كتاب: المدخل إلى عالم اللغةPerl

إعداد:معاذ مباركي

E-mail :

mebarkimouaadh@Gmail.com

المدخل إلى عالم اللغةPerl

مقدمة وتعريف بسيط للغة:

لغات البرمجة التفسيرية : هي لغات برمجة يكون البرنامج الناتج عنها، عبارة عن ملف يسمى--SS--و يتم تنفيذه عن طريق برنامج يسمى المفسر الذي يقوم بفك الشفرة الخاصة بالبيانات و الحصول على المعنى لهذه الصيغة لاستخدامها في الغرض الذي أعدت لأجله حيث يفسره أمرا أمرا و ينفذه واحدا تلو الآخر و ما يميز هذا النوع من اللغات أن نص --SS-- لا يتحول الى لغة الالة و انما الى ملف ثنائي byte code [2] و هو عبارة عن سلسلة من تعليمات لغة الآلة الافتراضية, (VM) ، و هو عبارة عن سلسلة من تعليمات لغة الآلة الافتراضية)ويكون هناك مفسر يسمى الآلة الإفتراضية يتم استدعاؤه في كل مرة يتم تنفيذ البرنامج فيها حيث يقوم بالتحويل من الآلة الافتراضية إلى الحقيقية. ويكون تنفيذ هذاbyte code أسرع من النص --SS-- ويعتبر تتبع الأخطاء في اللغات التفسيرية أمرا سهلا لأنها تنفذ سطرا فسطر , و الخلاصة أن البرنامج لا يعمل، من دون مفسر على الجهاز الهدف و الأمثلة عديدة بالنسبة لهذا النوع من لغات البرمجة, و من الأمثلة على هذه اللغات bash و perl و python و php، التي تتداول برامجها على شكل نصي --SS-- وهناك نوع من هذه اللغاتpython ، مثلاً يمكنه تحويل النص إلى لغة آلة وهمية افتراضية لكسب المزيد من السرعة ، ولكن أشهر تلك التي تتداول بشكل ثنائي غير نصي هي جافا java ولغة visual basicعلى الرغم مما يقال إلا أنها تفسيرية لأن الملف التنفيذي هو عبارة ، عن ملف يقوم باستدعاء الآلة الإفتراضية msvbvm60.dllمثلاً التي تتابع عملية، تفسير البرنامج. وعلى الرغم من كل شيء تظل هذه اللغات بطيئة إلا أنها تمتاز بأنها ، ستعمل على أي منصة على أي نظام تشغيل إذا وجد المفسر الخاص بها على ذلك النظام ، وهي فلسفة هكذا لغات "Write once run every where."أي أن تكتبه مرة واحدة، وتنفذه أينما كان . وعلى النقيض من ذلك توجد اللغات التي تعطي ملف بلغة الآلة، مباشرة مثل لغة C/C++ وتسمى عملية تحويل ملف من الكود المصدر، (الملف النصي) إلى لغة الآلة تصنيف compiling ويكون البرنامج، الناتج أسرع ما يكون خصوصا إذا كنت تستعمل مصنف gcc الذي، يمكنه تحسين optimization أي زيادة سرعة برامجك أو تقليل حجمها ، ويمكنه استغلال كافة تعليمات معالج معين ولكن لغة

C/C++ليست للهواة. الهدف الأسمى في اللغات التصنيفية الجيدة و القياسية هو تحقيق العبارة الشهيرة ،

"Write once, compile every where." أي أن تكتبه مرة واحدة وتصنفه أينما كان

مقدمة عن لغة البرمجة prel:

Perl (بيرل)	
التصنيف	لغة برمجة كائنيـــــــــة Oop
سنة الظهور	1987
مصممها	لاري وول
الرخصـــة	رخصة جنو العمومية أو رخصة ارتيستنك
متأثرة بـــ	سي، سي++، أوك، ليسب، سمول توك، باسكال
أثرت في	بايثون، بي إتش بي، روبى
امتدادات الملفات	pl. pm. T.
الموقع الرسمي	perl.org

غة **Practical Extracting and Reporting Language:Perl** هي لغة برمجة متعددة الاستخدام خاصة بالترسيمات، مرتبطة ب **CGI**، هذه اللغة ظهرت سنة **1987** على يد لاري وول. **Perl** لغة مفتوحة المصدر، مرتبطة أساسا بمعالجة المعلومات المرسلة بواسطة الترسيمات.

بيرل هي احد اقوى لغات برمجة المواقع , و قد بدأت و اشتقت من نظام يونكس المعروف بذلك فهي قوية بدرجة تمكنك من بناء موقع متميز و مفيد.

برامج بيرل هي عبارة عن نصوص كتابية "سكربتس" تعطي الاوامر للكومبلير "البرنامج الذي يحلل هذه الاوامر و يشغلها". تختلف بيرل عن الكثير من لغات برمجة المواقع الاخرى بانها سهلة جدا للتعلم , سريعة في تنفيذ الاوامر وسهلة الاستخدام بشكل عام حيث ان الاوامر في بيرل قريبة جدا الى ان تكون كلمات انجليزية و لذلك فانك ببساطة قد تتمكن من فهم الامر من قراءة اسمه فحسب.

لغة **Perl** لغة تخطيطات (مفسّرة) ديناميكية، ديناميكية النمط، عالية المستوى تقارن في الغالب مع لغتي **PHP** و**Python**. صيغة **Perl** أفادت جداً من الأدوات القديمة لبرمجة الصدفة، وقد اشتهرت بإفراطها في استعمال الرموز المربكة التي يستحيل البحث عن معظمها في **Google**. وإرث **Perl** من برمجة الصدفة جعلها لغةً عظيمة إذا تعلق الأمر بكتابة كود صمغي: أي التخطيطات التي تربط التخطيطات والبرامج الأخرى. وتكون مناسبة جداً لمعالجة المعلومات النصية وإنتاجها. ولغة **Perl** واسعة الانتشار، شهيرة، قابلة للحمل جداً، ومدعومة جيداً. وقد صممت **Perl** وفقاً لفذلكة "هناك أكثر من طريقة لإنجاز العمل" " **There's More Than One Way To Do It" (TMTOWTDI**) (بعكس لغة **Python**، حيث "يجب أن توجد طريقة واحدة –ويفضل وحيدة– واضحة لإنجاز العمل" **" there should be one - and preferably only one - obvious way to do it**").

لدى **Perl** أهوالٌ، ولكن لديها أيضاً ما يعوض ذلك من ميزات عظيمة، فمن هذا السياق تكون كأية لغة برمجة أخرى.

أُريد بهذه الوثيقة أن تكون تثقيفية، لا مرجعية. وتستهدف الأشخاص الذين هم مثلي:

ينفرون من وثائق **Perl** الرسمية على الموقع **http://perl.org** لكثافة معلوماتها التقنية، ولإعطائها مساحة واسعة لحالات متطرفة جداً.

يتعلمون لغات البرمجة الجديدة بالطريقة الأسرع "المسَلمات والأمثلة"

يتمنى على **Larry Wall** أن يختصر

يعرفون كيفية البرمجة بالمبادئ العامة

يهتمون بما هو ضروري لإنجاز العمل باستخدام **Perl** ولا يحفلون بما وراء ذلك.

أُريد بهذه الوثيقة أن تكون مختصرة قدر الإمكان، لا أن تكون أخصر من ذلك.

ملاحظات تمهيدية:

- **يمكن في كل ما يتعلق بجميع العبارات التصريحية في هذه الوثيقة أن يقال الآتي "إن هذا ليس صحيحاً إذا توخينا الدقة، إذ الحال في الواقع أكثر تعقيداً". إذا مررت بكذبة ذات أهمية، فأطلعنا عليها، إلا أنني أحتفظ بحقي في حدود ما يشبه الكذب على الأطفال لإضفاء الحيوية.**
- **أستخدم خلال الأمثلة في هذه الوثيقة عبارات print لطباعة خرج البيانات، ولكنني لا أصرح بإلحاق فواصل الأسطر. وقد فعلت ذلك لأمنع نفسي من الجنون، ولتسليط انتباهٍ أكبر على النص الفعلي الذي سيطبع في كل حالة، فهو أهم في جميع الحالات. ففي كثير من الأمثلة سيؤدي هذا إلى دمج الكلمات في سطرٍ واحدٍ معاً إذا ما نفذت الكود في الواقع. حاول أن تتجاهل هذا.**

أهلاً بالعالم – Hello World:

تكتب تخطيطات **Perl** في ملف نصي ينتهي اسمه باللاحقة .**pl**

هاكَ النص الكامل لبرنامج **helloworld.pl**:

```
use strict;

use warnings;

print "Hello world;"
```

تفسَّر تخطيطات **Perl** باستخدام مفسّر **Perl**؛ **perl** أو **perl.exe**:

```
perl helloworld.pl [arg0 [arg1 [arg2[[[...
```

وهاكَ ملاحظاتٍ سريعة: صيغة **Perl** متسامحة جداً؛ لذا ستسمح لك بإنجاز الأعمال بعبارات مشوشة ونتائج مفاجئة. وبما أنك تريد اجتناب هذه لنتائج؛ فلا فائدة كي أقوم بشرحها.
ولاجتنابها أضف **use strict; use warnings;** في أول كل تخطيطة **Perl** تكتبها.
إذ العبارات ذات الشكل **use foo;** تسمى توجيهات. يمرر التوجيه إلى المفسر **perl.exe**، ويظهر أثره عند بدء الفحص الإملائي قبل الشروع في تنفيذ البرنامج، ولا أثر لهذه الأسطر عندما يصادفها المفسر أثناء تشغيل البرنامج.

الرمز # يكوّن بداءة تعليق، ويستمر التعليق حتى نهاية السطر، إذ ليس في **Perl** صيغة تعليق متعدد.

المتغيرات:

تجد المتغيرات في **Perl** على ثلاثة أنواع: الحجميات والمصفوفات والدلاليات. ولكل نوع من هذه الأنواع سابقة رمزية: $ و@ و% على الترتيب المذكور. ويصرح عن المتغيرات بالعبارة **my**، ثم تبقى في المجال حتى نهاية الكتلة المغلقة أو الملف.

المتغيرات الحجمية:

يمكن للمتغيرات الحجمية أن تحوي:

- عدم تعيين **undef** (يوافق **None** في **Python**، أو **null** في **PHP**)
- عدد (لا تفرق **Perl** بين الأعداد الصحيحة والكسرية)
- نص
- مرجع إلى أي متغير آخر.

```
my $undef = undef;
print $undef; # يطبع نصاً فارغاً "" ويطلق تحذيراً
# undef المضمن:
my $undef2;
print $undef2; # يطبع "" ويطلق التحذير ذاته
my $num = 4040.5;
print $num; # "4040.5"
my $string = "world";
print $string; # "world"
```

يستخدم المُعامل . لسلسلة النصوص (تماماً كما في**PHP**)

```
print "Hello ".$string; # "Hello world"
```

"المنطقيات":

لا تملك Perl نوع بياناتٍ منطقي. فإذا وُجد متغيرٌ حجميٌ في عبارة الشرط if فسيُحسب على أنه "خطأ" فقط في إحدى هذه الحالات:

- **Undef**
- **العدد 0**
- **النص ""**
- **النص "0".**

تكرر وثائق Perl القول بأن توابعاً تعيد القيمة "صواب" أو "خطأ" في حالات معينة. عملياً، عندما يُذكر عن تابع أنه يعيد القيمة "صواب" فهو عادة ما عيد قيمة العدد 1. وعندما يُذكر أنه يعيد القيمة "خطأ" فهو عادة ما يعيد نصاً فارغاً، "".

النمط الهشّ:

يستحيل تحديد نوع محتويات متغير حجمي بين "العدد" أو "النص". لنقل بكلامٍ أدق؛ يجب ألا تحتاج إلى هذا أبداً.يعتمد سلوك الحجمي بين العدد أو النص على المُعامل المستعمل معه: فعندما يستعمل كنص سيسلك الحجمي سلوك النص، وعندما يستعمل كعدد سيسلك سلوك العدد (وسيطلق تحذيراً عند عدم إمكان هذا):

```
my $str1 = "4G";

my $str2 = "4H";

print $str1 .  $str2; # "4G4H"

print $str1 +  $str2; # "8" مع تحذيرين

print $str1 eq $str2; # "" (نص فارغ، أي خطأ)

print $str1 == $str2; # "1" مع تحذيرين

# الخطأ التقليدي

print "yes" == "no"; # "1" مع تحذيرين؛ إذ ستحسب كلا القيمتين 0 عندما يستعملان كعددين
```

تعلّم أن تستعمل دوماً المُعامل الصحيح بحسب الحال. لدينا معاملات مختلفة لمقارنة الحجميات كأعداد، وأخرى لمقارنة الحجميات كنصوص:

```
#المُعاملات العددية: *, +, <=>, =!, ==, =<, =>, <, >

#المُعاملات النصّية: lt, gt, le, ge, eq, ne, cmp, ., x
```

متغيرات المصفوفات:

متغير المصفوفة: قائمة بحجميات معنونة بعدد صحيح يبدأ من 0. تُعرَف في Pythonبالقائمة، وفي PHP بالمصفوفة.يُصرح عن المصفوفة باستخدام قائمة حجميات محصورة بقوسين:

```
my @array) =

"print,"

"these,"

"strings,"

"out,"

"for,"

"me", # لا بأس بالفاصلة الأخيرة

);
```

عليكَ استخدام علامة الدولار للوصول إلى قيمة من المصفوفة، لأن القيمة التي تُستجلب ليست مصفوفة بل حجمي:

```
print $array[0]; # "print"

print $array[1]; # "these"

print $array[2]; # "strings"

print $array[3]; # "out"

print $array[4]; # "for"

print $array[5]; # "me"

print $array[6]; # يعيد undefاً ويطلق تحذير ، ويطبع ""
```

يمكنك استخدام القيم المرجعية السالبة لاستجلاب المُدخلات بدءً من النهاية فصبّاً:

```
print $array[-1]; # "me"

print $array[-2]; # "for"

print $array[-3]; # "out"

print $array[-4]; # "strings"

print $array[-5]; # "these"

print $array[-6]; # "print"

print $array[-7]; # يعيد undefاً ويطلق تحذير ، ويطبع ""
```

لا اختلاف بين الحجمي $var وبين المصفوفة @var التي تحوي مُدخلاً حجمياً $var[0].

على أن هذا يربك القارئ فتجنبه.

لمعرفة طول مصفوفة:

```
"print "This array has ".(scalar @array)."elements"; # "This array has 6 elements

"print "The last populated index is ".$#array;      # "The last populated index is 5
```

تخزن الوسائط الممررة إلى تخطيط **Perl** الأصلي عند استدعائه في المصفوفة المضمنة **@ARGV**.

يمكن إقحام المتغيرات في النصوص:

```
print "Hello $string"; # "Hello world"

print "@array";       # "print these strings out for me"
```

احذر. ستُخزن –في يومٍ ما– بريد أحدٍ ما في نص، **"jeff@gmail.com".** يؤدي هذا إلى أن يبحث **perl** عن مصفوفة باسم **@gmail** لتفسيرها في النص ولن يجدها، مما ينتج خطأً في التشغيل. يمكن تجنب هذا الإقحام بطريقتين: سبقها برمز الشرطة المائلة الخلفية لتهريب، أو باستخدام الحاصرة المفردة عوضاً عن الحاصرة المزدوجة.

```
print "Hello \$string"; # "Hello $string"

print 'Hello $string';  # "Hello $string"

print "\@array";       # "@array"

print '@array';        # "@array"
```

المتغيرات الدلالية:

المتغير الدلالي قائمةٌ من الحجميات معنونة بنصوص. تعرف في **Python** بالمعجمات، وتعرف في **PHP** بالمصفوفات.

```
my %scientists) =

        "Newton"   => "Isaac,"

        "Einstein" => "Albert,"

        "Darwin"   => "Charles,"

;(
```

لاحظ الشبه بين هذا التصريح والتصريح عن مصفوفة.

في الواقع، يسمى رمز السهم المزدوج => بـــ"الفاصلة السمينة **"fat comma"** ، إذ ما هو إلا مرادف للفاصلة. يصرح عن الدلالي باستخدام قائمة تحوي عدداً زوجياً من العناصر المعنونة زوجياً (**0**, **2**, ...) تعامل جميها معاملة النصوص هنا من جديد، عليكَ استعمال علامة الدولار للوصول إلى قيمة من قيم الدلالي، لأن القيمة المستجلبة ليست دلالياً، إنما حجمي

```
print $scientists{"Newton"};   # "Isaac"

print $scientists{"Einstein"}; # "Albert"

print $scientists{"Darwin"};   # "Charles"

print $scientists{"Dyson يعيد #    ;{"undefاً، ويطلق تحذير"" ، ويطبع
```

لاحظ الأقواس المستخدمة هنا. ثم نقول كذلك هنا؛ لا اختلاف بين الحجمي **var$** وبين الدلالي **var%** الذي يحوي مُدخلاً حجمياً **$foo"}var{"**.

يمكنك تحويل الدلالي مباشرة إلى مصفوفة بضعفي عدد مدخلاته، مُبادلاً بين المفتاح والقيمة (والعكس بنفس السهولة):

```
my @scientists = %scientists;
```

ولكن، مفاتيح الدلالي –بخلاف المصفوفة– ليس لها ترتيب معيّن. ستُعاد بأي ترتيب فعّال. ولذلك، لاحظ إعادة الترتيب لكن مع الحفاظ على الأزواج في المصفوفة الناتجة:

```
print "@scientists" شيء شبيه بـ # ;"Einstein Albert Darwin Charles Newton Isaac"
```

لنجمل الكلامَ ولنقل: إن عليك استعمال الأقواس المربعة لاستجلاب قيمة من مصفوفة، وعليك استعمال الأقواس المزخرفة لاستجلاب قيمة من دلالي. الأقواس المربعة عاملٌ عدديٌّ فعّال، والأقواس المزخرفة عاملٌ نصيٌّ فعّال. ولا أهمية أبداً –في الواقع– من كون العنوان عدداً أو نصاً:

```
my $data = "orange;"

my @data = ("purple;("

my %data = ( "0" => "blue;("


print $data;     # "orange"

print $data[0];   # "purple"

print $data["0"]; # "purple"

print $data{0};   # "blue"

print $data{"0"}; # "blue"
```

القوائم:

القائمة في **Perl** شيءٌ مختلفٌ عن الدلالي أو المصفوفة، وقد رأيتَ سابقاً عدة قوائم:

```
)
    "print,"
    "these,"
    "strings,"
    "out,"
    "for,"
    "me,"
(


)
    "Newton"   => "Isaac,"
    "Einstein" => "Albert,"
    "Darwin"   => "Charles,"
(
```

يست القائمة متغيراً، إنما قيمةٌ غير ثابتة يمكن إسنادها

إلى مصفوفة أو دلالي. وهذا سبب كون صيغة التصريح عن مصفوفة ذاتها للتصريح عن دلالي. وهناك العديد من الحالات التي يمكن إطلاق المصطلحين "قائمة" و"مصفوفة"

بالتبادل، ولكن هناك حالات كثيرة بنفس القدر حيث يظهر فرقٌ خفي بين القائمة والمصفوفة فيسبب نتائج مربكة للغاية.

حسناً، تذكر أن => مجرد , مقنّعة، ثم تابع هذا المثال:

```
("one", 1, "three", 3, "five", 5)
("one" => 1, "three" => 3, "five" => 5)
```

يشير استعمال => إلى أن إحدى هاتين القائمتين تصرح عن مصفوفة والأخرى عن دلالي، ولكن ولا واحدة منهما تصرح بنفسها عن شيء، إنهما مجرد قائمتين متطابقتين.

كذلك:

```
()
```

فلا إشارة إلى شيء هنا، فيمكن أن تشير هذه القائمة إلى مصفوفة فارغة أو دلالي فارغ، ومن الواضح أن المفسر **perl** لا يستطيع بأية حال أن يعرف الفرق. حالما تفهم هذا الاعتبار الغريب لـــ**Perl**، ستفهم أيضاً سبب وجوب كون الحقيقة التالية صحيحة دوماً: تداخل القوائم غير ممكن. وجرب بنفسك:

```
my @array) =
    "apples,"
    "bananas,"
    )
      "inner,"
      "list,"
      "several,"
      "entries,"
```

```
    ,(
    "cherries,"
);
```

("inner", "list", "several", طريقة لمعرفة إذا ما كانت Perl ليس لدى أنها ليست هذه ولا تلك Perl مصفوفة داخلية أو دلالي داخلي. ولذلك يفترض ("entries" إنما يسّوي هاتين القائمتين في قائمة واحدة طويلة

```
print $array[0]; # "apples"
print $array[1]; # "bananas"
print $array[2]; # "inner"
print $array[3]; # "list"
print $array[4]; # "several"
print $array[5]; # "entries"
print $array[6]; # "cherries":
```

ويبقى هذا صحيحاً سواءً استخدمت الفاصلة السمينة أو لا:

```
my %hash) =
    "beer" => "good,"
    "bananas) <= "
        "green"  => "wait,"
        "yellow" => "eat,"
    ,(
;(
```

```
# يطلق المثال أعلاه تحذيراً لأن الدلالي صُرّح عنه بقائمة ذات 7 عناصر

print $hash{"beer"};    # "good"

print $hash{"bananas"}; # "green"

print $hash{"wait"};    # "yellow;"

print $hash{"eat"};     # undefويطلق تحذيراً "" لذلك يطبع ,
```

وطبعاً، يسهل هذا دمج عدة مصفوفات:

```
my @bones   = ("humerus", ("jaw", "skull"), "tibia;("

my @fingers = ("thumb", "index", "middle", "ring", "little;("

my @parts   = (@bones, @fingers, ("foot", "toes"), "eyeball",
"knuckle;("

print @parts
```

السياق في prel

أهم ميزة في Perl أن الكود يتحسس السياق. كلّ تعبيرٍ في Perl يُحسب إما في سياق الحجمي أو في سياق القائمة، وذلك بحسب ما إذا كان من المتوقع أن ينتج حجمياً أو قائمة. تعمل الكثير من تعبيرات Perl وتوابعها المضمنة بشكل مختلف جداً اعتماداً على نوع السياق الذي تعمل عليه.

فسيؤدي الإسناد للحجمي مثل $scalar = إلى احتساب تعبيره في سياق الحجمي. وفي هذا الحال التعبير "Mendeleev" وقيمة العائد نفسُ قيمة الحجمي "Mendeleev":

```
my $scalar = "Mendeleev;"
```

وسيؤدي الإسناد للمصفوفة أو الدلالي مثل @array = أو %hash = إلى احتساب تعبيره في سياق القائمة. وفي هذا الحال ستحسب قيمة القائمة في سياق قائمة وتعيد القائمة نفسها، مما يؤدي بعد ذلك إلى أن يرتب ليملأ المصفوفة أو الدلالي:

```
my @array = ("Alpha", "Beta", "Gamma", "Pie;("

my %hash = ("Alpha" => "Beta", "Gamma" => "Pie;("
```

حساب التعبير الحجمي في سياق قائمة يؤدي إلى قائمة وحيدة العناصر:

```
my @array = "Mendeleev' لــ مساوٍ # ;"my @array = ("Mendeleev';("
```

وحساب تعبير القائمة في سياق الحجمي يعيد الحجمي الأخير في القائمة:

```
"Pie" الآن "scalar$ قيمة # ;("my $scalar = ("Alpha", "Beta", "Gamma", "Pie
```

وحساب تعبير القائمة (أتذكر أن المصفوفة مختلفة عن القائمة؟) في سياق الحجمي يعيد طول المصفوفة:

```
my @array = ("Alpha", "Beta", "Gamma", "Pie");

my $scalar = @array; # قيمة $scalar4 الآن
```

يحسب التابع المضمن **print** جميع وسائطه في سياق القائمة. بل في الواقع؛ إن **print** يقبل عدداً غير محدد من الوسائط، ويطبعها تباعاً؛ مما يعني أنه يمكن استخدامه لطباعة المصفوفات مباشرة:

```
my @array = ("Alpha", "Beta", "Goo");

my $scalar = "-X";

print @array;          # "AlphaBetaGoo"

print $scalar, @array, 98; # "-X-AlphaBetaGoo98"
```

يكنك إجبار أي تعبير ليحسب في سياق الحجمي باستخدام التابع المضمن **scalar**. والواقع أن هذا سبب استعمالنا لـــ **scalar** لنحصل على طول مصفوفة.

ليس هناك أية قاعدة أو صيغة تفرض عليك أن تعيد قيمة حجمية عندما يحسب إجراء فرعي في سياق الحجمي، وكذلك الأمر في حال سياق القائمة. فكما قد رأيت مما مر؛ إن **Perl** مؤهلة تماماً لاستخراج النتيجة لك.

المراجع وبُنى المعطيات المتداخلة

كما مرّ في عدم إمكان احتواء القائمة لقائمةٍ أخرى كعناصرَ فيها؛ كذلك فإن المصفوفات والدلاليات لا يمكنها احتواء مصفوفات أو دلاليات أخرى كعناصر فيها. فلا إمكان إلا لاحتواء الحجميات. شاهد ما قد يحدث إذا ما حاولنا ذلك:

```
my @outer = ("Sun", "Mercury", "Venus", undef, "Mars");
my @inner = ("Earth", "Moon");

$outer[3] = @inner;

print $outer[3]; # "2"
```

$outer[3] حجمي، وسيتطلب قيمةً حجمية لهذا السبب. عندما تحاول إسناد قيمة مصفوفة كـ **@inner** إليه، فستحسب **@inner** في سياق الحجمي. وهذا مماثل لإسناد **scalar @inner**، أي طول المصفوفة **@inner**، ويساوي **2**.

ولكن قد يحتوي المتغير مرجعاً إلى أي متغير، يشمل هذا متغيرات المصفوفات ومتغيرات الدلاليات. وبهذه الطرق تنشأ بنى المعطيات الأعقد في **Perl**.

ينشأ المرجع باستخدام الشرطة المائلة الخلفية.

```
my $colour  = "Indigo";

my $scalarRef = \$colour;
```

عندما تريد استخدام اسم المتغير، يمكنك بدلاً عن ذلك فتح قوسين مزخرفين، ثم تكتب مرجعاً للمتغير داخلهما.

```
"print $colour;        # "Indigo

"(SCALAR(0x182c180" مثال #       ;print $scalarRef

print ${ $scalarRef }; # "Indigo"
```

وطالما ترى أن النتائج واضحة، يمكنك التخلي عن القوسين كذلك:

```
print $$scalarRef; # "Indigo"
```

إذا كان المرجع يشير إلى متغير مصفوفة أو دلالي فيمكنك الحصول على البيانات منه باستخدام الأقواس المزخرفة أو باستعمال معامل أشهر– > :

```
my @colours = ("Red", "Orange", "Yellow", "Green", "Blue;("

my $arrayRef = \@colours;

print $colours[0وصول مباشر للمصفوفة #       ;[

print ${ $arrayRef }[0استخدام المرجع للوصول للمصفوفة # ;[

print $arrayRef->[0نفس سابقه تماماً #    ;[

my %atomicWeights = ("Hydrogen" => 1.008, "Helium" =>
4.003, "Manganese" => 54.94;(

my $hashRef = \%atomicWeights;

print $atomicWeights{"Heliumوصول مباشر للدلالي # ;{"

print ${ $hashRef }{"Heliumاستخدام المرجع للوصول للدلالي #  ;{"

print $hashRef->{"Heliumنفس سابقه تماماً – وهذا شائعٌ جداً #    ;{"
```

التصريح عن بنية معطيات

لدينا هنا أربعة أمثلة، إلا أن آخرها الأكثر انتشاراً عند التطبيق.

```
my %owner1) =
        "name" => "Santa Claus,"
        "DOB"  => "1882-12-25,"
;(

my $owner1Ref = \%owner1;

my %owner2) =
        "name" => "Mickey Mouse,"
        "DOB"  => "1928-11-18,"
;(

my $owner2Ref = \%owner2;

my @owners = ( $owner1Ref, $owner2Ref;(

my $ownersRef = \@owners;
```

```
my %account) =
        "number" => "12345678,"
        "opened" => "2000-01-01,"
        "owners" => $ownersRef,
;(
```

وترى هنا إجهاداً واضحاً بلا ضرورة، إذ بإمكانك اختصاره إلى:

```
my %owner1) =
    "name" => "Santa Claus,"
    "DOB"  => "1882-12-25,"
;(

my %owner2) =
    "name" => "Mickey Mouse,"
    "DOB"  => "1928-11-18,"
;(

my @owners = ( \%owner1, \%owner2;(
```

```
    my %account) =
      "number" => "12345678,"
      "opened" => "2000-01-01,"
      "owners" => \@owners,
;(
```

بالإمكان كذلك التصريح عن مصفوفات أو دلاليات غير مسماة باستخدام رموز مختلفة. استخدم الأقوال المربعة لمصفوفة غير مسماة والأقواس الهلالية للدلاليات غير المسماة. ولعل قيمة العائد في كل حالةٍ مرجعٌ إلى بنية معطيات غير مسماة. انظر مدقّقاً، فهذه النتائج account% ذاته كالسابق أعلاه:

```
# الأقواس الهلالية تشير إلى دلالي غير مسمى
my $owner1Ref} =
      "name" => "Santa Claus,"
      "DOB"  => "1882-12-25,"
;{

my $owner2Ref} =
```

```
        "name" => "Mickey Mouse,"
        "DOB"  => "1928-11-18,"
;{


 #Square brackets denote an anonymous array
my $ownersRef = [ $owner1Ref, $owner2Ref;[


my %account) =
        "number" => "12345678,"
        "opened" => "2000-01-01,"
        "owners" => $ownersRef,
;(
```

أو، للاختصار (وهذا الشكل الذي يجب أن تستخدمه في الواقع عندما تصرح عن بنى معطيات معقدة سطرية):

```
my %account) =

        "number" => "31415926,"

        "opened" => "3000-01-01,"

        "owners] <= "

                }

                        "name" => "Philip Fry,"

                        "DOB"  => "1974-08-06,"

                ,{

                }

                        "name" => "Hubert Farnsworth,"

                        "DOB"  => "2841-04-09,"

                ,{

        ,[

;(
```

الوصول للمعلومات في البُنى:

والآن، لنفترض أن لديك **account%** ما يزال يعمل في الأرجاء ولكن كل شيء عداه (إن كان ثَم) تلاشى خارج المجال. بإمكانك طباعة المعلومات بعكس كل إجراء بذاته في كل حالة. مجدداً، لدينا هنا أربع أمثلة، والأخيرة منها أكثرها فائدة:

```
my $ownersRef = $account{"owners;{"

my @owners    = @{ $ownersRef;{

my $owner1Ref = $owners[0;[

my %owner1    = %{ $owner1Ref;{

my $owner2Ref = $owners[1;[

my %owner2    = %{ $owner2Ref;{

print "Account #", $account{"number"}, "\n;"

print "Opened on ", $account{"opened"}, "\n;"

print "Joint owners:\n;"

print "\t", $owner1{"name"}, " (born ", $owner1{"DOB"},
")\n;"

print "\t", $owner2{"name"}, " (born ", $owner2{"DOB"},
")\n;"
```

أو، للاختصار:

```
my @owners = @{ $account{"owners;{ {"

my %owner1 = %{ $owners[0;{ [

my %owner2 = %{ $owners[1;{ [

print "Account #", $account{"number"}, "\n;"

print "Opened on ", $account{"opened"}, "\n;"

print "Joint owners:\n;"

print "\t", $owner1{"name"}, " (born ", $owner1{"DOB"},
")\n;"

print "\t", $owner2{"name"}, " (born ", $owner2{"DOB"},
")\n;"
```

أو باستخدام المراجع والمُعامل –> :

```
my $ownersRef = $account{"owners;{"

my $owner1Ref = $ownersRef->[0;[

my $owner2Ref = $ownersRef->[1;[

print "Account #", $account{"number"}, "\n;"

print "Opened on ", $account{"opened"}, "\n;"

print "Joint owners:\n;"
```

```
print "\t", $owner1Ref->{"name"}, " (born ", $owner1Ref-
>{"DOB"}, ")\n;"

print "\t", $owner2Ref->{"name"}, " (born ", $owner2Ref-
>{"DOB"}, ")\n;"
```

وإذا تجاوزنا جميع القيم الابتدائية تماماً:

```
;"print "Account #", $account{"number"}, "\n

;"print "Opened on ", $account{"opened"}, "\n

;"print "Joint owners:\n

print "\t", $account{"owners"}->[0]->{"name"}, " (born ",
;"$account{"owners"}->[0]->{"DOB"}, ")\n

print "\t", $account{"owners"}->[1]->{"name"}, " (born ",
;"$account{"owners"}->[1]->{"DOB"}, ")\n
```

كيف تؤذي نفسك من غير أن تشعر مستخدماً المراجع إلى للمصفوفات:

هذه المصفوفة تحوي أربعة عناصر:

```
my @array1 = (1, 2, 3, 4, 5;(

print @array1; # "12345"
```

وهنا مصفوفة تحوي عنصراً وحيداً (والحال أنه مرجع إلى مصفوفة غير مسماة بخمس عناصر):

```
;[my @array2 = [1, 2, 3, 4, 5

"(ARRAY(0x182c180" مثال # ;print @array2
```

وهذا حجمي مرجع إلى مصفوفة غير مسماة بخمس عناصر:

```
my $array3Ref = [1, 2, 3, 4, 5];

print $array3Ref;     # مثال "ARRAY(0x22710c0)"

print @{ $array3Ref }; # "12345"

print @$array3Ref;    # "12345
```

الشروط: if / elsif / else /

لا مفاجآت هنا، عدا عن صيغة **elsif**:

```
my $word = "antidisestablishmentarianism;"

my $strlen = length $word;


if($strlen >= 15} (

        print "", $word, "' is a very long word;"

 {elsif(10 <= $strlen && $strlen < 15} (

        print "", $word, "' is a medium-length word;"

 {else}

        print "", $word, "' is a a short word;"

{
```

تقدم **Perl** صيغة أقصر من "عبارة **if** الشرطية" يستحسن استخدامها جداً في العبارات القصيرة:

```
print "", $word, "' is actually enormous" if $strlen >= 20;
```

unless ... else

```
my $temperature = 20;


unless($temperature > 30} (

        print $temperature, " degrees Celsius is not very hot;"

 {else}

        print $temperature, " degrees Celsius is actually pretty
hot;"

{
```

لأفضل عموماً اجتناب كُتَل **unless** كما يُجتنب البلاء لأنها مربكة للغاية. فكتلة " **unless ... else]]**" يمكن أن تكون ببساطة إعادة إنتاج لكتلة "**if [... else]**" بعكس الشرط [أو بإبقاءه وعكس الكُتل] ولا توجد –حمداً لله– عبارة **elsunless**.

ولكن العبارة التالية –بالمقارنة– مستحسنة للغاية لأنها سهلة القراءة جداً:

```
print "Oh no it's too cold" unless $temperature > 15;
```

مُعامل الشرط الثلاثي

يسمح معامل الشرط الثلاثي **?**: بتضمين عبارات **if** الشرطية البسيطة في عبارة. مثال صغير على استخدامه في صيغتي المفرد/ والجمع في اللغة الإنكليزية:

```
my $gain = 48;

print "You gained ", $gain, " ", ($gain == 1 ? "experience
point" : "experience points;"!" ,("
```

على الهامش: يفضل أن تكتب الكلمتين كاملتين في الحالين. لا تفعل شيئاً ذكياً كالتالي، فإذا بحث أحد في الملف المصدر ليستبدل الكلمتين **"tooth"** أو **"teeth"** لن يجد ذلك السطر أبداً:

```
my $lost = 1;

print "You lost ", $lost, " t", ($lost == 1 ? "oo" : "ee"), "th;"!
```

ويمكن تداخل المعاملات الثلاثية:

```
my $eggs = 5;

print "You have ", $eggs == 0 ? "no eggs: "

$                eggs == 1 ? "an egg:  "

"                some eggs;"
```

تَحسب عبارات **if** الشرطية الشروطَ في سياق الحجمي. فعلى سبيل المثال، تعيد **if(@array)** صواباً فقط إذا كانت **@array** تحتوي عنصراً أو أكثر. ولا يهم ما هي هذه العناصر – قد تحتوي **undef** أو قيمة أخرى تمثل "خطأً" منطقياً نريدها فيها.

الحلقات:

هناك أكثر من طريقة لإنجازها **There's More Than One Way To Do It**.

تملك **Perl** حلقة **while** المألوفة:

```
my $i = 0;

while($i < scalar @array} (

        print $i, ": ", $array[$i;[

        $i;++

{
```

كما تقدم **Perl** أيضاً العبارة:**until**

```
my $i = 0;

until($i >= scalar @array} (

        print $i, ": ", $array[$i;[

        $i;++

{
```

للسابقتين (سيطلق تحذيرٌ إذا كانت @array فارغة):

الحلقتين التاليتيين من نوع do تساويان تقريباً للسابقتين (سيطلق تحذيرٌ إذا كانت @array فارغة):

```
my $i = 0;

do}

        print $i, ": ", $array[$i;[

        $i;++

 {while ($i < scalar @array;(
```

وهذه الثانية

```
my $i = 0;

do}

        print $i, ": ", $array[$i;[

        $i;++

 {until ($i >= scalar @array;(
```

حلقة **for** الأساسية في لغة **C** متاحة نفسها أيضاً. لاحظ كيف وضعنا **my** داخل عبارة الحلقة **for**، مما يصرح عن **i$** في مجال الحلقة فقط:

```
for(my $i = 0; $i < scalar @array; $i} (++

        print $i, ": ", $array[$i;[

{

$ لم يعد #i موجوداً هنا، وهذا أكثر ترتيباً بكثير.
```

يعتبر هذا النوع من حلقاتfor طرازاً قديماً وينبغي اجتنابه عند الإمكان. المسح التكراري الأصيل للقوائم أجمل بكثير. ملاحظة: بخلاف PHP؛ عبارتيfor وforeach مترادفتين. فقط استعمل ما تراه أكثر قابلية للقراءة:

```
foreach my $string ( @array} (
        print $string;
{
```

إذا كنت تريد محددات، فإن معامل المجال .. ينشأ قائمة غير مسماة من الأعداد الصحيحة:

```
foreach my $i ( 0 .. $#array} (
        print $i, ": ", $array[$i;[
{
```

لا يمكن تطبيق المسح التكراري على الدلالي. ولكن، يمكنك تطبيقه على مفاتيح الدلالي. استخدم التابع المضمن keys للحصول على مصفوفة تحوي جميع مفاتيح الدلالي. ثم استخدم foreach بنفس طريقة استعمالها مع المصفوفات:

```
foreach my $key (keys %scientists} (
        print $key, ": ", $scientists{$key;{
{
```

وطالما أن الدلالي غير مرتب بترتيب معين، فإن المفاتيح ستعود بأي ترتيب. استعمل التابع المضمن sortلترتيب المصفوفة من المفاتيح ألف بائياً قبل استعمالها:

```
foreach my $key (sort keys %scientists} (
        print $key, ": ", $scientists{$key;{
{
```

إذا لم تعيّن ماسحاً تكرارياً، فسيستخدم **Perl** الماسح التكراري الافتراضي، $_. والمتغير $_ هو الأول والأكثر استخداماً من المتغيرات المضمنة:

```
foreach ( @array} (
        print;_$
{
```

عند استخدام المتغير الافتراضي، وكنت تريد فقط تطبيق عبارة وحيدة في حلقتك، فيمكنك استخدام صيغة الحلقة فائقة الاختصار:

```
print $_ foreach @array;
```

التحكم بالحلقة

يمكن استخدام **next**و **last**للتحكم بسير الحلقة، وتعرفان في معظم لغات البرمجة بـــ **continue**و **break**على الترتيب. كما يمكننا تسمية الحلقة بعلامة اختيارية، والعُرف أن تكتب العلامات كلها بحروف كبيرة **.ALLCAPITALS** عندما تسمى الحلقات، يمكن أن تستهدِف **next** و **last**تلك العلامات بعينها. هذا المثال يستخرج الأعداد الأولية تحت **100:**

```
CANDIDATE: for my $candidate ( 2 .. 100} (
        for my $divisor ( 2 .. sqrt $candidate} (
                next CANDIDATE if $candidate % $divisor == 0;
        {
        print $candidate." is prime\n;"
{
```

توابع المصفوفات:

تعديل المصفوفات في مكانها:

سنستعمل المصفوفة **@stack** في الأمثلة:

```
my @stack = ("Fred", "Eileen", "Denise", "Charlie;("
print @stack; # "FredEileenDeniseCharlie
```

popيستخرج ويعيد العنصر الأخير من المصفوفة. يمكن أن يمثل هذا أعلى المكدّس:

```
print pop @stack; # "Charlie"
print @stack;     # "FredEileenDenise"
```

pushيُلحِق عنصراً إضافياً آخرَ المصفوفة:

```
push @stack, "Bob", "Alice;"
print @stack; # "FredEileenDeniseBobAlice"
```

shift يستخرج ويعيد العنصر الأول من المصفوفة:

```
print shift @stack; # "Fred"

print @stack;      # "EileenDeniseBobAlice"
```

unshift يدرج عناصراً جديدة بدايةَ المصفوفة:

```
unshift @stack, "Hank", "Grace;"

print @stack; # "HankGraceEileenDeniseBobAlice"
```

التوابع **pop**، و**push**، و**shift**، و**unshift** جميعها حالات خاصة من التابع **splice**.

التابع **splice** يحذف ويعيد قطعة من مصفوفة، مستبدلاً إياها بقطعة أخرى:

```
print splice(@stack, 1, 4, "<<<", ">>>"); #
"GraceEileenDeniseBob"

print @stack;                    # "Hank<<<>>>Alice"
```

إنشاء مصفوفات جديدة من قديمة:

تقدم **Perl** التوابع التالية التي تعمل على المصفوفات لإنشاء مصفوفات أخرى.

فالتابع **join** يُسلسلُ عدة نصوص في نص واحد:

```
my @elements = ("Antimony", "Arsenic", "Aluminum",
"Selenium;("

print @elements;          #
"AntimonyArsenicAluminumSelenium"

print "@elements";        # "Antimony Arsenic Aluminum
Selenium"

print join(", ", @elements); # "Antimony, Arsenic, Aluminum,
Selenium"
```

في سياق القائمة؛ يعيد التابع **reverse** قائمةً بترتيب معكوس. في سياق الحجمي، يسلسل التابع**reverse** القائمة كلها معاً ثم يعكسها باعتبارها كلمة واحدة:

```
print reverse("Hello", "World");      # "WorldHello"

print reverse("HelloWorld");          # "HelloWorld"

print scalar reverse("HelloWorld");   # "dlroWolleH"

print scalar reverse("Hello", "World"); # "dlroWolleH"
```

التابع **map** يأخذ دخلاً مصفوفةً ويطبق عملية على كل حجمي $_ في هذه المصفوفة، ثم يبني مصفوفة جديدة من النتائج. تكتب العملية المراد تطبيقها بتعبيرٍ وحيدٍ محصوراً بقوسين مزخرفين

```
my @capitals = ("Baton Rouge", "Indianapolis", "Columbus",
"Montgomery", "Helena", "Denver", "Boise");


print join ", ", map { uc $_ } @capitals;

# "BATON ROUGE, INDIANAPOLIS, COLUMBUS,
MONTGOMERY, HELENA, DENVER, BOISE"
```

التابع **grep** يأخذ دخلاً مصفوفةً ويعيد خرجاً مصفوفةً مرشحة. وتشبه صيغته صيغة **map**. ولكن هنا؛ يحسب الوسيط الثاني لكل حجمي $_ في مصفوفة الدخل، فإذا أعاد قيمة منطقية صواباً؛ يوضع الحجمي في مصفوفة الخرج، وإلا فلا.

```
print join ", ", grep { length $_ == 6 } @capitals;

# "Helena, Denver"
```

كما هو واضح، طول المصفوفة الناتجة هو عدد المطابقات الناجحة، مما يعني أنك يمكنك استخدام **grep** للفحص السريع عن احتواء المصفوفة لعنصر ما:

```
print scalar grep { $_ eq "Columbus" } @capitals; # "1"
```

يمكن استخدام **grep** و**map** معاً لتشكيل القوائم الشاملة، وهي ميزة قوية جداً مفقودة في كثير من لغات البرمجة أخرى.

افتراضياً، يعيد التابع **sort** مصفوفة الدخل، مرتبةً ترتيباً (ألف بائياً)

```
my @elevations = (19, 1, 2, 100, 3, 98, 100, 1056;(


print join ", ", sort @elevations;

"98 ,3 ,2 ,19 ,1056 ,100 ,100 ,1" #
```

ولكن، كما هو الحال في **grep** و**map**؛ يمكنك تزويد بعض الكود الخاص بك. فالترتيب يتم دوماً باختبار سلسلة مقارنات بين عنصرين. فتستقبل كتلتك **$a** و**$b** دخلاً ويجب أن تعيد **-1** إذا كان **$a** "اقل من" **$b**،**0** إذا كانا "متساويين"، و**1** إذا كان**$a** "أكبر من" **$b**.

والمعامل **cmp** يقوم بهذا العمل على النصوص:

```
print join ", ", sort { $a cmp $b } @elevations;

"98 ,3 ,2 ,19 ,1056 ,100 ,100 ,1" #
```

أما "معامل سفينة الفضاء"، <=>؛ فيقوم بهذا على الأعداد:

```
print join ", ", sort { $a <=> $b } @elevations;

"1056 ,100 ,100 ,98 ,19 ,3 ,2 ,1" #
```

يكون**$a** و**$b** دوماً حجميان، ولكنهما يمكن أن يكونا مرجعين إلى أغراض معقدة، مما يصعب عملية المقارنة. إذا احتجت لمساحة إضافية لإجراء المقارنة، فيمكنك إنشاء إجراء فرعي منفصل ثم تزويد اسمه بدلاً عن ذلك:

```
sub comparator}
        #lots of code…
        #return -1, 0 or 1
{

print join ", ", sort comparator @elevations;
```

ولا يمكنك فعل ذلك في معاملات **grep** أو **map**.

لاحظ كيف أن الإجراء الفرعي والكتلة لا تخصصان أبداً مع **$a** و**$b**. في الواقع؛ إن **$a** و**$b** مثلَ **$_** متغيران عامان تسكنهما أزواج من القيم لتتم مقارنتهما في كل مرة.

التوابع المضمنة

رأيت حتى الآن دزينة –على الأقل– من التوابع المضمنة: **print**، **sort**، **map**، **grep**، **keys**، **scalar** الخ. التوابع المضمنة إحدى أعظم نقاط القوة في **Perl**. وهذه التوابع:

- كثيرة.
- مفيدة للغاية.
- وثائقها غنية بالمعلومات
- مختلفة جداً في صيغها، لذلك راجع الوثائق.
- قبل أحياناً التعابير النظامية وسائطاً.
- أحياناً تقبل كتلة كاملة من الكود وسيطاً.
- أحياناً لا تتطلب فصل الوسائط بفاصلة.
- أحياناً تأخذ عدداً معيناً من الوسائط مفصولة بفاصلة وأحياناً لا.
- أحياناً تملأ وسائطها الخاصة إذا زودت بما يقل عن المطلوب.
- لا تتطلب عموماً إحاطة وسائطها بقوسين إلا في ظروف الالتباس.

النصيحة الفضلى فيما يتعلق بالتوابع المضمنة؛ أن تعلم أنها موجودة. تصفح الوثائق للمرجعية مستقبلاً. إذا مررت بمهمة تشعر أنها من مهام البنى التحتية، وشائعة كفاية لتكون قد احتيجت مراتٍ كثيرةً من قبل، فهناك فرصة أن تجدها هناك بالفعل.

الإجراءات الفرعية المعرفة:

يُصرح عن الإجراءات الفرعية بعبارة **sub**. وعلى العكس من التوابع المضمنة، فإن الإجراءات الفرعية المعرفة تقبل دائماً الدخل ذاته: قائمةً من الحجميات. وقد تحوي طبعاًعنصراً وحيداً، أو قد تكون فارغةً. والحجمي الوحيد يؤخذ كقائمة تحوي عنصراً وحيداً. والدلالي ذو الـــ**N** عنصراً يؤخذ كقائمة تحوي **N2** عنصراً.

مع أن الأقواس اختيارية، إلا أنه ينبغي استدعاء الإجراءات الفرعية دوماً باستخدام الأقواس، حتى عندما تُستدعى بلا وسائط. إذ إن هذا يوضح عن حدوث عملية استدعاء لإجراء فرعي.

وحالما تصبح داخل الإجراء الفرعي. تتاح الوسائط باستخدام متغير المصفوفة المضمنة @_.

مثال:

```
sub hyphenate {


  # Extract the first argument from the array, ignore everything
else

  my $word = shift @_;


  # An overly clever list comprehension

  $word = join "-", map { substr $word, $_, 1 } (0 .. (length
$word) - 1);

  return $word;

}


print hyphenate("exterminate"); # "e-x-t-e-r-m-i-n-a-t-e"
```

تفريغ الوسطاء:

هناك أكثر من طريقة (**More Than One Way**) لتفريغ @_، ولكن بعضها أفضل من بعض.

والإجراء الفرعي المثال **left_pad** بالأسفل؛ يزيح نصاً المسافةَ المطلوبة، مستخدماً محرف الإزاحة المُزوِّد به. (يسلسلُ التابع **x** عدداً من نُسَخِ النص ذاته على التوالي.) (ملاحظة: بسبب الاختصار، فإن جميع هذه الإجراءات الفرعية تفتقر إلى فحص أخطاء أساسي. أي التأكد من أن محرف الإزاحة واحد فقط، وأن العرض أكبر أو يساوي طول النص الموجود، والتأكد من أن جميع الوسطاء المطلوبة قد مُرّرت.)

أنموذج عن استدعاء **left_pad** كالتالي:

```
print left_pad("hello", 10, "+"); # "+++++hello"
```

01: بعض الناس لا يفرغون الوسطاء على الإطلاق ويستعملون @_ "مباشرة". وهذا قبيحٌ وينصح باجتنابه:

```
sub left_pad}
        my $newString = ($_[2] x ($_[1] - length $_[0])) . $_[0;[
        return $newString;
{
```

02: تفريغ @_ يُجتنَب فقط أقل من سابقه بقليل:

```
sub left_pad}

        my $oldString = $_[0;[

        my $width     = $_[1;[

        my $padChar   = $_[2;[

        my $newString = ($padChar x ($width - length
$oldString)) . $oldString;

        return $newString;

{
```

03: تفريغ @_ بحذف البيانات منها باستخدام **shift** مستَحسَنٌ لـــ**4** وسطاء فأقل:

```
sub left_pad}

        my $oldString = shift;_@

        my $width     = shift;_@

        my $padChar   = shift;_@

        my $newString = ($padChar x ($width - length
$oldString)) . $oldString;

        return $newString;

{
```

إذا لم يزود **shift** بأية مصفوفة، فسيعمل على @_ تضميناً. وترى هذا الأسلوبَ شائعاً جداً:

```
sub left_pad}

        my $oldString = shift;

        my $width     = shift;

        my $padChar   = shift;

        my $newString = ($padChar x ($width - length
$oldString)) . $oldString;

        return $newString;

{
```

عندما يزيد عدد الوسطاء عن **4**؛ تصعب متابعة "أين" أسند إلى "أيّ" وسيط.

04: يمكنك تفريغ @_ دفعةً واحدة، بالإسناد إلى عدة حجميها بالتزامن. وكذلك هنا، لا بأس بهذا لـــ**4** وسطاء فأقل:

```
sub left_pad {

        my ($oldString, $width, $padChar) = @_;

        my $newString = ($padChar x ($width - length
$oldString)) . $oldString;

        return $newString;

}
```

05: في الإجراءات الفرغية التي تستدعى مع عدد كبير من الوسطاء، أو حينما يكون بعض الوسطاء اختياراً، أو لا يمكن استخدامه مع مجموعات من وسطاء أخرى؛ فالاختيار الأفضل أن تطلب من المستخدم تمرير دلالي من الوسطاء عندما يستدعي الإجراء الفرعي، ثم تقوم بتفريغ @_ مجدداً في ذلك الدلالي من الوسطاء على هذه الطريق، سيبدو استدعاء إجرائنا الفرعي مختلفاً قليلاً:

```
print left_pad("oldString" => "pod", "width" => 10, "padChar "
;("+" <=
```

وسيبدو الإجراء الفرعي بهذا الشكل:

```
sub left_pad}

    my %args;_@ =

    my $newString = ($args{"padChar"} x ($args{"width"} -
length $args{"oldString"})) . $args{"oldString;{"

    return $newString;

{
```

إعادة القيم:

قد تُظهر الإجراءات الفرعية – كالكثير من تعبيرات **Perl** – سلوكاً مرتبطاً بالسياق. يمنكك استخدام التابع **wantarray** (الذي كان يجب أن يُدعى **wantlist** ولكن ما علينا) للكشف عن أي سياق قد حُسب فيه الإجراء الفرعي، وإعادة قيمة مناسبة لهذا السياق:

```
sub contextualSubroutine {
        # المستدعي يريد قائمة، إعادة قائمة
        return ("Everest", "K2", "Etna") if wantarray;

        # المستدعي يريد حجمياً، إعادة حجمي
        return 3;
}

my @array = contextualSubroutine();
print @array; # "EverestK2Etna"

my $scalar = contextualSubroutine();
print $scalar; # "3"
```

استدعاءات النظام:

معذرة إذا كنت على علمٍ بالحقائق التالية غير المرتبطة بـــPerl. في كل مرة ينتهي الإجراء في نظام ويندوز أو لينكس (ولعله في أكثر الأنظمة الأخرى). يختتم بكلمة 16–بت للتعبير عن حالته تُسمى كلمة الحالة. الـــ8 بتات العلوية تعيّن كود الإعادة بين 0 و255 متضمنة الـــ0 والـــ255، حيث يمثل 0 عادةً نجاحاً تاماً، وتمثل القيم الأخرى درجات مختلفة من الفشل. أما الـــ8 بتات الأخرى فقلما تفحص – فهي "تعكس نمط الفشل، كإشارة الإنهاء، وبيانات خرج التنقيح".

يمكنك الخروج من برنامج Perl مستخدماً كود إعادة من اختيارك (من 0 إلى 255) باستخدام التابع exit.

تقدم Perl أكثر من طريقة (More Than One Way To) لإنتاج إجراء –باستدعاءٍ وحيد–، وإمكاث البرنامج الحالي حتى ينتهي الإجراء الابن، ثم استكمال تفسير البرنامج الحالي. مهما كانت الطريقة المستعملة، ستجد بعد ذلك مباشرة، أن المتغير الحجمي المضمن $? يحمل كلمة الحالة من انتهاء الإجراء الابن، ويمكنك أخذ الـــ8 بتات العلوية من هذه الـــ16 بت: $? >> 8.

يمكن استخدام التابع system لاستدعاء برنامج آخر بالوسطاء المسرودة. والقيمة المعادة من system هي ذات القيمة التي يحملها $?:

```
my $rc = system "perl", "anotherscript.pl", "foo", "bar",
"baz";

$rc >>= 8;

print $rc; # "37"
```

ويمكن بدلاً عن ذلك استعمال حاصرتين خلفيتين `` لتشغيل أمر حقيقي في سطر الأوامر وأخذ الخرج من ذلك الأمر. في سياق الحجمي؛ يكون الخرج العائد كله نصاً وحيداً. أما في سياق القائمة فيكون الخرج العائد مصفوفة من النصوص، كل نص يمثل سطراً من الخرج.

```
my $text = `perl anotherscript.pl foo bar baz;`

print $text; # "foobarbaz"
```

سترى هذا الناتج إذا كانت محتويات **anotherscript.pl** على سبيل المثال:

```
use strict;

use warnings;


print @ARGV;

exit 37;
```

الملفات ومقابض الملفات:

يمكن أن يحوي المتغير الحجمي مقبض ملف بدلاً من رقم/نص/مرجع أو **undef**. مقبض الملف أساساً مرجعٌ إلى مكان معيّن داخل ملفٍ معين.

استخدم **open** لتحويل متغير حجمي إلى مقبض ملف. يجب تزويد **open** بنمط الفتح. النمط < يشير إلى أننا نريد أن نفتح الملف لنقرأ منه:

```
my $f = "text.txt;"

my $result = open my $fh, "<", $f;


if(!$result} (

        die "Couldn't open '".$f."' for reading because;!$." :

{
```

إذا نجح، **open** فسيعيد قيمة صواب منطقي، وإلا فسيعيد قيمة خطأ، وسيخزن رسالة خطأ في المتغير المضمن $!. وكما رأيت أعلاه؛ يجب عليك دوماً فحص عملية **open** والتأكد من تمام نجاحها. هذا الفحص يصبح مملاً قليلاً؛ فصار التالي شكلاً شائعاً:

```
open(my $fh, "<", $f) || die "Couldn't open '".$f."' for reading
because;!$." :
```

لاحظ حاجة **open** إلى إحاطة وسطاء استدعائها بأقواس.

لقراءة سطر من مقبض ملف؛ استخدم التابع المضمن **readline**. يعيد **readline** سطراً نصياً كاملاً، متضمناً فاصل الأسطر في نهاية السطر (عدا عن إمكانية عدمه عند آخر سطر من الملف)، أو يعيد **undef** إذا وصلتَ إلى نهاية الملف.

```
while(1) {
        my $line = readline $fh;
        last unless defined $line;
        # معالجة السطر...
}
```

لقطع تذييل فاصل الأسطر، استخدم **chomp**:

```
chomp $line;
```

لاحظ أن **chomp** يعدل**$line** في مكانه. أي إنك قد لا تريد الشكل التالي **$ = line chomp $line**.

يمكنك كذلك استخدام **eof** للكشف عن الوصول إلى نهاية الملف:

```
while(!eof $fh} (
        my $line = readline $fh;
        $ معالجة #line...
{
```

ولكن احذر من استخدامك الشكل **while(my $line = readline $fh)**، لأنه عندما يصبح **$line** مساوياً للـــ"**0**"، فستتوقف الحلقة مبكراً. إذا أردت كتابة شيءٍ مشابه، تقدم

Perl المعامل <> الذي يقدم readline بشكلٍ آمن نسبياً. سترى التالي شائعاً جداً وآمناً بامتياز:

```
while(my $line = <$fh>) (<

        $ معالجة #line...

{
```

وحتى هذا:

```
while(<$fh>) {

        # معالجة $_...

}
```

وتبدأ الكتابة في الملف بفتحه في نمط مختلف. النمط > يشير إلى أنك تريد فتح الملف والكتابة فيه. (سيدمر > محتويات الملف الهدف إن كانت موجودة في الملف سابقاً. لتفتح الملف وتكتب في نهايته استخدم نمط >>.) ثم، ببساطة زود التابع print بمقبض الملف على أنه الوسيط الصفر.

```
open(my $fh2, ">", $f) || die "Couldn't open '".$f."' for
writing because;!$." :

print $fh2 "The eagles have left the nest;"
```

لاحظ غياب الفاصلة بين $fh2 والوسيط التالي.

تغلق مقابض الملفات في الواقع تلقائياً عندما تخرج من المجال، وإلا:

```
close $fh2;
```

```
close $fh;
```

هناك ثلاثة مقابض ثوابت عامةSTDIN :، وSTDOUT، وSTDERR، وهي تفتح تلقائياً عندما يبدأ تشغيل البرنامج. لقراءة سطر وحيد من دخل المستخدم:

```
my $line = <STDIN> ;
```

لتقتصر على انتظار المستخدم ليضغط زر الإدخال Enter:

```
<STDIN>;
```

واستدعاء <> أي مقبض ملف سيقرأ البيانات من STDIN، أومن أية ملفات ذكرت في الوسطاء عندما استدعي تخطيط Perl.

وكما قد تكون خمنت؛ فإن print تطبع إلى STDOUT افتراضياً إذا لم يذكر أي مقبض ملف

اختبارات الملفات:

التابع **-e** تابعٌ مضمنٌ يفحص ما إذا كان الملف المذكور موجوداً.

```
print "what" unless -e "/usr/bin/perl;"
```

لتابع **-d** تابعٌ مضمنٌ يفحص ما إذا كان الملف المذكور دليلاً.

التابع**-f** تابعٌ مضمنٌ يفحص ما إذا كان الملف المذكور ملفاً عادياً.

هذه مجرد ثلاثة أمثلة من طائفة واسعة من التوابع ذات الشكل **-X** حيث **X** محرفٌ صغير -أو كبير- وتُدعى هذه التوابع باختبارات الملفات. لاحظ علامة الناقص السابقة. في بحث **Google** تستعمل علامة الناقص لاستثناء العبارة من النتائج، وهذا يجعل البحث عن اختبارات الملفات صعباً فيه! ابحث فقط عن "**perl file test**" عوضاً عن ذلك.

التعابير النظامية:

تظهر التعابية النظامية في كثير من الملفات والأدوات غير **Perl**. نواة صيغة التعابير النظامية في **Perl** مبدئياً ذاتُها في أي مكان آخر، ولكن الإمكانات الكاملة للتعابير النظامية معقدة بشكلٍ مخيف وصعبة الفهم. أفضل نصيحة يمكنني أن أقدمها لك أن تتجنب هذا التعقيد حيثما أمكنك.

عمليات المطابقة يمكن أن تنفذ باستخدام =~ m//. ففي سياق الحجمي؛ تعيد =~ m// صواباً عند النجاح، وخطأً عند الفشل.

```
my $string = "Hello world;"

if($string =~ m/(\w+)\s+(\w} (/(+

        print "success;"

{
```

نفّذ الأقواس مطابقة فرعية. وبعد تنفيذ عملية مطابقة ناجحة، فإن المطابقات الفرعية تخزن في المتغيرات المضمنة $1، $2، $3، ...:

```
print $1; # "Hello"

print $2; # "world"
```

أما في سياق القائمة فتعيد //m =~ المتغيرات $1، $2، ... كقائمة.

```
my $string = "colourless green ideas sleep furiously";

my @matches = $string =~
m/(\w+)\s+((\w+)\s+(\w+))\s+(\w+)\s+(\w+)/;


print join ", ", map { "'".$_."'" } @matches;

# يطبع "'colourless', 'green ideas', 'green', 'ideas', 'sleep',
'furiously'"
```

تنفذ عمليات الاستبدال باستخدم =~ s///.

```
my $string = "Good morning world";

$string =~ s/world/Vietnam/;

print $string; # "Good morning Vietnam"
```

لاحظ كيف تغيرت محتويات $string. يجب عليك أن تمرر متغيراً حجمياً على الجانب الأيسر من العملية =~ s/// فإذا مررت نصاً حرفياً، ستحصل على خطأ.

يشير العلم /g إلى "مطابقة جماعية".

في سياق الحجمي؛ يبحث كل نداء لـ=~ m//g عن مطابقة أخرى بعد المطابقة السابقة؛ فيعيد صواباً عند النجاح، وخطأ عند الفشل. يمكنك الوصول إلى $1 وأمثاله بعد ذلك بالطريقة المعتادة. على سبيل المثال:

```
my $string = "a tonne of feathers or a tonne of bricks;"

while($string =~ m/(\w+)/g} (

  print """.$1."'\n;"

{
```

أما في سياق القائمة؛ فيعيد نداء m//g =~ جميع المطابقات دفعةً واحدة.

```
my @matches = $string =~ m/(\w+)/g;

print join ", ", map { """.$_."" } @matches;
```

وينفذ s///g =~ بحثاً واستبدالاً جماعياً ويعيد عدد المطابقات. وهنا، سنستبدل جميع الحروف الصوتية بالحرف "r".

```
# تجربة مرة بدون /g.

$string =~ s/[aeiou]/r/;

print $string; # "r tonne of feathers or a tonne of bricks"


# مرة أخرى.

$string =~ s/[aeiou]/r/;

print $string; # "r trnne of feathers or a tonne of bricks"


# وتنفيذ جميع الباقي باستخدام /g

$string =~ s/[aeiou]/r/g;

print $string, "\n"; # "r trnnr rf frrthrrs rr r trnnr rf brrcks"
```

ويجعل العلم /i المطابقات والاستبدالات غير حساسة لحالة الأحرف.

ويسمح العلم /x للتعبيرات النظامية أن تحوي مسافات بيضاء (كفواصل الأسطر) وتعليقات

```
"Hello world" =~ m/

  (\w+) # كلمة مكونة من محرف أو أكثر

  [ ]  # مسافة حرفية وحيدة،  مخزنة في صف محرف

  world # "world" حرفياً

/x;


# يعيد صواباً
```

الوحدات والرزم:

في **Perl**؛ الوحدات والرزم شيئين مختلفين.

- الوحدات:

الوحدة ملف .**pm** تستطيع تضمينها في ملف **Perl** آخر (تخطيط أو وحدة). الوحدة ملف نصي بصيغة ملفات تخطيطات .**pl** ذاتها. مثال على وحدة قد تكون موجودة في **C:\foo\bar\baz\Demo\StringUtils.pm** أو **foo/bar/baz/Demo/StringUtils.pm/**، وتقرأ كالتالي:

```
use strict;

use warnings;


sub zombify}

        my $word = shift;_@

        $word =~ s/[aeiou]/r/g;

        return $word;

{
```

```
return 1;
```

يجب عليك أن تعيد قيمة الصواب في نهاية الوحدة لإظهار أنها حُملت بنجاح؛ لأن الوحدة تنفذ من الأعلى للأسفل عندما تُحمّل.

ولجعل مفسر **Perl** مستطيعاً للعثور عليهم، يجب سرد الأدلة التي تحوي وحدات **Perl** في المتغير البيئي **PERL5LIB** قبل استدعاء **perl**. اسرد الدليل الجذر الحاوي للوحدات، لا تسرد أدلة الوحدات أو الوحدات نفسها:

```
set PERL5LIB=C:\foo\bar\baz;%PERL5LIB%
```

أو

```
export PERL5LIB=/foo/bar/baz:$PERL5LIB
```

بمجرد إنشاء وحدة **Perl** وصار **perl** أين يبحث عنها، يمكنك استخدام التابع المضمن **require** للبحث عنها وتنفيذها أثناء برنامج **Perl**. على سبيل المثال؛ استدعاء **require Demo::StringUtils** سيؤدي إلى أن يبحث مفسر **Perl** في كل دليل مسرود في **PERL5LIB** بالدور، باحثاً عن ملف يُدعى **Demo/StringUtils.pm**. وبعد أن تُنفَذ الوحدة، تصبح الإجراءات الفرعية التي عُرفت هناك متاحةً فجأة في التخطيط الأساسي. سنسمي مثالنا **main.pl** ويُقرأ كالتالي:

```
use strict;
use warnings;

require Demo::StringUtils;

print zombify("i want brains"); # "r wrnt brrrns"
```

لاحظ استعمال اثنتين من النقاط المزدوجة :: كفاصل بين الأدلة.

والآن تظهر هذه المشكلة: إذا حوى **main.pl** كثيراً من استدعاءات **require** وكلٌ من الوحدات التي ستُستَدعى تحوي كثيراً من استدعاءات **require** أيضاً؛ قد يصعّب ذلك تعقب التصريح الأصلي عن الإجراء الفرعي **zombify**() وحل هذه المشكلة أن تستخدم الرزم.

- الرزم

الرزمة نطاق أسماء حيث يمكن التصريح عن الإجراءات الفرعية. وأي إجراء فرعي ستصرح عنه، سيكون تصريحاً ضمنياً في الرزمة الحالية. في بداية التنفيذ، تكون أنت في الرزمة **main** ولكن يمكنك تبديل الرزمة باستخدام التابع المضمن **package**:

```
use strict;
use warnings;

sub subroutine {
        print "universe";
}

package Food::Potatoes;

# لا تعارض:
sub subroutine {
        print "kingedward";
```

```
}
```

لاحظ استخام اثنتين من النقاط المزوجة :: كفاصل بين نطاقات الأسماء.

حينما تستدعي إجراءً فرعياً؛ فأنت تستدعي –ضمنياً– إجراءً فرعياً من داخل الرزمة الحالية. أو بدلاً عن ذلك يمكنك تخصيص اسم الرزمة. شاهد ما يحصل إذا أكملنا التخطيط السابق:

```
subroutine();             # "kingedward"

main::subroutine();       # "universe"

Food::Potatoes::subroutine(); # "kingedward"
```

فالحل المنطقي إذاً للمشكلة المشروحة أعلاه؛ تعديل

C:\foo\bar\baz\Demo\StringUtils.pmأول

/foo/bar/baz/Demo/StringUtils.pmليصير:

```
use strict;

use warnings;


package Demo::StringUtils;


sub zombify {

        my $word = shift @_;

        $word =~ s/[aeiou]/r/g;
```

```
        return $word;
}


return 1;
```

وتعديل **main.pl** ليصير:

```
use strict;
use warnings;


require Demo::StringUtils;


print Demo::StringUtils::zombify("i want brains"); # "r wrnt brrrns"
```

والآن اقرأ التالي بعناية.

الوحدات والرزم ميزتين منفصلتين تماماً، ومتباينتين تماماً في لغة البرمجة **Perl.** وحقيقة أن كليهما يَستعملان اثنتين من النقاط المزدوجة مربكة للغاية. في الإمكان تبديل الرزم عدة مرات خلال سير التخطيط أو الوحدة. وفي الإمكان أيضاً استخدام نفس التصريح عن الرزمة في عدة أمكنة في عدة ملفات. الاستدعاء **require Foo::Bar** لا يبحث ولا يستدعي ملفاً ذا تصريح **package Foo::Bar**مكانٍ ما فيه، ولا يُحمّل بالضرورة الإجراءات الفرعية في نطاق الأسماء **.Foo::Bar** الاستدعاء **require Foo::Bar** فقط يستدعي الملف المُسمى **Foo/Bar.pm**، الذي لا يحتاج إلى أي نوع من التصريح عن الرزمة بداخله على الإطلاق،

بل –في الواقع– قد يصرح عن **package Baz::Qux** وبعض الهراء بداخله من قبيل ما تعلمته.

وبالمثل، فإن استدعاء الإجراء الفرعي **Baz::Qux::processThis()** لا يعني بالضرورة أنه التصريح عنه موجودٌ داخلَ ملف باسم **Baz/Qux.pm.** بل يمكن أن يصرح عنه في "أي مكانٍ" حرفياً.

وفصل هذين المفهومين واحدةٌ من أغبى ميزات**Perl** ، ومعاملتهما على أنهما مفهومين منفصلين يجعل النتائج دوماً في أكواد مجنونةً فوضوية. ولحسن حظنا؛ فإن معظم مبرمجي **Perl** يطيعون القانونين التاليين:

- يجب على تخطيط **Perl** (ملف .**pl**) ألا يحوي أبداً أي تصريحات **package**.
- يجب على وحدة **Perl** (ملف .**pm**) أن تحوي دوماً تصريح **package** وحيداً تماماً، متوافقاً مع اسمه وموقعه. مثال: الوحدة **Demo/StringUtils.pm** يجب أن تبدأ بـ**package Demo::StringUtils**.

ولهذا السبب ستجد في أن معظم الرزم والوحدات –في الواقع العملي– التي كتبها أناسٌ آخرون موثوقون؛ يمكن اعتبارها متطابقة والإشارة إليها بالتبادل. ولكن من المهم ألا تأخذ هذا على أنه من المسلمات، لأنك ستقابل يوماً ما كوداً كتبه شخصٌ مجنون.

Perl كائنية التوجه:

ليست **Perl** تلك اللغة العظيمة عندما يتعلق الأمر بالبرمجة كائنية التوجه، فالإمكانات كائنية التوجه في **Perl** أضيفت إليها لاحقاً بعد إصدارها، وهذا يُظهر أنّ:

- الكائن –ببساطة– مرجعٌ (أي متغير حجمي) يعلم أي صف تنتمي مرجعيته إليه. لتخبر مرجعاً أن مرجعيته إلى صف؛ استخدم **bless**. لتعرف أي صفٍ تعود إليه مرجعية المرجع (إن وجدت) استخدم **ref**.
- الوظيفة –ببساطة– إجراء فرعي يتوقع مرجعاً (أو في حال وظائف الصفوف؛ يتوقع اسم رزمة) في وسيطه الأول. تستدعى وظائف الكائنات باستخدام **$obj->method()**، وتستدعى وظائف الصفوف باستخدام **Package::Name->method()**.
- الصف –ببساطة– رزمة حَوَت وظائفاً.

المثال السريع يجعل هذا أوضح. وحدة المثال **Animal.pm** تحوي صف الحيوان **Animal** كالتالي:

```
use strict;
use warnings;

package Animal;

sub eat {
        # الوسيط الأول دوماً هو الكائن الذي سيُعمل عليه.
```

```
        my $self = shift @_;


        foreach my $food ( @_ ) {
                if($self->can_eat($food)) {
                        print "Eating ", $food;
                } else {
                        print "Can't eat ", $food;
                }
        }
}


# افترض أن الحيوان ،  لأغراض التمثيل Animal يمكنه أكل أي شيء.
sub can_eat {
        return 1;
}


return 1;
```

ويمكننا استخدام هذا الصف كالتالي:

```
require Animal;


my $animal} =
```

```
        "legs"   => 4,
        "colour" => "brown,"
$ #                    مرجع عادي لدلالي{animal};
print ref $animal;      # "HASH"
bless $animal, "Animal" والآن صار كائناً من صف # ;"Animal"
print ref $animal;      # "Animal"
```

لاحظ: أي مرجع يمكن أن يرتبط بأي صف حرفياً. ويعود إليك أن تتأكد أن (1) المرجعية يمكن حقاً أن تستخدم كنسخة من هذا الصف و(2)أن الصف قد يكون موجوداً وقد تم تحميله.

ويمكنك العمل مع الدلالي الأصلي بالطريقة المعتادة:

```
print "Animal has ", $animal->{"legs"}, " leg(s)";
```

ولكن يمكنك الآن أيضاً استدعاء وظائف الكائن باستخدام ذات المعامل –>، كالتالي:

```
$animal->eat("insects", "curry", "eucalyptus");
```

هذا الاستدعاء الأخير مساوٍ لـــ Animal::eat($animal, "insects", "curry", "eucalyptus")

التوابع البانية:

الباني وظيفة صف تعيد كائناً جديداً. إذا أردت بانٍ فصرح عنه. ويمكنك استخدام أي اسمٍ أردت. وبالنسبة لوظائف الصف، فالوسيط الأول الممرر ليس كائناً إنما اسم صف وفي هذه الحالة "**Animal**":

```
use strict;
use warnings;


package Animal;


sub new {
        my $class = shift @_;
        return bless { "legs" => 4, "colour" => "brown" }, $class;
}


# ...الخ.
```

ثم استخدمه كالتالي:

```
my $animal = Animal->new;()
```

الوراثة:

لإنشاء صف يرث من صفٍ أب، استخدم **use parent**. لنفترض أننا نورّث **Animal** لـــ**Koala**، الموجود في **Koala.pm**:

```
use strict;
use warnings;


package Koala;


# يرث من Animal
use parent ("Animal");


# تجاوز وظيفة واحدة
sub can_eat {
	my $self = shift @_; # Not used. You could just put "shift @_;" here
	my $food = shift @_;
	return $food eq "eucalyptus";
}
```

```
return 1;
```

وكود مثال:

```
use strict;
use warnings;

require Koala;

my $koala = Koala->new();

$koala->eat("insects", "curry", "eucalyptus"); # يأكل فقط the eucalyptus
```

يحاول استدعاءُ الوظيفةِ الأخيرِ أن ينفذ **Koala::eat($koala, "insects", "curry", "eucalyptus"**")، ولكن الإجراء الفرعي **eat**() غير معرف في رزمة **Koala**. ولكن، بسبب أن **Koala** لديه صف أب **Animal**، سيحاول مفسر **Perl** استدعاء **Animal::eat($koala, "insects", "curry", "eucalyptus**") بدلاً عن ذلك، فينجز العمل. لاحظ كيف حُمّل الصف **Animal** تلقائياً في **Koala.pm**.

بما أن **use parent** تقبل قائمة من أسماء الصفوف، فإن **Perl** تدعم الوراثة المتعددة، يشمل ذلك كل محاسنها ومساوئها.

كُتل BEGIN:

تنفذ كتلة **BEGIN** حالما ينتهي **perl** من تحليل تلك الحزمة، حتى لو بقي عليه بعد تنفيذها أن يتم تحليل باقي الملف. ثم يتم تجاهلها في وقت التنفيذ:

```
use strict;
use warnings;

print "This gets printed second";

BEGIN {
        print "This gets printed first";
}

print "This gets printed third";
```

تُنفذ كتلة **BEGIN** أولاً دوماً. وإذا أنشأت عدة كتل **BEGIN** (لا تفعل ذلك)؛ ستنفذ هذه الكتل بالترتيب من الأعلى للأسفل كلما وصل إليها المفسر. تنفذ كتلة **BEGIN** أولاً دوماً حتى لو وضعت في أثناء الملف (لا تفعل هذا) أو في آخره (ولا هذا). لا تعبث بالترتيب الطبيعي للكود. ضع كتل **BEGIN** في البداية!

تُنفذ كتلة **BEGIN** حالما تُحلل. وحالما ينتهي تنفيذها، يُستكمل التحليل من نهاية كتلة **BEGIN**. وفقط عندما ينتهي تحليل كامل التخطيط أو الوحدة؛ ينفذ الكود خارج كتل **BEGIN**.

```
use strict;
use warnings;

print "This 'print' statement gets parsed successfully but never executed";

BEGIN {
        print "This gets printed first";
}

print "This, also, is parsed successfully but never executed";

...because e4h8v3oitv8h4o8gch3o84c3 there is a huge parsing error down here.
```

ولأن كتل **BEGIN** تنفذ في زمن الترجمة؛ فإنها لو وضعت داخل عبارة كتلة شريطة ستبقى تنفذ اولاً، حتى لو كانت نتيجة الشرط خطأً على الرغم من حقيقة أن نتيجة الشرط قد لا تكون حُسبت أصلاً بعدُ، بل – في الواقع – قد لا تحسب أبداً.

```
if(0} (
        BEGIN}
                print "This will definitely get printed;"
        {
        print "Even though this won't;"
```

لا تضع كتل **BEGIN** في شروط! إذا أردت فعل شيءٍ ما مشروطاً في زمن الترجمة؛ فضع الشرط داخل كتلة:**BEGIN**

```
BEGIN}
        if($condition} (
                #etc.
        {
{
```

Use:

حسناً. والآن بما أنك صرت تدرك اختلاف سلوك ودلالات الرزم والوحدات ووظائف الصفوف وكتل **BEGIN**؛ يمكنني شرح التابع المضمن الشائع للغاية **use**.

هذه العبارات الثلاث التالية:

```
use Caterpillar ("crawl", "pupate;("

use Caterpillar;()

use Caterpillar;
```

مساوية على الترتيب لهذه:

```
BEGIN}

        require Caterpillar;

        Caterpillar->import("crawl", "pupate;("

{

BEGIN}

        require Caterpillar;

{
```

```
BEGIN}

        require Caterpillar;

        Caterpillar->import;()

{
```

- لا، ليست الأمثلة الثلاثة في الترتيب الخطأ. كل ما في الأمر أن **Perl** حمقاء.
- ستدعاء **use** تمثيلٌ لكتلة **BEGIN**. فاحذر هنا مما مر التحذير به هناك. يجب وضع عبارات **use** في أعلى الملف دوماً، ولا توضع أبداً داخل شروط.
- **Import**(= ليس تابعاً مضمناً في **Perl**، إنما وظيفة صف معرّفة. يقع العبء على عاتق مبرمج الرزمة **Caterpillar** لتعريف أو وراثة **import**()، ويمكن للوظيفة نظرياً أن تقبل أي شيء كوسطاء وتفعل أي شيء بهذه الوسطاء. **use Caterpillar;** يمكن أن تفعل أي شيء. راجع وثائق **Caterpillar.pm** لمعرفة ما سيحدث بالضبط.
- لاحظ كيف يحمّل **require Caterpillar** الوحدة المسماة **Caterpillar.pm**، بينما يستدعي **Caterpillar->import**() الإجراء الفرعي **import**() المعرّف في الرزمة **Caterpillar package**. لنرجو أن تتحد الوحدات والرزم يوماً ما!

Exporter:

أكثر الطرق شيوعاً لتعريف الوظيفة import() أن ترثها من الوحدة Exporter.

فـــExporter وحدةٌ أساسيةٌ وميزة جوهرية أيضاً من ميزات لغة البرمجة Perl. في تطبيق import()، المأخوذ من Exporter تفسير قائمة الوسطاء التي تمررها كقائمة من أسماء الإجراءات الفرعية وعندما يضاف إجراء فرعي بـــ import()؛ يصبح متاحاً في الرزمة الحالية تماماً كرزمته الأصلية.

أسهل طريقة لإدراك هذا المفهوم استخدام مثال. وإليك كيف يبدو Caterpillar.pm:

```
use strict;
use warnings;

package Caterpillar;

# الوراثة من Exporter
use parent ("Exporter");

sub crawl  { print "inch inch";  }
```

```
sub eat    { print "chomp chomp"; }

sub pupate { print "bloop bloop"; }


our @EXPORT_OK = ("crawl", "eat");

return 1;
```

يجب أن يحوي متغير الرزمة @EXPORT_OK قائمة من أسماء الإجراءات الفرعية.

يمكن بعد ذلك لقطعة أخرى من الكود أن تضيف بـــimport() هذه الإجراءات الفرعية من أسمائها، باستخدام عبارة use عادةً:

```
use strict;

use warnings;

use Caterpillar ("crawl");


crawl(); # "inch inch"
```

في هذه الحالة، الرزمة الحالية هي main، إذاً الاستدعاء crawl() يستدعى في الحقيقة إلى main::crawl()، الذي (بسبب أنه قد أضيف) يشير إلى Caterpillar::crawl().

ملاحظة: بغض النظر عن محتويات @EXPORT_OK؛ فإن كل وظيفة يمكن دوماً أن تستدعى بالكتابة العادية:

```
use strict;

use warnings;
```

```
use Caterpillar (); # لم تسم أية إجراءات فرعية، ولم تحصل أية استدعاءات
import()


# وأيضاً...
Caterpillar::crawl();  # "inch inch"
Caterpillar::eat();    # "chomp chomp"
Caterpillar::pupate(); # "bloop bloop
```

لا تملك **Perl** وظائفاً خاصة. وعادةً ما تسمى الوظائف المعدة للاستخدام الخاص بسابقة شرطة أو شرطتين تحتيتين.

@EXPORT:

وحدة المصدر **Exporter** تعرّف أيضاً متغير رزمة يدعى **@EXPORT** والذي يمكن أن يُزَوِّد بقائمة بأسماء الإجراءات الفرعية.

```
use strict;
use warnings;


package Caterpillar;


# وراثة من Exporter
use parent ("Exporter");


sub crawl  { print "inch inch";   }
sub eat    { print "chomp chomp"; }
sub pupate { print "bloop bloop"; }


our @EXPORT = ("crawl", "eat", "pupate");
```

```
return 1;
```

تستخرج الإجراءات الفرعية المذكورة في @EXPORT إذا استدعي import() بدون وسائط على الإطلاق، وهذا ما يحصل هنا:

```
use strict;

use warnings;

use Caterpillar استدعاء # ;importمن غير وسائط()


crawl();  # "inch inch"

eat();    # "chomp chomp"

pupate(); # "bloop bloop"
```

لكن لاحظ الحال التي صرنا إليها هنا؛ إذ فقدنا الدلائل مما قد يصعب معرفة ما إذا كان crawl()عُرّف أصلاً. والعبرة من هذه القصة في نقطتين:

- عند إنشاء وحدة تستخدم Exporter، لا تستخدم أبداً @EXPORT لاستخراج الإجراءات الفرعية افتراضياً. واجعل المستخدم دوماً يستدعي الإجراءات الفرعية "بكتابة عادية" أو بإضافتهم بـــimport() تخصيصاً (باستخدام use Caterpillar "crawl"" مثلاً)، مما يشير بقوة إلى البحث في Caterpillar.pm عن تعريف crawl().
- عند استخدام وحدة بـــuse وهذه الوحدة تستخدم Exporter؛ خصص دوماً أسماء الإجراءات الفرعية التي تريد إضافتها بـــimport(). وإذا كنت لا تريد إضافة

(**import**()) أية إجراءات فرعية، وتريد الإشارة إليها بالكتابة العادية؛ يجب كتابة قائمة فارغة خاصة: **use Caterpillar** ().

النهايـــــــة:20-08-2015

إعداد:معاذ مباركي

الفهرس	
مقدمة وتعريف بسيط للغة	03
مقدمة عن لغة البرمجة **prel**	05
أهلاً بالعالم – **Hello World**:	08
المتغيرات:	09
المتغيرات الحجمية:	10
"المنطقيات":	11
النمط الهشّ:	12
متغيرات المصفوفات:	13
المتغيرات الدلالية:	16
القوائم:	18
السياق في **prel**	22
المراجع وبُنى المعطيات المتداخلة	24
التصريح عن بنية معطيات	26
الوصول للمعلومات في البُنى:	31
كيف تؤذي نفسك من غير أن تشعر مستخدماً المراجع إلى للمصفوفات	34
الشروط:**else / elsif / if /**	35
الحلقات	38
إنشاء مصفوفات جديدة من قديمة:	44
التوابع المضمنة	48