- Inclusão:
  - Feita pela herança de classes.

```
1  package Carro;
2  use Produto;
3  @ISA = qw(Produto); #heranca
```

```
1  package Carro;
2  use Moose;
3  extends 'Produto';
```

Concorrência

Threads:

- Deve ser usado o módulo threads.

```
use threads;
```

- O "interpreter-based threads" fornecido pelo Perl não é o sistema rápido e leve para multitarefas que se poderia esperar. Poucas pessoas sabem como usá-las corretamente ou podem fornecer ajuda.

  - O uso de threads no perl é oficialmente desencorajado.[1]

1 - Conforme documentação da linguagem em: https://perldoc.perl.org/threads.html

Thread-safe VS thread-unsafe

Por padrão, no Perl os dados não são compartilhados entre os threads, logo os módulos Perl têm uma alta chance de ser thread-safe ou pode ser feito thread-safe facilmente.

Mas pode ser usado o módulo **threads::shared** que compartilha as estruturas de dados entre os threads, mas deve ser utilizada com muita cautela.

```perl
#!/usr/bin/perl

use threads;
use threads::shared;

my $var :shared;
my %hsh :shared;
my @ary :shared;


my ($scalar, @array, %hash);
share($scalar);
share(@array);
share(%hash);

```

Podemos compartilhar a variável com :share ou share();

Entretanto devemos ter cuidado para os problemas de executar várias threads simultâneas e acessar algum valor não esperado.

Podemos usar lock() para bloquear o uso de uma variável por mais de uma thread, bloqueando ela no bloco da função que a thread chamar e só liberando ao fim do escopo.

```perl
my $var :shared;
{
  lock($var);
  # $var is locked from here to the end of the block
  ...
  }
# $var is now unlocked
```

Thread

A maneira mais simples e direta de criar um thread é com new ():

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  use threads;
7
8  my $thr = threads->new(\&sub1);
9
10 sub sub1 {
11         print "In the thread\n";
12 }
```

O método new () faz referência a uma sub-rotina e cria um novo thread, que inicia a execução na sub-rotina referenciada.

```
Perl exited with active threads:
        1 running and unjoined
        0 finished and unjoined
        0 running and detached
In the thread
```

OBS: create () também pode ser usado no lugar de new ().

Thread

```perl
$Param3 = "foo";
@ParamList = ('test1','test2','test3');

my $thr = threads->new(\&sub1, "Param 1", "Param 2", $Param3);
$thr = threads->new(\&sub1, @ParamList);
$thr = threads->new(\&sub1, qw(Param1 Param2 Param3));

sub sub1 {
   my @InboundParameters = @_;
   print "In the thread\n";
   print "got parameters >", join("<>",@InboundParameters),"<\n";
}
```

Podemos passar parâmetros para a sub-rotina como parte da inicialização do thread.

Podemos chamar várias threads usando a mesma sub-rotina. Cada thread executa a mesma sub-rotina, mas em threads separadas e sem compartilhar os argumentos.

```
In the thread
got parameters >Param 1<>Param 2<>foo<
In the thread
got parameters >test1<>test2<>test3<
In the thread
got parameters >Param1<>Param2<>Param3<
Perl exited with active threads:
        0 running and unjoined
        3 finished and unjoined
        0 running and detached
```

```
6  @ParamList = ('test1');
7
8  $thr = threads->new(\&sub1, "Param 1");
9  @ReturnData = $thr->join;
10 print "Thread returned @ReturnData".$/;
11
12 $thr = threads->new(\&sub1, @ParamList);
13 @ReturnData = $thr->join;
14 print "Thread returned @ReturnData".$/;
15
16 $thr = threads->new(\&sub1, "NADA");
17 @ReturnData = $thr->join;
18 print "Thread returned @ReturnData".$/;
19
20 sub sub1 {
21    @Param = @_;
22    if ($Param[0] eq "Param 1"){
23          return 1;
24    }
25    elsif ($Param[0] eq "test1"){
26          return 2;
27    }
28    else{
29          return "nothing";
30    }
31 }
```

```
Thread returned 1
Thread returned 2
Thread returned nothing
```

Como as threads também são sub-rotinas, elas podem retornar valores. Às vezes devemos esperar um thread executar e sair para poder usar quaisquer valores que possa retornar.

Para isso, podemos usar usar o método join (), que faz três coisas: aguarda a saída de um thread, limpa depois, executando qualquer limpeza no SO necessária, e retorna todos os dados que o thread possa ter produzido.

Usamos $thr->detach(); caso não precise do retorno e liberar recursos para executar próxima thread em sequência ao mesmo tempo;

```perl
#!/usr/bin/perl
use threads;

$thr = threads->new(\&sub1, "Param 1");
print "Thread 1 run".$/;
@ReturnData1 = $thr->join;
print "Thread returned thread 1 is @ReturnData1".$/;

$thr = threads->new(\&sub1, @ReturnData1); #precisa do retorno do primeiro thread
print "Thread 2 run".$/;

$thr = threads->new(\&sub1, "NADA");
print "Thread 3 run".$/;

sleep(15);
$thr = threads->new(\&sub1, "NADA");
print "Thread 3 re-run after sleep 10s".$/;

sub sub1 {
   @Param = @_;
   if ($Param[0] eq "Param 1"){
        sleep(2);
        return "test1";
        print "Thread 1 finished".$/;
   }
   elsif ($Param[0] eq "test1"){
        sleep(10);
        print "Thread returned thread 2 is sleep 10s".$/;
        print "Thread 2 finished".$/;
   }
   else{
        print print "nothing".$/;
        print "Thread 3 finished".$/;
   }
}
```

```
Thread 1 run
Thread returned thread 1 is test1
Thread 2 run
Thread 3 run
nothing
1Thread 3 finished
Thread returned thread 2 is sleep 10s
Thread 2 finished
Thread 3 re-run after sleep 10s
Perl exited with active threads:
        1 running and unjoined
        2 finished and unjoined
        0 running and detached
nothing
1Thread 3 finished
```

Thread

Podemos checar o estado das threads com:

$thr->is_running()

$thr->is_joinable()

```
# Check thread's state
 if ($thr->is_running()) {
     sleep(1);
 }
 if ($thr->is_joinable()) {
     $thr->join();
 }
```

Thread::Semaphore

use Podemos controlar as threads com o módulo Thread::Semaphore

- up(): Incrementa a variável de controle;
- down(): Decrementa a variável de controle;
- down_nb(): Decrementa o semáforo apenas se obtiver sucesso imediato;
- down_force(): Decrementa o semáforo mesmo se o contador estiver abaixo de 0;
- down_timed(TIMEOUT): Decrementa o semáforo num tempo determinado em segundos.

Corrida Coelho x Tartaruga

```perl
#!/usr/bin/perl

use threads;
use Thread::Semaphore;

my $sem = Thread::Semaphore->new();

my $posCoelho = 1;
my $posTartaruga = 1;

sub corrida{
  ...
}

my $Coelho = threads->new(\&corrida, 1);
my $Tartaruga = threads->new(\&corrida, 2);
$Coelho->join;
$Tartaruga->join;
```

```
sub corrida{
  my $corredor = shift;
  my $i = 0;
  while ($i < 20){
  if($corredor == 1){
        $sem->down;
        print "Coelho   : ";
        for (my $Coelho = 0; $Coelho < $posCoelho; $Coelho++){
              print "-";
        }
        print ">\n";
        $posCoelho++;
        sleep(int(rand(3)));
        $sem->up;
    }
    else{
         $sem->down;
         print "Tartaruga: ";
         for (my $Tartaruga = 0; $Tartaruga < $posTartaruga; $Tartaruga++){
              print "-";
         }
         print ">\n";
         $posTartaruga++;
         sleep(int(rand(3)));
         $sem->up;
      }
              $i++;
   }
}
```

Corrida Coelho X Tartaruga

Coelho : ->
Tartaruga: ->
Coelho : -->
Tartaruga: -->
Coelho : --->
Tartaruga: --->
Coelho : ---->
Tartaruga: ---->
Tartaruga: ----->
Coelho : ----->
Tartaruga: ------>
Tartaruga: ------->
Coelho : ------>
Coelho : ------->
Coelho : -------->
Tartaruga: -------->
Coelho : --------->
Tartaruga: --------->
Coelho : ---------->
Tartaruga: ---------->
Tartaruga: ----------->
Tartaruga: ------------>

Coelho : ------------>
Coelho : ------------>
Coelho : ------------>
Tartaruga: ------------>
Coelho : ------------->
Tartaruga: ------------->
Coelho : --------------->
Coelho : ---------------->
Coelho : ----------------->
Tartaruga: --------------->
Coelho : ------------------>
Tartaruga: ---------------->
Coelho : ------------------->
Coelho : -------------------->
Tartaruga: ----------------->
Tartaruga: ------------------->
Tartaruga: -------------------->
Tartaruga: --------------------->

←Coelho WINS!

Exceções

eval{};

O Perl tem um mecanismo de tratamento de exceções embutido, também conhecido como o bloco eval {}.

A variável $@ é verificada para ver se ocorreu uma exceção.

```perl
#!/usr/bin/perl

eval {
      print $Exist || die Exception->new("The variable \"\$Exist\" don't exist!")
};

if ($@){
  print $@->what;
}


package Exception;
sub new{
  my $class = shift;
  bless { msg=>"@_" }, $class;
}

sub what{
  my ($self) = @_;
  return sprintf "Exception: %s\n", $self->{msg};
}
```

```
Exception: The variable "$Exist" don't exist!
```

```perl
#!/usr/bin/perl

eval {
  print $Exist || die Exception->new("The variable \"\$Exist\" don't exist!");
};

if ($@){
  print $@->what;
}
print "Continued".$/;

eval {
  eval {
        die Exception->new("INTERNAL");
  };
  if ($@){
    die Exception->new($@->what);
  }
};
#precisa relançar caso queira mandar para fora;
if ($@){
  print $@->what;
}

package Exception;
sub new{
  my $class = shift;
  bless { msg=>"@_" }, $class;
}

sub what{
  my ($self) = @_;
  return sprintf "Exception: %s\n", $self->{msg};
}
```

```
Exception: The variable "$Exist" don't exist!
Continued
Exception: Exception: INTERNAL
```

eval{}; e warn();

Podemos usar também o warn ();

Exemplo:

```
3 warn ("ERROR!");
```

```
ERROR! at excpt.pl line 3.
```

Porém tanto o eval() como o warn() não são bons como o try() e catch() do Java, por exemplo.

Mas podemos usar o módulo **Error.pm** e poderemos usar de forma semelhante a Java.

```perl
#!/usr/bin/perl
use Error qw(:try);

try {
    my $result = divide($value, 0); # divide() throws DivideByZeroException
    return $result;
}
catch DivideByZeroException with {
  my $ex = shift;
  print "Error: Caught DivideByZeroException\n";
  return 0;
}
catch MathException with {
  my $ex = shift;
  print "Error: Caught MathException occurred\n";
  return;
};

sub divide {
  @Param = @_;
  if  ($Param[1] == 0){
    throw DivideByZeroException();
  }
  return ($Param[0])/($Param[1]);
}

package MathException;
  use base qw(Error);
  use overload ('""' => 'stringify');
  1;

package DivideByZeroException;
  use base qw(MathException);
  1;
```

```
Error: Caught DivideByZeroException
```

Serialização

Recomendado o uso do pacote padrão Storable.

Existem outros pacotes da comunidade que oferecem outros tipos de serialização (XML, YAML, JSON, etc.)

```perl
#!/usr/bin/perl

use Storable qw(store retrieve);

%cor = ('Azul' => 0.1, 'Vermelho' => 0.8, 'Preto' => 0, 'Branco' => 1);

store(\%cor, 'cores.dat')
    or die "Não foi possível salvar os dados!\n";

$corRef = retrieve('cores.dat');
die "Não foi possível recuperar os dados!" unless defined $corRef;
printf "Vermelho: %lf\n", $corRef->{'Vermelho'};
```

Ainda recebe constantemente pequenas atualizações:

First appeared
December 18, 1987; 30 years ago

Stable release
5.28.0[1] / June 23, 2018; 4 months ago
5.26.2[2] / April 14, 2018; 7 months ago

Preview release
5.29.4[3] / October 20, 2018; 33 days ago



Evolutibilidade

Há a versão 6 do Perl, mas devido a diversos problemas:

- Desenvolvimento lento
- Falta de adesão
- Surgimento de poderosas linguagens de script (ex.: Python e Ruby)
- Críticas quanto à performance

Não é utilizada pela grande parte da comunidade.

Exemplos de algumas diferenças:

Em PERL 6 é obrigatório o uso do espaço após do uso de uma palavra-chave.

A indexação de arrays e hashes não utilizam mais o sign $, mas os próprios símbolos (@ e %).

A referenciação não é feita mais com a contra-barra, mas com item() ou **&** ;

Em PERL 6 não é mais utilizado a seta '->', mas apenas o ponto.

Evolutibilidade

Avaliação da Linguagem

Avaliação da Linguagem:

Critério	C	JAVA	Perl 5
Aplicabilidade	Sim	Parcial	Parcial
Confiabilidade	Não	Sim	Não
Aprendizado	Não	Não	Não
Eficiência	Sim	Parcial	Parcial
Portabilidade	Não	Sim	Sim
Método de Projeto	Estruturado	OO	Estruturado, OO e Funcional
Evolutibilidade	Não	Sim	Parcial
Reusabilidade	Parcial	Sim	Sim
Integração	Sim	Parcial	Sim

Avaliação da Linguagem:

Critério	C	JAVA	Perl 5
Escopo	Sim	Sim	Parcial
Expressões e Comandos	Sim	Sim	Sim
Tipos primitivos e compostos	Sim	Sim	Sim
Gerenciamento de memória	Programador	Linguagem	Linguagem
Persistência dos dados	Biblioteca de funções	JDBC, biblioteca de classes, serialização	Biblioteca de sub-rotinas, serialização.
Passagem de parâmetros	Lista variável e por valor	Lista variável, por valor e por cópia de referência.	Lista variável por referência.

Avaliação da Linguagem:

Critério	C	JAVA	Perl 5
Encapsulamento e proteção	Parcial	Sim	Não
Sistema de tipos	Não	Sim	Parcial
Verificação de tipos	Estática	Estática/Dinâmica	Estática/Dinâmica
Polimorfismo	Coerção e sobrecarga	Todos	Todos
Exceções	Não	Sim	Parcial[1]
Concorrência	Não	Sim	Sim[2]

1 - Possui o bloco eval {}, mas não é eficiente para o tratamento de erros. Possui módulos não nativos para usar try e catch como o módulo Error.pm
2 - Segundo o próprio site: "The use of interpreter-based threads in perl is officially discouraged."

Referências:

- http://www.perl.org/
- http://perldoc.perl.org/
- https://people.redhat.com/tcallawa/whitepapers/PerlThreadingTutorial.pdf
- https://perldoc.perl.org/threads.html
- https://perldoc.perl.org/threads/shared.html
- https://perldoc.perl.org/functions/fork.html
- https://docstore.mik.ua/orelly/perl2/prog/ch17_02.htm
- https://www.perl.com/pub/2002/11/14/exception.html
- https://metacpan.org/pod/distribution/POD2-PT_BR/lib/POD2/PT_BR/perlintro.pod