

P3rL WaRRiOr Of Th3 PaSt C3NtUriEs

P3rL SpEcial VariablEs

Perl Sub Routines

P3rL References

P3rL Packages

P3rL ObjEct

P3rL SpEcial VariabLEs

توفر لغة البيزل للمستخدم الذي يتعامل معها إضافة الى أنواع المتغيرات التي يتم التعامل معها وفق رغبة المستخدم تقدم البيزل نوعا آخر من المتغيرات وهذا النوع هو المتغيرات الخاصة أو ما يعرف بـ `perl` **special Variable** أن الذي يميز هذا النوع من المتغيرات عن النوع الاخر من المتغيرات التي يقوم بها المستخدم بالتعامل معها هو أن هذه المتغيرات تكون متميزة بنقطتين هما أولا :- أن الاسماء الخاصة بهذه المتغيرات هي اسماء ثابتة لا تتغير ثانيا :- ان الوظيفة البرمجية التي تقوم بها هذه المتغيرات هي وظيفة ثابتة في أي مكان في البرنامج او في المقطع البرمجي والمتغيرات الخاصة في لغة البيزل هي

1-

**Code(1)*

```
$_
```

يتم التعامل مع المتغير في الجدول أعلاه على أنه المتغير الافتراضي في لغة البيزل حيث يتم التعامل في أغلب حالات التعبيرات القياسية ويتم اعتباره المتغير الافتراضي لأغلب الدوال البرمجية ويكون التمثيل البرمجي الخاص بهذا المتغير الخاص كما يلي من خلال المقطع البرمجي الاتي

**Code(2)*

```
$_="perl is programming language";
s/perl/PERL/;
print $_;
```

ويكون ناتج تنفيذ هذا المقطع أعلاه كما يلي من خلال الشكل ادناه

PERL is programming language

Figure(1)

2-

**Code(3)*

```
@_
```

المتغير البرمجي أعلاه يتم استعمال هذه المصفوفة بصورة عامة في برمجة الروتينات الفرعية حيث يتم استعمال هذه المصفوفة لكي يتم تمرير المتغيرات الى الروتين الفرعي ويكون التمثيل البرمجي الخاص بهذه المصفوفة هو كالاتي من خلال المقطع أدناه

**Code(4)*

```
sub spawn {
  @_ = ($a)
  ...
  ..
  .
}
```

وسيتم التطرق الى عمل هذه المصفوفة لاحقا في القسم الخاص ببرمجة الروتينات الفرعية في لغة البيزل

3-

**Code(5)*`&`

يكون التمثيل البرمجي الخاص بهذا المتغير كما يلي من خلال المقطع البرمجي الاتي

**Code(6)*

```
"spawn is perl programmer" =~ /perl/;
print &,"\\n";
```

إذا ما تم تنفيذ هذا المقطع البرمجي فإن الناتج من عملية تنفيذه سوف تكون كما يلي من خلال الشكل الاتي

```
perl
```

Figure(2)

يوضح الشكل أعلاه ناتج تنفيذ المقطع البرمجي المتعلق بالمتغير الخاص سابق الذكر

4-

**Code(7)*`'`

يكون التمثيل البرمجي الخاص بهذا النوع من المتغيرات في لغة البيرل كما يلي في المقطع البرمجي الاتي

**Code(8)*

```
"spawn is perl programmer" =~ /perl/;
print $',"\\n";
```

و الآن اذا ما تم تنفيذ المقطع أعلاه فإن الناتج من عملية التنفيذ سيكون كما يلي من خلال الشكل التوضيحي الاتي

```
programmer
```

Figure(3)

تكون الوظيفة البرمجية التي يقوم بها هذا النوع من المتغيرات الخاصة هي انه بعد انه يتم تحديد الكلمة في التعبير القياسي لكي يتم مطابقتها كما حصل في المثال السابق هذا المتغير يكون دوره في إعطاء المستخدم هي ماهي الكلمة التي تقع بعد الكلمة التي تمت مطابقتها وفي هذه الحالة الكلمة التي تقع بعد كلمة perl الكلمة programmer

5-

**Code(9)*```

يكون التمثيل البرمجي لهذا المتغير في لغة البيرل كما يلي من خلال المقطع البرمجي الاتي

**Code(10)*

```
"spawn is perl programmer" =~ /perl/;
print $`,"\\n";
```

و الآن اذا تم تنفيذ المقطع اعلاه فإن الناتج من عملية التنفيذ سوف يكون كما يلي في الشكل التوضيحي أدناه

```
spawn is
```

Figure(4)

تكون الوظيفة البرمجية التي يقوم بها هذا المتغير هي أظهار ذلك الجزء من السلسلة النصية الذي يسبق الجزء

المطلوب مطابقته وفي هذه الحالة يكون الجزء الذي يسبق الكلمة المراد مطابقتها هي كلمة spawn is

6-

*Code(11)

```
$+
```

يكون التمثيل البرمجي لهذا النوع من المتغيرات في لغة البيزل كما يلي من خلال المقطع البرمجي أعلاه وكما يلي

*Code(12)

```
"spawn is perl programmer" =~ /(perl) | (PERL);  
print $+;
```

والان اذا تم تنفيذ المقطع البرمجي اعلاه فان الناتج من عملية تنفيذه سوف تكون



Figure(5)

تكون الوظيفة البرمجية التي يتم بهذا المقطع البرمجي اعلاه هي انه يعمل على مقارنة القيم التي تم اعطائها في خانة التعبير القياسي مع كلمة او حرف او رمز في السلسلة النصية المقابلة التي يتم استخدامها ولكن الغاية البرمجية من هذا المتغير هي عندما يكون المبرمج غير متأكد من اي تعبير قياسي سوف تتم مطابقته في السلسلة يكون دور هذا المتغير في حل هذه المشكلة واذا ما تمت ملاحظة السلسلة النصية في المقطع البرمجي اعلاه فانه perl في السلسلة النصية على النسق الصغير اي ال LowerCase لذا عندما تم طباعة المتغير الخاص تم طباعة كلمة ال perl بالنسق الصغير.

7-

*Code(13)

```
@+
```

يكون التمثيل البرمجي لهذا المتغير في لغة البيزل كما يلي من خلال المقطع البرمجي الآتي

*Code(14)

```
"spawn is perl programmer" =~ /s/;  
print @+;
```

عندما يتم تنفيذ المقطع البرمجي الآتي فان الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الآتي



Figure(6)

ان الاسلوب البرمجي الذي تتبعه هذه المصفوفة هو انه تعمل على اعطاء موقع الحرف الموجود في السلسلة النصية بعد أن يتم مقارنته بالحرف او الرمز الذي تم اعطائه في التعبير القياسي وأن موقع الحرف في السلسلة النصية هو هو الموقع (1) اي الموقع الاول.

8-

**Code(15)*

```
$.
```

يكون التمثيل البرمجي العام لهذا المتغير الخاص في لغة البيزل كما يلي من خلال المقطع البرمجي الاتي

**Code(16)*

```
open (FH,"/home/spawn/aa");
while (<FH>){
print $.;
print "\n";
}
```

الان اذا تنفيذ المقطع البرمجي اعلاه فأن الناتج من عملية التنفيذ كما يلي من خلال الشكل التوضيحي الاتي

```
1
2
3
```

Figure(7)

تكون الوظيفة البرمجية التي يقوم بها هذا المتغير هي انه يعمل على عد كافة الاسطر التي يحتويها الملف الذي يتم التعامل معه اي انه الملف الذي في المقطع البرمجي اعلاه هو عبارة عن ملف نصي مكون من 3 اسطر فقط

```
spawn is perl programmer
storm is c programmer
striky is python programmer
```

Figure(8)

9-

**Code(17)*

```
$\
```

يكون التمثيل البرمجي لهذا النوع من المتغيرات في لغة البيزل كما يلي من خلال المقطع البرمجي الاتي

**Code(18)*

```
$_="-";
print $a="perl";
print $b="c";
print $c="python";
```

يكون ناتج تنفيذ المقطع البرمجي اعلاه كما يل من خلال الشكل التوضيحي الاتي

```
perl-c-python-
```

Figure(9)

تكون الوظيفة البرمجية التي يقوم بها هذا المتغير الخاص هي كالاتي حيث ان هذا المتغير يقوم بمهمة وهي **فصل روابط المخرجات**

وذلك من خلال فاصلة تحدد من قبل المستخدم وكما هو ملاحظ من المقطع البرمجي السابق كانت الفاصلة هي ال(-)

10-

**Code(19)*

```
$/
```

يكون التمثيل البرمجي لهذا النوع من المتغيرات في لغة البيرل كما يلي من خلال المقطع البرمجي الآتي

**Code(20)*

```
open(FH,"/home/spawn/aa");
while(<FH>){
print $_,$/;
}
```

وإذا تنفيذ المقطع البرمجي أعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي في الشكل الآتي

```
spawn is perl programmer
storm is c programmer
striky is python programmer
```

New lines added
By the Special
Variable

Figure(10)

تكون الوظيفة البرمجية التي يقوم بها هذا المتغير هي انه يعمل على اضافة سطر جديد بين كل سطر حيث ان الملف الذي تم استعماله في المقطع البرمجي أعلاه هو عبارة عن ملف مكون من 3 أسطر وعندما تم استعمال هذا المتغير تم اضافة سطر جديد بين كل سطر وشكل الملف الاصل هو كالاتي

```
spawn is perl programmer
storm is c programmer
striky is python programmer
```

Figure(11)

الملف قبل استعمال هذا المتغير يكون مكون من 3 أسطر وبعد استعمال المتغير اصبح 6 أسطر

11-

**Code(21)*

```
$/,
```

يكون التمثيل البرمجي لهذا المتغير هو كما يلي من خلال المقطع البرمجي الآتي

**Code(22)*

```
$/,="-";
print "C","perl","python";
```

عندما يتم تنفيذ هذا المقطع فأن الناتج من عملية تنفيذه تكون كالاتي في الشكل التوضيحي المدرج أدناه

```
C-perl-python
```

Figure(12)

تجدد الإشارة الى أن فعالية هذا المتغير تكون فقط على جملة الطباعة اي فقط على جملة ال `print`

12-

**Code(23)*

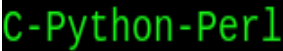
```
$"
```

يكون التمثيل البرمجي لهذا المتغير يكون كما يلي من خلال المقطع البرمجي الاتي

**Code(24)*

```
$"="-";
@prog=("C","Python","Perl");
print "@prog";
```

والان اذا تم تنفيذ المقطع البرمجي اعلاه فان الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي الاتي



Figure(13)

13-

**Code(25)*

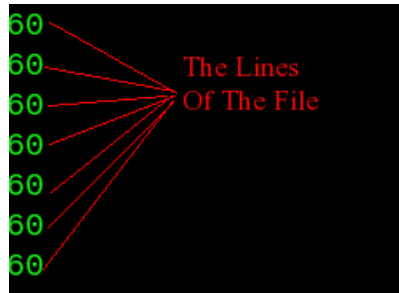
```
$=
```

يكون التمثيل البرمجي لهذا المتغير الخاص في لغة البيزل هو كما يلي من خلال المقطع البرمجي الاتي

**Code(26)*

```
open(FH,"/home/spawn/aa");
while (<FH>){
print $=,"\\n";
}
```

أما الآن اذا تم تنفيذ المقطع البرمجي أعلاه فان الناتج من عملية التنفيذ تكون كما يلي من خلال الشكل التوضيحي الاتي



Figure(14)

تكون الوظيفة البرمجية التي يقوم بهذا المتغير الخاص في لغة البيزل هي انه يقوم بعرض طول الصفحة والرقم 60 هو الحالة الافتراضية للمتغير

وإذا ما تم القاء نظرة على الملف الذي يحمل الاسم aa سيلاحظ ان تركيبه هو كما يلي في الشكل أدناه


```

1 spawn is perl programmer
2 storm is c programmer
3 striky is python programmer
4 |
5 striky is python programmer
6 storm is c programmer
7 spawn is perl programmer

```

Figure(15)

يلاحظ ان الملف اعلاه هو فعلا يتكون من سبعة اسطر فقط لذا كان عدد الاسطر التي تحمل الرقم 60 في ال 14 كان سبعة أسطر فقط

14-

*Code(27)

```
$?
```

يكون التمثيل البرمجي لهذا المتغير في لغة البيزل كما يلي من خلال المقطع البرمجي أدناه

*Code(28)

```
system "ls -A /home/spawn & konsolee";
print "The Error is:-", $?,"\\n";
```

عندما يتم تنفيذ المقطع البرمجي اعلاه فان الناتج من عملية التنفيذ يكون كما يلي من خلال الشكل التوضيحي أدناه

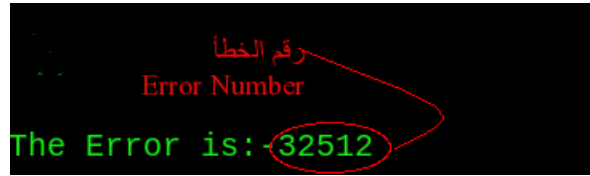
```

sh: konsolee: command not found
aa          .directory  .gimp-2.4    .marble     .opera      .thumbnails
aa~         .dmrc       .gnome2     .mdk-menu-migrated perl         tmp
.adobe     Documents  .gnome2_private .MdkOnline  Pictures     v
.bash_history Download  .gstreamer-0.10 .mixxxbpm scheme.xml pro         War
.bash_logout .esd_auth  .gtk-bookmarks .mixxx.cfg  .pulse      .Xauthority
.bash_profile Flash      .gvfs       .mixxxtrack.xml .pulse-cookie .xine
.bashrc     .fontconfig .kde4       Movies      .qt         .xournal
.cache     .fonts     .kderc     .mozilla    .recently-used .xsession-errors
.config    .fonts.conf .kino-history .mplayer    .recently-used.xbel
.dbus     .gconf     .kinorc    Music       .screenrc
Desktop   .gconfd    .local     .ooo3      Templates
The Error is:-32512

```

Figure(16)

من الممكن ملاحظة انه في المقطع البرمجي الخاص بهذا المتغير تم استعمال دالة ال `system` وتم استعمال مدخلين معها وكانت قيمة المدخل صحيحة و المدخل الاخر كان ذو قيمة خاطئة لذا عندما تم التنفيذ كان المدخل الاول صالح لذا تم تنفيذه وتم الحصول على ناتج هذا الامر ولكن المدخل الثاني كونه متغير خاطئ هنا يدخل دور هذا المتغير كونه متغير خاطئ فقد قام هذا المتغير باعطاء المستخدم رقم الخطأ الخاص بهذه العملية التي تم تنفيذها



Figure(17)

15-

*Code(29)

\$)

يكون التمثيل البرمجي لهذا النوع من المتغيرات في لغة البيرل كما يلي من خلال المقطع البرمجي أدناه

*Code(30)

```
print "Hi i am a linux user and my GID is ",$, "\n";
```

والان اذا قام المستخدم بتنفيذ السطر البرمجي اعلاه فأنالنتاج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي الاتي

500 500

Figure(18)

تكون الوظيفة البرمجية التي يقوم بها السطر البرمجي اعلاه هي انه يظهر Effective Group Id للمستخدم عندما يقوم بتنفيذ هذا السطر البرمجي

16-

*Code(30)

\$(

تكون الوظيفة البرمجية التي يقوم بهذا المتغير الخاص اعلاه هي انه يقوم بإظهار ال Real Group ID للعملية الحالية التي يتم العمل عليها

17-

*Code(31)

\$\$

يكون التمثيل البرمجي لهذا المتغير في لغة البيرل هو كما يلي المقطع البرمجي الاتي

*Code(32)

```
print "The current PID of this process is: -", $$, "\n";
```

عندما يقوم المستخدم بتنفيذ المقطع السطر البرمجي فأن الناتج من عملية التنفيذ تكون كما يلي من خلال الشكل التوضيحي الاتي

5066

Figure(19)

عندما يقوم المستخدم بتنفيذ السطر البرمجي اعلاه فان الناتج من عملية التنفيذ سوف تكون اظهار رقم فقط وهذا الرقم هو معرف العملية او process id (pid) للعملية التي يتم العمل عليها

18-

**Code(33)*`$<`

يكون التمثيل البرمجي لهذا المتغير في لغة البيزل كما يلي من خلال المقطع البرمجي أدناه

**Code(34)*`print "The real UID of this process is:-", $<, "\n";`

عندما يتم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ ستكون كما يلي من خلال الشكل الآتي



Figure(20)

يقوم هذا المتغير عندما يتم تنفيذه بأظهار ال real uid للمستخدم

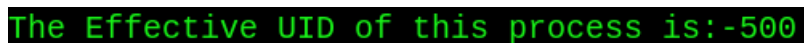
19-

**Code(35)*`$>`

التمثيل البرمجي لهذا النوع من المتغيرات يكون كما يلي من خلال المقطع البرمجي الآتي

**Code(36)*`print "The Effective UID of this process is:-", $>, "\n";`

و عندما يتم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي في الشكل أدناه



Figure(21)

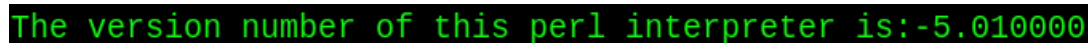
20-

**Code(37)*`$];`

يكون التمثيل البرمجي لهذا المتغير في لغة البيزل كما يلي من خلال المقطع البرمجي أدناه

**Code(38)*`print "The version number of this perl interpreter is:-", $];, "\n";`

ناتج تنفيذ هذا المقطع اعلاه هو كما يلي من خلال الصورة أدناه



Figure(22)

21-

**Code(39)*`$^O`

يكون التمثيل البرمجي لهذا المتغير في لغة البيزل كما يلي من خلال المقطع البرمجي ادناه

*Code(40)

```
print "This version copy of perl interpreter is running under $^O operating system";
```

عندما يتم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل أدناه

```
This version copy of perl interpreter is running under linux operating system
```

Figure(23)

تكون الوظيفة البرمجية التي يقوم بها المتغير أعلاه هي انه يقوم بذكر اسم نظام التشغيل الذي يتم العمل عليه وقد وفي هذه الحالة كان النظام هو Linux

22-

*Code(41)

```
$^W
```

التمثيل البرمجي لهذا المتغير في لغة البيزل يكون كما يلي من خلال المقطع التالي

*Code(42)

```
print "the current warring status now is $^W", "\n";
```

الان لو تم تنفيذ هذا المقطع البرمجي فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

```
the current warring status now is 0
```

Figure(24)

*Code(43)

```
print "the current warring status now is $^W", "\n";
```

الان لو تم تنفيذ السطر البرمجي اعلاه فان الناتج من عملية التنفيذ سوف تكون كما يلي في الشكل ادناه

```
the current warring status now is 1
```

Figure(25)

من الامور التي تجذب الانتباه بخصوص عمل هذا المتغير هو انه تم تنفيذه مرتين وفي كل من المرتين التي تم تنفيذه فيها تم اعطاء نتيجة مختلفة السبب في ذلك هو ان هذا المتغير يحمل نوعين من القيم

1. (0)
2. (1)

الحالة الاولى : عندما يظهر للمستخدم الرقم 0 عند التنفيذ فهذا يعني ان وضع التحذير غير مفعّل
الحالة الثانية : عندما يظهر للمستخدم الرقم 1 عند التنفيذ فهذا يعني ان وضع التحذير قد تم تفعيله
ملاحظة:-

يتم تفعيل وضع التحذيرات في لغة البيزل كما يلي

*Code(44)

```
perl -w
```

23-

*Code(45)

```
@INC
```

يعتبر هذا المتغير من المتغيرات المعروفة في لغة البيرل ويكون تمثيله البرمجي كما يلي من خلال المقطع البرمجي أدناه

**Code(46)*

```
print @INC,"\n";
```

ناتج تنفيذ هذا السطر البرمجي هو كما يلي من خلال الشكل أدناه

```
/usr/lib/perl5/site_perl/5.10.0/x86_64-linux-thread-multi/usr/lib/perl5/site_perl/5.10.0/u  
sr/lib/perl5/vendor_perl/5.10.0/x86_64-linux-thread-multi/usr/lib/perl5/vendor_perl/5.10.0  
/usr/lib/perl5/5.10.0/x86_64-linux-thread-multi/usr/lib/perl5/5.10.0/usr/lib/perl5/site_pe  
rl/usr/lib/perl5/vendor_perl.
```

Figure(26)

هذا الایعاز البرمجي عندما يتم تنفيذه يقوم بأظهار المسارات التي تحتوي على مقاطع البيرل البرمجية وغالبية هذه المقاطع هي المقاطع البرمجية التي تحتوي على

1. do
2. use
3. require

Perl Sub Routines

تعتبر لغة البيرل من اللغات التي توفر تقنية الروتينات الفرعية المعرفة من قبل المستخدم وهذه الروتينات الفرعية من الممكن ان يتم برمجة الروتين الفرعي في أي مكان من البرنامج الرئيسي ومن الممكن ان يتم ومن الممكن ان يتم جلب الروتينات الفرعية من ملفات اخرى عن طريق الدوال

1. *do*
2. *use*
3. *require*

ومن الممكن ان يتم استدعاء الروتين الفرعي اكثر من مرة داخل البرنامج الرئيسي وفي كل مرة سوف يتم استدعاءه فيها سوف يقوم بنفس المهمة التي يقوم بها اذ من الممكن ان يتم اعتبار الروتينات الفرعية هي

user defined function

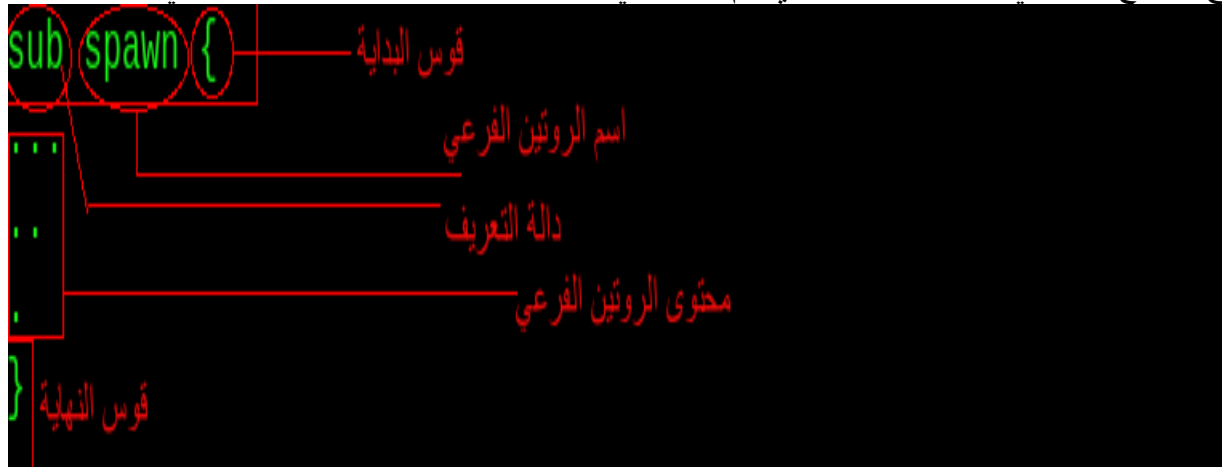
Sub Routine declaration

يكون التعريف العام للروتينات الفرعية في لغة البيرل كما يلي من خلال المقطع البرمجي ادناه الذي يوضح الالية التي على المستخدم اتباعها فيما لو اراد ان يقوم بعلمية تعريف لروتيني فرعي

*Code(47)

```
sub spawn {
...
..
.
}
```

يوضح المقطع البرمجي اعلاه الاسلوب الذي يتم أتباعه في لغة البيرل من أجل كتابة روتين فرعي بسيط



Figure(27)

The Basic rule of perl sub routine

1-

sub

يجب ان يتم استعمال الدالة التي تحمل الاسم sub فهذه الدالة هي الدالة المسؤولة عن تعريف الروتين الفرعي وفي كل مرة يرغب مبرمج بيرل ان يعرف روتين فرعي عليه ان يبدأ بهذه الدالة

2-

XXX(name)

بعد أن ان يتم استعمال دالة التعريف الخاص بتعريف الروتين الفرعي على المبرمج في هذه الحالة اختيار اسم للروتين الفرعي الذي سيتم العمل عليه

3-

{

القوس الذي يشير الى انتهاء الفقرة التعريفية وبعد كتابة القوس سوف يبدأ العمل على المحتوى البرمجي لهذا الروتين الفرعي

4-

CONTEXT

محتوى الروتين الفرعي اي جسم الروتين البرمجي

5-

}

انتهاء الروتين الفرعي

The First routine

المقطع البرمجي أدناه سوف يكون ايسر روتين فرعي في لغة البيرل

*Code(48)

```
sub perl {
print "HeLlO World \n";
}
&perl();
```

الان لو تم تنفيذ المقطع البرمجي اعلاه فان الناتج من عملية التنفيذ سوف تكون كما يلي

HeLlO world

Figure(28)

How to execute

يكون تنفيذ الروتينات الفرعية في لغة البيرل بالطريقة الاتية
 أولا :- يتم كتابة علامة (&) في بداية سطر التنفيذ
 ثانيا :- يتم كتابة اسم الروتين الفرعي بعد علامة (&) كخطوة ثانية
 ثالثا :- يتم كتابة الاقواس المزدوجة الفارغة و عندها عند الانتهاء من هذا سيتم تنفيذ البرنامج

Arguments

توفر لغة البيزل طرق لكي يتم تمرير المتغيرات او المعاملات الى الروتين الفرعي الطريقة الاولى

**Code(49)*

```
$a,$b=(10,15);
sub perl {
if ($a > $b) {
print " A is bigger than B\n";
}
else {
print "B is bigger than A\n";
}
}
&perl();
```

الان لو يتم تنفيذ المقطع البرمجي المذكور اعلاه ان الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي ادناه

B is bigger than A

Figure(29)

توفر لغة البيزل أمكانية التعامل مع متغيرات تكون هذه المتغيرات من موقع خارج الروتين الفرعي اي في داخل الكتلة الرئيسية للبرنامج ولغة البيزل لا تمنع الروتين الفرعي من التعامل مع المتغيرات خارج النص

Private Variables in perl subroutine

يلاحظ من المقطع البرمجي السابق ان المتغيرات التي تم تعريفها في المقطع البرمجي هي عبارة عن متغيرات تم تعريفها خارج جسم الروتين الفرعي أي ان المتغيرات عرفت في جسم البرنامج الرئيسي وهذا يعني ان هذه المتغيرات متاحة في البرنامج في اي مكان لذا كان لابد من اتباع طريقة لتعريف المتغيرات وهذه المتغيرات التي يتم تعريفها هي متغيرات خاصة بالروتين الفرعي وهذه العملية تتم كما يلي من خلال المقطع البرمجي الآتي

**Code(50)*

```
$a= "We Are Perl Programmers";
sub perl {
my ($a,$b) = (10,15);
($a,$b) = (@_);
if ( $_[0] > $_[1] ) {
print "A is bigger than B\n";
}
else {
print print "B is bigger than A\n";
}
}
&perl();
print $a,"\n";
```


الآن عندما يتم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي أدناه

```
B is bigger than A
1We Are Perl Programmers
```

Figure(30)

يلاحظ من المقطع البرمجي اعلاه انه هذا البرنامج مكون من روتين فرعي يحمل الاسم `perl` واطرافه الروتين الفرعي يوجد في البرنامج متغير يحمل الاسم `$a` وله قيمة لغة البيرل تتعامل مع المتغير الاول الواقع خارج نطاق الروتين الفرعي بصورة مستقلة عن المتغير الاخر الذي يحمل الاسم `$a` والذي يقع داخل كتلة الروتين الفرعي هذه العملية تتم من خلال استعمال الدالة التي تحمل الاسم

*Code(51)

```
my
```

```
$a= "We Are Perl Programmers";
sub perl {
my ($a,$b) = (10,15);
($a,$b) = (@_);
if ( $_[0] > $_[1] ) {
print "A is bigger than B\n";
}
```

Figure(31)

أن الدالة المسؤولة عن خصخصة المتغيرات في داخل الروتين الفرعي وجعلها تعود للروتين الفرعي فقط هي دالة `my` وهكذا لو تم استعمال جملة الطاعة مع المتغير `$a` فإن القيمة التي سوف يقوم بطباعتها هي جملة `we are perl programmers`

The Other Way

ذكر في الصفحات السابقة ان الطريقة التي يتم تنفيذ روتين فرعي هي كما يلي

*Code(52)

```
sub perl {
print "Hello World\n";
}
&perl();
```

وتكون النتيجة هي كما يلي في الشكل أدناه

```
Hello World
```

Figure(32)

```
sub perl {
print "Hello World\n";
}
&perl();
```

الفقرة البرمجية المسؤولة عن تنفيذ الروتين الفرعي

Figure(33)

وضحت الطريقة السابقة اسلوب تنفيذ الروتينات الفرعية في لغة البيرل باستخدام الطريقة التقليدية التي تم التطرق اليها ولكن من الممكن ان يتم تنفيذ الروتين الفرعي بطريقة اخرى وهي

*Code(53)

```
sub perl {
print "Hello World\n";
}
perl;
```

عندما يتم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي

```
Hello World
```

Figure(34)

اي ان الروتين الفرعي من الممكن ان يتم تنفيذه لو قام المبرمج بالغاء كل من الاقواس المزدوجة () وعلامة ال& وعندما يتم التنفيذ سيتم الحصول على نفس الناتج اي يتم تنفيذ الروتين الفرعي بصورة صحيحة ولكن تجدر الاشارة الى ان وجود او عدم وجود علامة ال& في الروتين الفرعي ليس اختياريا كما قد يظن المبرمج في هذه الحالة لانه توجد بعض الحالات التي يكون وضع هذه العلامة اي علامة ال& ضروري جدا لكي يتم تنفيذ الروتين الفرعي بالصورة الصحيحة ومن هذه الطرق الطريقة الاتية

*Code(54)

```
$a="perl programmer";
$b= (length($a));
print $b,"\n";
sub length {
print "Hello world through perl subroutine\n";
}
&length;
```

الان لو تم تنفيذ الروتين الفرعي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل أدناه

```
15
Hello world through perl subroutine
```

Figure(35)

```

$a="perl programmer";
$b= (length($a));
print $b, "\n";
sub length {
print "Hello world through perl subroutine\n";
}
&length;

```

هذه الـ length تمثل الدالة المبنية داخل لغة البيزل

اسم الروتين الفرعي ولا يمت بالدالة المذكورة أعلاه بعلاقة

الفقرة البرمجية المسؤولة عن تنفيذ الروتين الفرعي الذي يحمل الاسم length

Figure(36)

وضح الشكل اعلاه أن لغة البيزل تملك القدرة على التعامل مع هكذا نوع من المواقع البرمجية حيث ان الدالة الاولى التي تحمل الاسم

*Code(55)

length

هي الدالة البرمجية التي تكون مسؤولة عن حساب الطول للسلسلة النصية التي تحمل الاسم

*Code(56)

perl programmer

وعندما يتم تنفيذ هذه الفقرة يتم حساب طول السلسلة النصية اعلاه اما بالنسبة للروتين الفرعي الذي يحمل الاسم

*Code(57)

length

فأنه عبارة عن روتين فرعي بسيط تكون الوظيفة البرمجية التي يقوم بها هو طباعة سلسلة نصية اما بالنسبة لطريقة التنفيذ ففي هذه الحالة يكون مبرمج البيزل مجبرا على استخدام العلامة & في هذه الحالة لانه لو لم يتم باستخدامها في هذه الحالة فأن لغة البيزل سوف تتعامل مع الموقف كما لو ان المبرمج في هذه الحالة يريد ان يقوم بعملية استدعاء لدالة الطول ولكن عندما يتم وضع العلامة & فأن لغة البيزل سوف تكون على علم انه المبرمج في هذه الحالة يتعامل مع روتين فرعي بهذا الاسم

Anonymous Sub Routine

لغة البيزل توفر نوع اخر من الروتينات الفرعية وهذا النوع من الروتينات الفرعية كما هو معنون أعلاه تدعى بالروتينات المجهولة او الروتينات الخالية من الاسم وذلك لان هذه الروتينات فعلا تحمل اي اسم لها ويكون التمثيل البرمجي لهذا النوع من الروتينات الفرعية في لغة البيزل كما يلي من خلال المقطع البرمجي أدناه

*Code(58)

```

$perl = sub {
print "We Are Perl Programmers"
};
&$perl;

```

الآن عندما يتم تنفيذ المقطع البرمجي اعلاه فأن الناتج من المقطع البرمجي اعلاه سوف يكون كما يلي من خلال

الشكل التوضيحي ادناه

We Are Perl Programmers

Figure(37)

```
$perl = sub {
print "We Are Perl Programmers"
};
&$perl;
```

المتغير الذي يحمل قيمة الروتين الفرعي
البرمجية

الطريقة المتبعة في تنفيذ الروتين

Figure(38)

هكذا يكون التمثيل البرمجي للروتينات الفرعية المجهولة في لغة البيرل ولكن على المبرمج ان يلاحظ الفقرة التالية وهي وجود الفارزة المنقوطة بعد الانتهاء من كتابة البرنامج ويجب وضعها بعد إغلاق قوس الانتهاء وبدونه لن يتم تنفيذ البرنامج

*Code(59)

```
$perl = sub {
print "We Are Perl Programmers"
}
&$perl;
```

وكناتج لهذه العملية الناتج سوف يكون كما يلي

```
$perl = sub {
print "We Are Perl Programmers"
}
&$perl;
```

لم يتم وضع الفارزة المنقوطة

```
[spawn@localhost ~]$
```

Figure(39)

يوضح الشكل أعلاه أنه في هذه الحالة لم يتم وضع الفارزة المنقوطة لذا في هذه الحالة عندما تم تنفيذ البرنامج كانت المحصلة هي عدم الحصول على النتيجة المرغوبة والمطلوب من الروتين الفرعي أن يقوم بها

The return function

يتم استعمال دالة ال `return` كدالة تعيد عنصر او مصفوفة في الروتين الفرعي ومن الممكن ان يتم تمثيلها برمجيا كما يلي من خلال المقطع البرمجي أدناه

*Code(60)

```
@perl = perl (10,15);
print "@perl";
sub perl {
my ($a,$b) = (10,15);
($a,$b) = (@_);
$c=$a+$b;
return ($a,$b,$c)
}
```

إذا تم تنفيذ المقطع البرمجي أعلاه فإن الناتج سوف يكون كما يلي من خلال الشكل التوضيحي أدناه

10 15 25

Figure(40)

يحتوي السطر الاول من البرنامج على مصفوفة برمجية تحمل الاسم

*Code(61)

```
@perl
```

وتكون هذه المصفوفة تتعامل بصورة مباشرة مع الروتين الفرعي الذي يحمل الاسم

*Code(62)

```
perl
```

```
@perl = perl (10,15);
print "@perl";
sub perl {
my ($a,$b) = (10,15);
($a,$b) = (@_);
$c=$a+$b;
return ($a,$b,$c)
}
```

الروتين الفرعي المسمى perl
يتم استعماله كدالة مع الأرقام 10,15

Figure(41)

اما عن الوظيفة البرمجية التي يقوم بها الروتين الفرعي هي انه يتعامل مع متغيرين ومن ثم يتم تمريرهم الى مصفوفة تمرير العناصر في لغة البيزل ومن ثم يتم التعامل مع متغير ثالث يقوم بجمع العنصرين عندها يتم استدعاء دالة

*Code(63)

```
return
```

التي تعيد قيم المتغيرات الثلاثة كما قد وضح الشكل الذي يحمل الرقم 40

ومن الممكن ان يتم أستعمال دالة الارجاع return بطريقة أخرى وهذه الطريقة متمثلة كما يلي من خلال المقطع البرمجي أدناه

**Code(64)*

```
sub perl {  
return "Hello everyone.\nHere We Are Perl Programmers\n";  
}  
print &perl;
```

وناتج تنفيذ المقطع البرمجي أعلاه كما يلي في الشكل الآتي

```
Hello everyone.  
Here We Are Perl Programmers
```

Figure(42)

Using the caller() Function in Subroutines

من خلال هذه التقنية أعلاه فعندها من الممكن ان يكون في متناول المبرمج معلومات كاملة عن الروتين الفرعي الذي يتم التعامل معه و التمثيل البرمجي لهذه العملية يكون كما يلي من خلال المقطع البرمجي أدناه

*Code(65)

```
use Data::Dumper;
sub Info {
my @info = caller(0);
print Dumper \@info;
}
Info;
```

و عندما يتم تنفيذ المقطع البرمجي اعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

```
$VAR1 = [
    'main',
    '- ',
    6,
    'main::Info',
    1,
    undef,
    undef,
    undef,
    256,
    '',
    undef
];
```

Figure(43)

الشكل البرمجي أعلاه يقدم معلومات عن الروتين الفرعي الذي يتم التعامل معه و المعلومات التي يغطيها هي

1-

\$package

package namespace عندما تم الاستدعاء

2-

\$filename

اسم الملف الذي تم الاستدعاء منه بعبارة اخرى لو كان المقطع البرمجي اعلاه مخزون في ملف عند التنفيذ سيتم عرض اسم الملف المخزون فيه

3-

\$line

رقم السطر الذي يحمل الفقرة البرمجية عن تنفيذ الروتين الفرعي

4-

\$subroutine

اسم الروتين الفرعي الذي يتم التعامل معه

5-

\$hasargs

هل يحتوي الروتين الفرعي الذي يتم التعامل معه على متغيرات

6-

\$wantarray

المصفوفة = 1

المتغير الذي يكون من نوع Scalar = 0void = غير معرفة

7-

\$evaltext

إذا كان المقطع البرمجي هو عبارة عن () eval block

8-

\$is_requireسوف تكون القيمة true لو كان قد صنع بواسطة use أو require

More Details

من الممكن ان يتم تطبيق التقنية السابقة مع اضافة بعض التفاصيل الاضافية الاخرى وهي كما يلي من خلال المقطع البرمجي أدناه

**Code(66)*

```
package spawn;
use Data::Dumper;
sub Info {
my @info = caller(0);
print Dumper \@info;
}
Info;
```

وألان اذا تم تنفيذ المقطع البرمجي اعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التالي


```
$VAR1 = [
    'spawn',
    '-',
    7,
    'spawn::Info',
    1,
    undef,
    undef,
    undef,
    256,
    '',
    undef
];
```

Figure(44)

الجزء الاول من الصورة والذي يحمل المسمى

*Code(67)

spawn

يشير الى اسم الحزمة التي يحتويها الملف وستتم مناقشة هذه الفقرة لاحقا

تجدد الاشارة الى انه لو تم تنفيذ البرنامج من برنامج الصدفة بالطريقة الاتية

*Code(68)

perl sub.pl

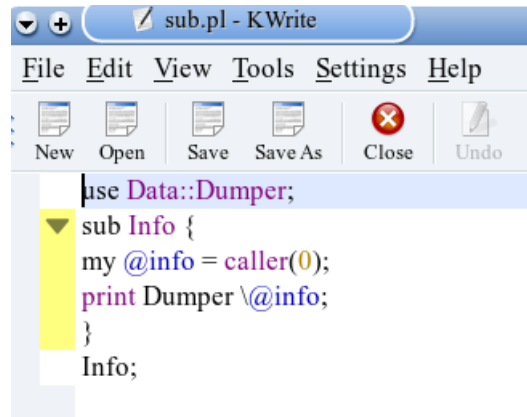
عندها سوف يلاحظ المبرمج حدوث تغيير في ناتج الامر وهذا التغيير هو الاتي

```
$VAR1 = [
    'main',
    'sub.pl',
    6,
    'main::Info',
    1,
    undef,
    undef,
    undef,
    256,
    '',
    undef
];
```

Figure(45)

الفقرة الواقعة في المستطيل الاحمر تشير الى اسم الملف الذي تم استدعاء الملف منه وهو الملف الذي يحمل الاسم *Code(69)

```
sub.pl
```



Figure(46)

كما تجدر الاشارة الى فقرة اخرى في الحالة التي يتم تنفيذ البرنامج بها من خلال الصدفة هي انه يتم اعتماد الحزمة الافتراضية هي كحزمة العمل وليس الحزمة التي يحتويها البرنامج

Local Function

تحتوي لغة البيزل على دالة اخرى تكون هذه الدالة من الدوال التي لها اسلوب خاص بالتعامل مع المتغيرات التي تحتويها الروتينات الفرعية والدالة التي يتم التكلم عنها الان هي الدالة التي تحمل الاسم

*Code(70)

```
local
```

ويكون التمثيل البرمجي لهذه الدالة كما يلي من خلال المقطع البرمجي الاتي

*Code(71)

```
my $x = 10;
$_ = "perl";
{
my $x = 20;
local $_ = "python";
perl();
}
perl();
sub perl{
print "\$x is $x\n";
print "\$_ is $_\n";
}
```

والان عندما يتم تنفيذ المقطع البرمجي اعلاه فان الناتج من عملية التنفيذ تكون كما يلي من خلال الشكل الاتي

```
$x is 10
$_ is python
$x is 10
$_ is perl
```

Figure(47)

أوجز الشكل التوضيحي أعلاه الكيفية التي تم من خلالها استعمال الدالة `undef` الذكر وتكون الوظيفة البرمجية التي تقوم بها هذه الدالة هي إعطاء قيمة برمجية للمتغير في داخل الروتين الفرعي الذي يعود إليه أي القيمة التي يحملها هذا المتغير داخل الذي تم استعمال دالة `local` معه هي قيمة وقتية فقط و الشكل اعلاه وضح الكيفية وعلى المبرمج دوما ان يقوم باستعمال دالة `my` لان هذه الدالة تعتبر من اسرع واكثر أمانا من الدالة السابقة

When to use local and my

يذكر المبرمج Mark-Jason Dominus طريقة فعالة وهذه الطريقة هي

Don't use local. Always use my.

Figure(48)

Passing References to a Subroutine

تعتبر الطريقة اعلاه هي الطريقة التي من خلالها يتم تمرير الـ `reference` في لغة البيزل الى الروتين الفرعي ولاحقا في الكتاب سيتم مناقشة ما هي تقنية الـ `reference` في لغة البيزل يتم تمثيل هذه الطريقة في لغة البيزل كما يلي ومن خلال المقطع الاتي

*Code(72)

```
my $a = 5;
perl(\$a);
print $a;
sub perl {
print $$reference;
}
```

عندما يتم تنفيذ المقطع البرمجي اعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي

5

Figure(49)

لا يوجد شيء مميز او خاص بالطريقة السابقة الذكر ولكن تعتبر اسلوب برمجي اخر من الاساليب التي تحتويها لغة البيزل التي يتم من خلالها التعامل مع البيانات تجدر الإشارة

```

my $a = 5;
perl(\$a);
print $a;
sub perl {
print $$reference;
}

```

الفقرة البرمجية الخاص بتقنية الـ `perl` وكيفية تمثيلها في لغة البيزل

Figure(50)

Passing Arrays to subroutine

توفر تقنية الروتينات الفرعية في لغة البيزل للمبرمج إمكانية التعامل مع المصفوفات ويكون أسلوب تمرير المصفوفات في لغة البيزل كما يلي من خلال المقطع البرمجي الآتي

*Code(73)

```

$message = "Testing";
@count = ("python", "perl", "C");
passing ($message, @count);
sub passing {
$message = shift;
print "@_";
}
testing;

```

عندما يتم تنفيذ المقطع البرمجي أعلاه فان الناتج من عملية التنفيذ أعلاه سوف تكون كما يلي من خلال الشكل الآتي

```
python perl C
```

Figure(51)

يحتوي المقطع البرمجي أعلاه على نوعين من المتغيرات وكما هو موضح في المقطع البرمجي ألبرمجي انه يحتوي

1-Scalar

2-array

وتم معاملة المتغير الاول مع الدالة `shift` لكي يتم أخذ القيمة التي تحتويها وهي قيمة سلسلة نصية ومن ثم يتم طباعة مصفوفة المتغيرات اي المصفوفة الافتراضية في داخل الروتين الفرعي وهكذا عندما تتم عملية الطباعة سيتم عرض كافة العناصر التي تحتويها المصفوفة تعتبر هذه الطريقة من افضل و اسهل الطرق التي يتم من خلالها تمرير المصفوفات الى الروتين الفرعي

Passing hashes to subroutine

توفر لغة بيرل للمبرمج إمكانية التعامل مع المتغيرات التي تكون من نوع الهاش وتقنية الروتينات الفرعية في لغة بيرل توفر للمبرمج إمكانية التعامل مع هذا النوع من المتغيرات في لغة بيرل ويكون التمثيل البرمجي لهذا النوع من تمرير المتغيرات كما يلي من خلال المقطع البرمجي الآتي

*Code(74)

```
%hash = (
  Spawn    =>  "perl",
  Storm    =>  "C",
  Striker   =>  "Python",
);
$message = "perl";
testing ($message,%hash);
sub testing {
($message, %count) = @_ ;
print "@_","\n";
}
```

والآن عندما يتم تنفيذ الروتين الفرعي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي الآتي

```
perl Spawn perl Striker Python Storm C
```

Figure(52)

التقنية التي تم أتباعها مع هذا النوع من المتغيرات هي الطريقة ذاتها التي تم أتباعها مع المصفوفات

Prototypes

ال prototypes في لغة بيرل هي ليست ذات ال prototypes التي تحتويها اللغات البرمجية حيث يتم استخدام ال prototypes في اللغات البرمجية الأخرى من أجل تعريف الاسم ونوع المعاملات الى الروتين الفرعي ولكن تكون ال prototypes في لغة بيرل تعطي للمبرمج إمكانية تعريف نوع المعاملات التي سيتم تمريرها الى الروتين الفرعي اي من خلال ال prototypes في لغة بيرل يكون الروتين الفرعي على معرفة بنوعية المتغيرات التي سيكون عليه التعامل معها ويكون التمثيل البرمجي لهذه التقنية البرمجية كما يلي من خلال المقطع البرمجي الآتي

*Code(75)

```
sub perl ($$) {
  if (@_ == 2) {
    print "Good,Two Arguments" ,"\n";
  }
  else {
    print "Bad,Any number of args but not two", "\n";
  }
}
perl(10,15);
```

عندما يتم تمثيل المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الآتي

Good,Two Arguments

Figure(53)

```

sub perl ($$) {
  if (@_ == 2 ) {
    print "Good,Two Arguments" , "\n";
  }
  else {
    print "Bad,Any number of args but not two", "\n";
  }
}
perl(10,15);
Good,Two Arguments

```

الطريقة التي يتم بها تعريف الprototype في لغة البيرل

Figure(54)

الاسلوب البرمجي الذي تم أتباعه من قبل لغة البيرل في هذه التقنية هي أن المقطع البرمجي أعلاه تمت برمجته لكي يقوم باستقبال متغيرين فقط وهذين المتغيرين هما لا بد ان يكونان على

1-Scalar

وأذا تم ادخال اكثر من متغيرين فعندما يتم تنفيذ المقطع البرمجي سوف يتم الحصول على خلل برمجي لأن الروتين الفرعي أعلاه تمت برمجته لكي يقوم بالتعامل مع متغيرين فقط تجدر الاشارة

*Code(76)

```

sub perl ($$) {
  if (@_ == 2 ) {
    print "Good,Two Arguments" , "\n";
  }
  else {
    print "Bad,Any number of args but not two", "\n";
  }
}
perl("perl","python");

```

عندما تم تصميم المقطع البرمجي اعلاه لكي يستقبل متغيرين فقط فهذا لايعني ان نوع المتغيرات التي يتعامل معها هي لا بد ان تكون متغيرات رقمية اذ من الممكن ان يتم التعامل مع متغيرات ذات قيمة نصية وعندما يتم تنفيذ المقطع البرمجي اعلاه فان الناتج من عملية التنفيذ سوف تكون كما يلي

Good,Two Arguments

Figure(55)

P3rL References

يمكن تعريف تقنية الـ `reference` في لغة الـ `C` باعتبار الـ `reference` هو قطعة من البيانات تخبر المبرمج عن موقع قطعة أخرى من البيانات في لغة الـ `C` يكون دائما عبارة عن

*Code(77)

```
scalar
```

دائما تكون الـ `reference` في لغة الـ `C` على الصيغة البرمجية أعلاه ولكن هذا لا يعني ان البيانات التي يشير إليها يجب أن تكون بهذه الصيغة اللغات البرمجية الأخرى مثل لغة الـ `C++` تحتوي على نفس هذه التقنية ولكن تحت مسمى آخر وهو مسمى المؤشرات أو `pointers`

How To make reference

يتم صناعة الـ `reference` في لغة الـ `C` من خلال استعمال الرمز

*Code(78)

```
|
```

حيث يتم وضع العلامة أعلاه قبل المتغير المطلوب

Reference to scalar

توفر تقنية الـ `reference` في لغة الـ `C` امكانية التعامل مع المتغيرات التي تكون من النوع أعلاه ويكون تمثيل هذه العملية برمجيا كما يلي

*Code(79)

```
$a="programming-fr34ks";
$a_ref=$a;
```

1-

الخطوة البرمجية الأولى هي عبارة عن متغير برمجي عادي يحمل قيمة سلسلة نصية

2-

الخطوة البرمجية الثانية هي عبارة عن متغير يحمل قيمة `reference` للقيمة الأولى ويتم تنفيذ المقطع البرمجي أعلاه لكي يتم الوصول الى القيمة البرمجية المخزونة في داخل المتغير الثاني كما يلي من خلال المقطع البرمجي أدناه

*Code(80)

```
$a="programming-fr34ks";
$a_ref=$a;
print $a_ref;
```

يكون ناتج تنفيذ المقطع البرمجي أعلاه عندما يتم التنفيذ كما يلي من خلال الشكل التوضيحي الآتي

SCALAR(0x1b2b140)

Figure(56)

*Code(81)

```
$a="programming-fr34ks";
$a_ref=$a;
print $$a_ref;
```

أما الآن إذا تم تنفيذ المقطع البرمجي أعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي الآتي

```
programming-fr34ks
```

Figure(57)

```
$a="programming-fr34ks";
$a_ref=\$a;
print $$a_ref;
programming-fr34ks
```

الجزء البرمجي المسؤول عن تنفيذ القيمة التي يحملها المتغير الذي تم اسناده الى الـ \$a_ref

Figure(58)

وضح المقطع البرمجي كيفية التعامل مع تقنية الـ \$a_ref واسلوب اسنادها الى المتغيرات البرمجية التي تكون من نوع

*Code(82)

```
scalar
```

Reference to arrays

تقنية الـ \$a_ref في لغة البيزل تمكن المبرمج الذي يتعامل معها على امكانية التعامل مع المتغيرات التي تكون من نوع المصفوفات ويكون التمثيل البرمجي للتعامل تقنية الـ \$a_ref مع المتغيرات التي تكون من نوع المصفوفات كما يلي من خلال المقطع البرمجي الآتي

*Code(83)

```
@array = ("Striker","Storm","Spawn");
$array_ref= \@array;
print $$array_ref[0],"\n";
```

والآن عندما يتم تنفيذ المقطع البرمجي أعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي الآتي

```
Striker
```

Figure(59)

```
@array = ("Striker","Storm","Spawn");
$array_ref= \@array;
print $$array_ref[0],"\n";
Striker
```

الجزء البرمجي المسؤول عن تنفيذ القيمة التي يحملها المتغير الذي تم اسناده الى الـ \$array_ref

Figure(60)

Reference to hashes

توفر تقنية الـ *reference* في لغة البيزل للمبرمج إمكانية التعامل مع المتغيرات التي تكون من نوع الـ

**Code(84)*

```
hash
```

ويكون التمثيل البرمجي لهذه التقنية في لغة البيزل كما يلي من خلال المقطع البرمجي الآتي

**Code(85)*

```
%hash = (
  Storm      => "C programmer",
  Striker    => "python programmer",
  Spawn      => "perl programmer",
);
$hash_ref = \%hash;
print $$hash_ref {"Storm"};
```

والآن عندما يتم تنفيذ المقطع البرمجي اعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي

C programmer

Figure(61)

```
%hash = (
  Storm      => "C programmer",
  Striker    => "python programmer",
  Spawn      => "perl programmer",
);
$hash_ref = \%hash;
print $$hash_ref {"Storm"};
```

C programmer

الجزء البرمجي المسؤول عن تنفيذ القيمة التي يحملها المتغير الذي تم أسناده الى الـ *reference*

Figure(62)

Anonymous data

في لغة البيزل يتم تقسيم هذا الموضوع الى عدة اقسام والقسم الاول الاول من هذا الموضوع هو

1-

Anonymous array Composer

يكون التمثيل البرمجي لهذه التقنية كما يلي من خلال المقطع البرمجي الاتي

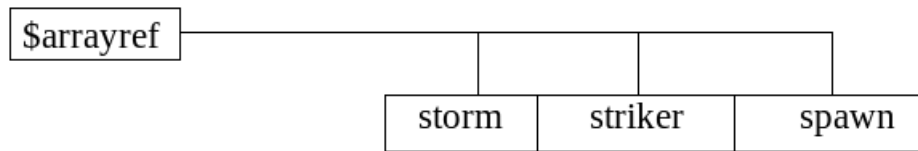
*Code(86)

```
$arrayref = [storm, striker,spawn];
print $arrayref->[0];
```

والآن اذا تم تنفيذ المقطع البرمجي أعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي



Figure(63)



Figure(64)

عندما يكون المبرمج في حاجة الى ان يقوم بتعريف رفرنس الى مصفوفة مجهولة يتم استعمال الدوال المربعة الشكل في عملية برمجتها وعملية استدعاء او طباعة العناصر التي تحتويها يتم من خلال استعمال اسم المتغير ومن ثم السهم مع موقع العنصر المطلوب في المصفوفة كما وضح المقطع البرمجي السابق

More complex Stuff

*Code(87)

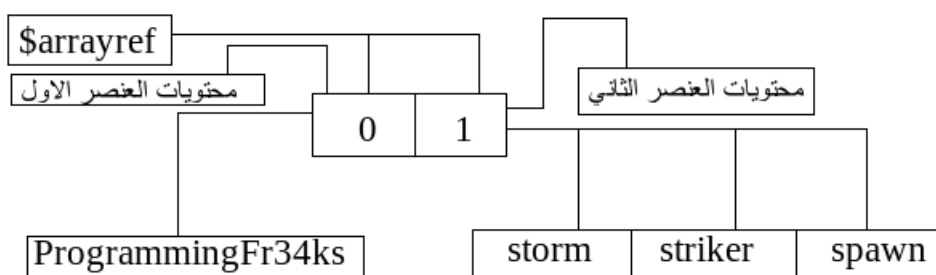
```
$arrayref = [ProgrammingFr34ks,[storm, striker,spawn]];
print $arrayref->[1][2];
```

الآن عندما يتم تنفيذ المقطع البرمجي أعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي



Figure(65)

ويكون الاسلوب الذي تتعامل معه لغة البيزل مع هذا المقطع البرمجي الذي تم وضعه في الاعلى كما يلي من خلال المقطع التخطيطي الاتي



Figure(66)

في المقطع البرمجي أعلاه كان العنصر الاول الذي يحتويه هو عبارة عن عنصر عادي ولكن النقطة التي قد تسبب بعض الارتباك هي في العنصر الثاني هي يعتبر العنصر الثاني هو عبارة عن عنصر في مصفوفة ولكن يتم التعامل معه على اعتبار هو رفرنس لمصفوفة مجهولة وقد وضح المقطع السابق الاسلوب الخاص بطباعة العناصر التي تكون من هذا النوع

2-

Anonymous Hash composer

يكون التمثيل البرمجي لهذه التقنية في لغة البيرل كما يلي من خلال المقطع البرمجي أعلاه

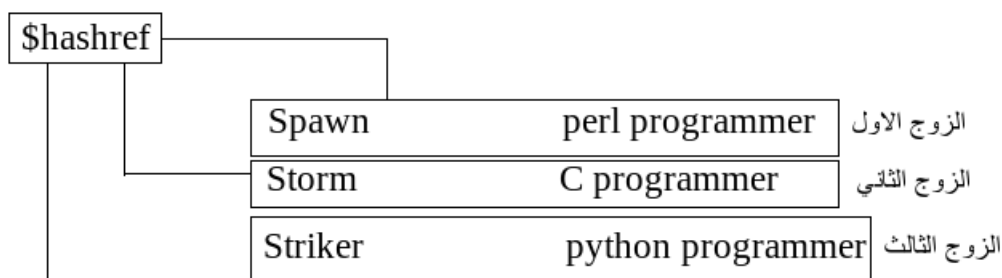
*Code(88)

```
$hashref = {
    Spawn => "perl programmer",
    Storm  => "C programmer",
    Striker => "python programmer",
};
print $hashref->{Spawn};
```

والان عندما يتم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال من الشكل التوضيحي الاتي

```
perl programmer
```

Figure(67)



Figure(68)

A pseudo-hash

توفر نوع آخر من المركبات البيانية (data structure) ويكون التمثيل البرمجي الخاص بها يكون كما يلي من خلال المقطع البرمجي الاتي

*Code(89)

```
$pseudo_hash = [ {a=>1,b=>2,c=>3}, "val a", "val b", "val c" ];
print $pseudo_hash->[1];
print "\n";
```

يكون الناتج من عملية تنفيذ المقطع البرمجي أعلاه كما يلي من خلال الشكل الاتي

```
val a
```

Figure(69)

وضح الجزء الاول من المقطع البرمجي الكيفية التي يتم من خلالها الوصول الى العناصر التي يحتويها المركب البياني لكن هذا الجزء قد وضح الكيفية التي يتم من خلالها الوصول الى العناصر التي محتواة في هيكل المصفوفة و التي تبدأ من العنصر الثاني حيث أن العنصر الاول من المركب البياني اعلاه له اسلوب خاص للوصول الى البيانات التي يحتويها و المقطع البرمجي ادناه يوضح الطريقة التي يتم من خلالها الوصول الى العنصر الاول من المركب البياني اعلاه

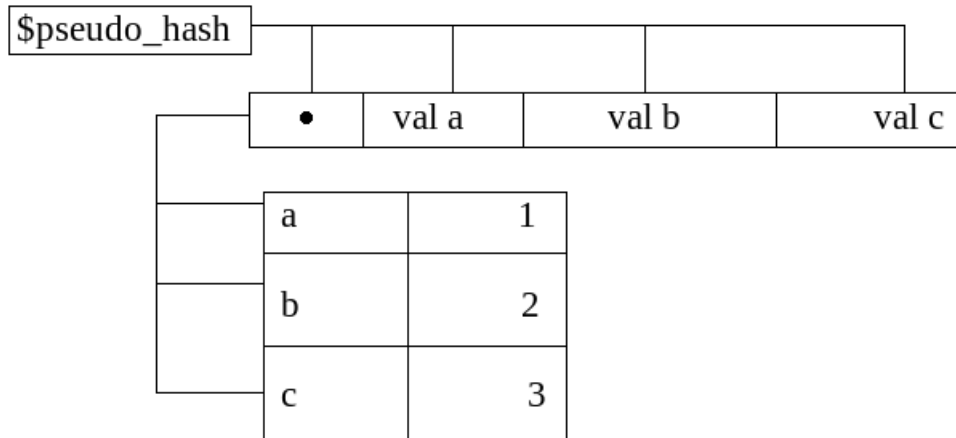
*Code(90)

```
$pseudo_hash = [ {a=>1,b=>2,c=>3}, "val a", "val b", "val c" ];
print ($pseudo_hash->[0]->{"a"});
print "\n";
```

الان عندما يتم تنفيذ المقطع البرمجي يكون ناتج التنفيذ كما يلي من خلال الشكل الاتي

```
1
```

Figure(70)



Figure(71)

أن اهم ما يميز المركب البياني اعلاه هو ان العنصر الاول الذي يحتويه لابد ان يكون متغير من نوع الهاش اي ان المبرمج اذا كان يرغب في برمجة هذا النوع من المركبات البيانية ففي هذه الحالة لابد ان يكون العنصر الاول هو هاش اما العناصر الاخر فهي تكون عناصر مصفوفة عادية ولكن العنصر الاول لابد من ان يكون هاش

Storing Data Structures

توفر لغة البيزل للمبرمج إمكانية تخزين التراكيب البيانية التي يتم العمل عليها حيث يمكن لمبرمج البيزل أن يقوم بخزن التراكيب البيانية على ملفات لكي يتم تسهيل العمل عليها وتتم هذه العملية من خلال استعمال احدى الموديلات البرمجية التي تقوم بهذه المهمة

*Code(91)

```
Storable
```

ويكون التمثيل البرمجي لهذه العملية كما يلي من خلال المقطع البرمجي أدناه

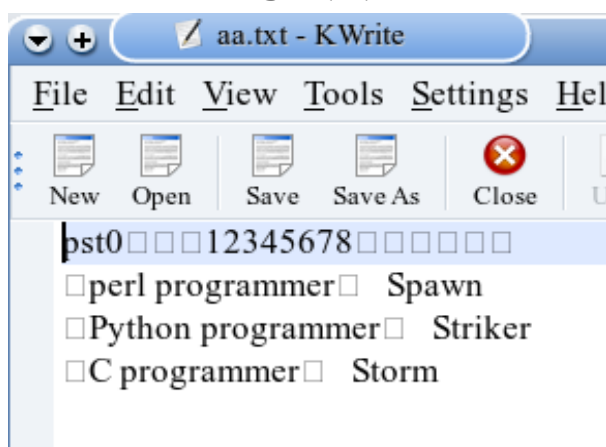
*Code(92)

```
use Storable;
%hash = (
    Spawn    => "perl programmer",
    Storm    => "C programmer",
    Striker  => "Python programmer",
);
$hash_ref = \%hash;
store (\%hash, "/home/spawn/aa.txt");
```

الان عندما يتم تنفيذ المقطع البرمجي أعلاه يكون ناتج التنفيذ كما يلي من خلال الشكل التوضيحي أدناه



Figure(72)



Figure(73)

عندما يتن تنفيذ المقطع البرمجي اعلاه فأن الناتج يكون كما يلي من خلال الشكلين السابقين حيث ان الشكل الاول هو عبارة عن شكل توضيحي يخبر المبرمج من خلاله ان الان هو بصدد التعامل مع ملف ثنائي و الشكل الثاني هو عبارة عن الشكل الذي يخبر المبرمج بمحتوى الملف الذي تم خزن الهاش في داخله

References to Subroutines

تمكن لغة البيزل المبرمج من ان يتعامل مع المصادر والروتينات الفرعية حيث من الممكن ان يتم اخذ مرجع لروتين فرعي ويكون التمثيل البرمجي لهذه العملية كما يلي من خلال المقطع البرمجي الآتي

*Code(93)

```
sub perl {
print "Here we Go \n";
}
$perl_ref = \&perl;
```

المقطع البرمجي أعلاه يوضح الاسلوب البرمجي الذي يتم أتباعه من قبل لغة البيزل في أخذ مرجع او رفرنس لروتين فرعي

وطريقة الاستدعاء لهذه التقنية تمثل برمجيا في لغة البيزل كما يلي من خلال المقطع البرمجي اعلاه

*Code(94)

```
sub perl {
print "Here we Go \n";
}
$perl_ref = \&perl;
&{$perl_ref};
$perl_ref->();
$perl_ref->($_);
```

يكون ناتج تنفيذ المقطع البرمجي أعلاه كما يلي من خلال الشكل الاتي

```
Here we Go
Here we Go
Here we Go
```

Figure(74)

```
1 sub perl {
2 print "Here we Go \n";
3 }
4 $perl_ref = \&perl;
5 &{$perl_ref};
6 $perl_ref->();
7 $perl_ref->($_);
```

الطريقة الاولى لعملية الاستدعاء

الطريقة الثانية لعملية الاستدعاء

الطريقة الثالثة لعملية الاستدعاء

Figure(75)

وضح المقطع البرمجي أعلاه الطريقة التي يتم من خلالها تنفيذ الروتين الفرعي الذي تم معاملته مع الـ `ref` و يوضح الشكل السابق الطريقة التي يتم من خلالها عمل الاستدعاءات و المبرمج في هذه الحالة له الحرية في اختيار الاسلوب الخاص بعملية الاستدعاء التي يرغب بها

P3rL Packages

الحزم في لغة البيزل هي عبارة عن مقطع برمجي يتم التعامل معه من خلال ال `namespace` الخاصة به يكون التعريف العام للحزم في لغة البيزل من خلال استعمال دالة برمجية من الدوال المبنية في لغة البيزل حيث ان هذه الدالة تقوم بهذه العملية وهذه الدالة هي

*Code(95)

```
Package
```

هذه الدالة هي الدالة المسؤولة عن اخبار لغة البيزل انها الان بصدد التعامل مع حزمة برمجية ويكون التمثيل البرمجي للحزم في لغة البيزل كما يلي من خلال المقطع البرمجي الاتي

*Code(96)

```
package perl;
$pe = "We are perl programmer\n";
print $pe;
```

الان عندما يتم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

```
we are perl programmer
```

Figure(76)

*Code(97)

```
$pe = "programming-fr34ks\n";
print $pe;
package perl;
$pe = "We are perl programmer\n";
print $pe;
```

الان اذا تم تنفيذ المقطع البرمجي اعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي الاتي

```
programming-fr34ks
we are perl programmer
```

Figure(77)

المتغير الاول من البرنامج هو عبارة عن متغير رئيسي تابع للبرنامج العام اي متغير `main` أما بالنسبة الى المتغير الثاني فهو عبارة عن متغير خاص يعود الى الحزمة التي تحمل الاسم بيزل أن متغيرات الحزم في لغة البيزل من الممكن ان يتم الوصول او الولوج اليها من خلال الحزم الاخرى وذلك بأستخدام الاسم الكامل للحزمة

Using 'strict' Variables

بما أنه لغة البيزل تمنح المبرمج امكانية كبيرة في تعريف متغير من النوع العام اي متغير `global` فإنه كان لابد من توفير حل اخر بديل يسمح للمتغيرات بالاحتفاظ بخصيتها وهنا كان الحل في استخدام الموديل الذي يحمل الاسم

*Code(98)

```
strict
```

ويكون التمثيل البرمجي لهذا الموديل في لغة البيزل كما يلي من خلال المقطع البرمجي الاتي

*Code(99)

```
use strict;
package perl;
$package_variable = "This variable is in ", 'MyPackage';
```

الان عندما يتم تنفيذ المقطع البرمجي اعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

```
Global symbol "$package_variable" requires explicit package name at - line 3.
Execution of - aborted due to compilation errors.
```

Figure(78)

والحل لهذه المشكلة التعريفية تكمن في استعمال الدالة

*Code(100)

```
my
```

*Code(101)

```
use strict;
package perl;
my $package_variable = "This variable is in ", 'MyPackage';
```

الان عندما يتم تنفيذ البرنامج سوف يلاحظ المبرمج اختفاء الرسالة التي تشير الى حدوث خطأ اثناء التنفيذ وذلك بسبب استعمال الدالة انفة الذكر

Declaring Global Package Variables

المعنى التقليدي للمتغير الذي يحمل النمط `global` هو أن يكون ممكن للمبرمج أن يراه من كافة اجزاء البرنامج أن المتغير يكون عادة متغير `global` في داخل الحزمة التي تم تعريفه فيها وهذا يعني أنه من الممكن لأي متغير في الحزمة ان يصل الى هذه المتغيرات من دون الحاجة الى الاستدعاء الكامل لها ولغة البيزل تتعامل مع متغيرات لانه الحزم على أنها المتغيرات ال `global` الحقيقية لانه من الممكن الولوج اليها من اي مكان من البرنامج فقط باستعمال الاسم الكامل لها

والطريقة الابسط لكي يتم كتابة متغير حزمة في لغة البيزل هي كما يلي ان يتم كتابة اسم المتغير بصورة كاملة كما يلي من خلال المقطع البرمجي الاتي

*Code(102)

```
package perl ;
$perl_pack = "We are perl programmers\n";
print $perl::perl_pack;
```

```

package perl ;
$perl_pack = "We are perl programmers\n";
print $perl::perl_pack;

```

اسم المتغير الموجود في داخل الحزمة

اسم الحزمة

```

We are perl programmers

```

Figure(79)

Declaring Global Package Variables with 'use vars'

توفر لغة البيزل للمبرمج إمكانية تعريف المتغيرات التي سوف يتعامل معها في المصفوفة وذلك من خلال استعمال

*Code(103)

```
use vars
```

ويكون التمثيل البرمجي لهذه التقنية كما يلي من خلال المقطع البرمجي الآتي

*Code(104)

```
use vars qw($perl_pack,@array_pack,$anotherpack);
```

وهذه الطريقة تعمل على تعريف المتغيرات في الحزمة التي سيتم العمل عليها وهذا يعني أن هذه المتغيرات مرئية في مكان من الحزمة ومن الممكن أن يتم الوصول إليها بصورة مباشرة من خلال تخصيص اسم المتغير كاملاً إضافة إلى اسم الحزمة

*Code(105)

```

sub perl {
use vars qw($pack_var);
$pack_var = "We are perl programmers";
}
print $pack_var;
perl;
print $pack_var;

```

والآن عندما يتن تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي الآتي

```
We are perl programmers
```

Figure(80)

```

sub perl {
use vars qw($pack_var);
$pack_var = "We are perl programmers";
}
print $pack_var;
perl;
print $pack_var;

```

لغة البيزل تتعامل مع هذا المتغير على أنه متغير "مرئي" ولكنه غير معرف

لغة البيزل الآن تتعامل مع هذا المتغير على أنه متغير "مرئي" والآن قد تم تعريفه

Figure(81)

Lexical Variables

ال Lexical Variables هي متغيرات يتم تعريفها في داخل Lexical scope وهذه المتغيرات من غير الممكن يتم التعامل معها خارج نطاق المدى الذي تم تعريفها فيه

*Code(106)

```
use strict;
use warnings;
{
my $speak = 'moo';
}
print "speak is '$speak'\n";
```

الآن عندما يتم تنفيذ المقطع البرمجي أعلاه ان الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الآتي

Global symbol "\$speak" requires explicit package name at - line 6.
Execution of - aborted due to compilation errors.

Figure(82)

هذه المتغيرات من غير الممكن أن يتم التعامل معها من خارج النطاق الذي تم تعريفها فيه وهذه المتغيرات من غير الممكن أن يتم اضافتها ال symbol table الخاص بالحزم لهذا فإنه من غير الممكن أن يتم الوصول إليها وهذا ما يلاحظ من خلال المقطع البرمجي أعلاه حيث ان ناتج تنفيذه يخبر المبرمج أن لغة البيزل تتعامل مع المتغير الموجود في الأقواس المغلقة {} هو متغير هو موجود خارج الأقواس تجدر الإشارة إلى ما يأتي بخصوص ال lexical variables

أولاً : ال lexical variables لا تعود إلى أي حزمة برمجية لذا فإن من غير الممكن ان يتم اعتبارها انها تنتمي إلى حزمة برمجية معينة
ثانياً : من الممكن الوصول إلى هذه المتغيرات فقط في داخل البلوك البرمجي الذي تم تعريفها فيه وخارج نطاق هذا البلوك لغة البيزل تتعامل مع هذا المتغير على أنه متغير غير موجود أصلاً

The main package

*Code(107)

```
$perl = "We are perl programmers";
print $perl, "\n";
```

المقطع البرمجي أعلاه يعتبر مقطع برمجي عادي وبسيط وعندما يتم تنفيذه فإن الناتج من عملية التنفيذ تكون طباعة قيمة المتغير والتي هي عبارة عن سلسلة نصية عادية ولكن الأمر في هذه الحالة هو أوسع من هذا بقليل بسبب انه لغة البيزل تتعامل مع المقطع البرمجي أعلاه على اعتبار انه حزمة !! وهذا النوع من الحزم يتم اعتباره من الحزم الافتراضية من الممكن ان يتم كتابة المقطع البرمجي أعلاه بصورة أخرى مختلفة ولكن مع اعطاء الناتج ذاته

*Code(108)

```
$perl = "We are perl programmers";
print $main:::perl, "\n";
```

الآن اذا تم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي

```
We are perl programmers
```

Figure(83)

```
$perl = "We are perl programmers";
print $main::perl, "\n";
We are perl programmers
```

الحزمة الافتراضية في لغة البيزل
default package

Figure(84)

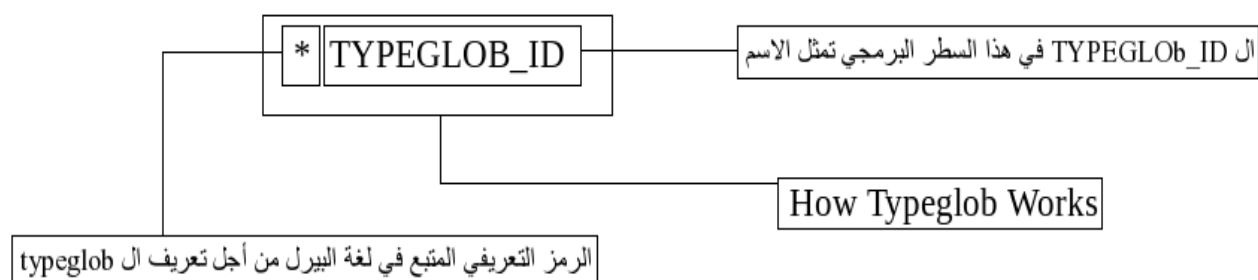
في الحالات البرمجية المشابهة للحالة السابقة فإن الحالة الافتراضية التي توفرها لغة البيزل هي الحزمة `main` حيث يتم اعتبارها الحيز الذي يتم تعريف المتغيرات فيه

*Code(109)

```
$perl = "It`s programming language";
print $main::perl, "\n";
```

Typeglobs

يتم عنونة الـ `typeglob` في لغة البيزل من خلال استعمال الرمز `*` الذي يعتبر بمثابة الرمز التعريف لهذه التقنية في لغة البيزل



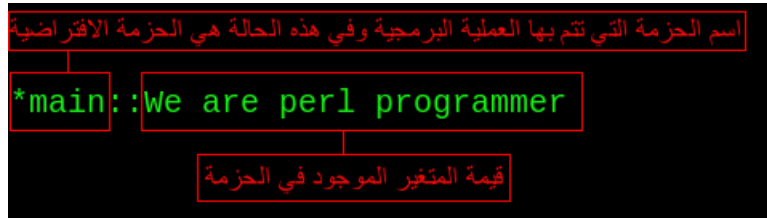
Figure(85)

جميع المتغيرات العامة في لغة البيزل تعود الى الحزمة الافتراضية او الحزمة `main::` ويكون الاسلوب البرمجي الذي تتبعه الـ `typeglob` مع المتغيرات التي تكون من `scalar` كما يلي من خلال المقطع البرمجي الاتي

*Code(110)

```
$perl = "We are perl programmer";
*perl_Tg = $perl;
print *perl_Tg;
```

الان عندما يتم تنفيذ المقطع البرمجي أعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي



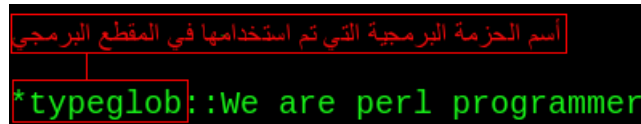
Figure(86)

يعزى سبب ظهور الحزمة `main` عندما تم تنفيذ المقطع البرمجي السابق هو أنه المتغير الموجود في المقطع البرمجي السابق لم يتم تعريفه في اي حزمة متخصصة لذا تم اعتباره متغير يعود الى الحزمة الافتراضية

*Code(111)

```
package typeglob ;
$perl = "We are perl programmer";
*perl_Tg = $perl;
print *perl_Tg;
```

الان عندما يتم تنفيذ المقطع اعلاه فان الناتج هو كما يلي من خلال الشكل التوضيحي الاتي



Figure(87)

*Code(112)

```
@perl = ("Storm","Striker","Spawn");
*perl_Tg = @perl;
print *perl_Tg;
```

الان لو يتم تنفيذ المقطع البرمجي اعلاه فان الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

```
*main::3
```

Figure(88)

وضح المقطع البرمجي اعلاه الطريقة التي يتم من خلالها التعامل مع المصفوفات من خلال تقنية الـ `typeglob` ويلاحظ انه عندما تم تنفيذ البرنامج كان الناتج من عملية التنفيذ هو ظهور رقم فقط وليس العناصر التي والسبب هو أنه في هذه الحالة اي في حالة التعامل مع المصفوفات فان الناتج يشير الى عدد العناصر التي تحتويها المصفوفة وبما أن عدد العناصر هو 3 فان الرقم الذي تم الحصول عليه عند التنفيذ هو 3

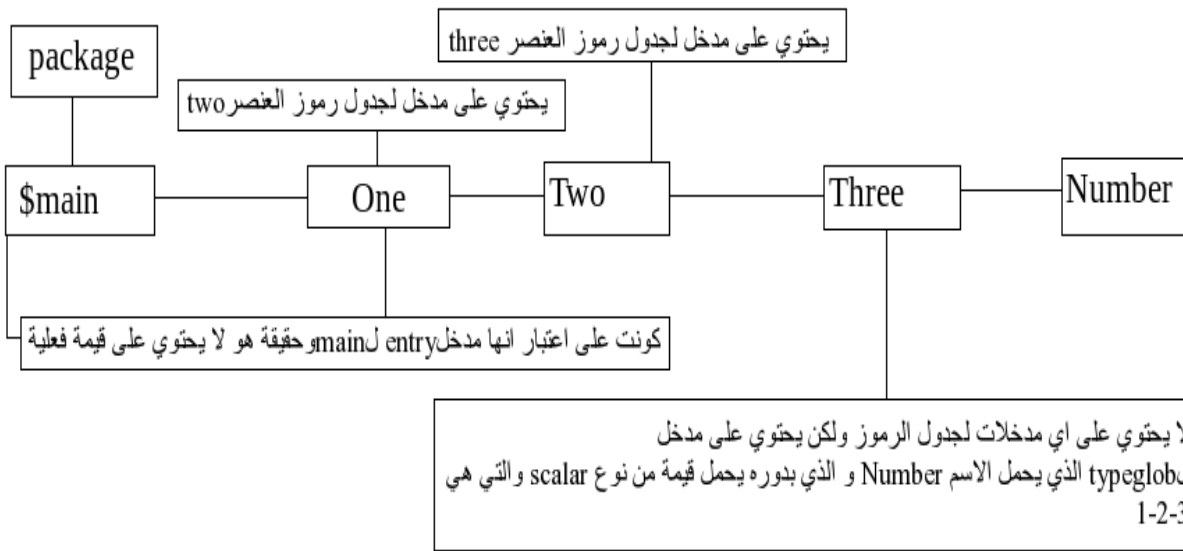
The Symbol Table

عندما يتم تكوين حزمة جديدة في لغة البيزل فهذا يعني ان البيزل سوف تقوم بتكوين جدول `symbol table` لكي يتعامل مع المتغيرات وفي لغة البيزل يتم التفريق بين اسماء الحزم بواسطة العلامة المزدوجة :: كما في عالم المجلدات حيث يتم استعمال ال كل من العلامتين \ / للتفريق بين المجلدات وعلى سبيل المثال لو تم تعريف حزمة كهذه الحزمة

*Code(113)

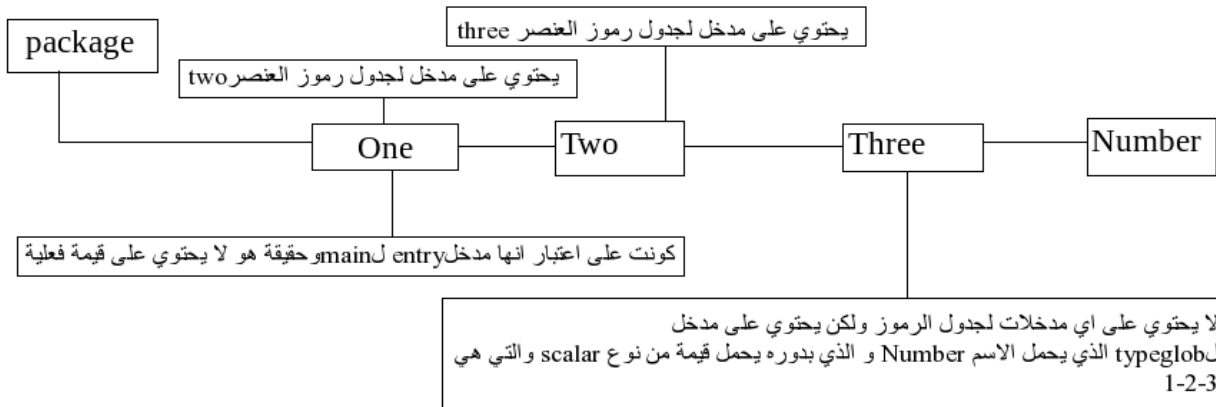
```
package One::Two::Three;
$Number = "1-2-3";
```

يوضح المقطع البرمجي أعلاه الفكرة الاتية



Figure(89)

وبما ان `main` هي الجذر ولا تحتوي على قيمة فعلية لذا من الممكن ان يتم الاستغناء عنها



Figure(90)

P3rL ObjEcT

الكائن هي عبارة عن طريقة للوصول الى البيانات وفي اغلب اللغات البرمجية يكون الكائن عبارة عن مستودع يتم يحتوي على البيانات ولكن بصورة عامة اي شئ يوفر طريقة للوصول الى البيانات من الممكن ان يكون كائن و البيانات التي يوفر لها الكائن لها الوصول تعرف بالمصطلح الاتي Attribute values والمستودعات التي تخزن هذه القيم تعرف بأسم ال Attributes ومن الممكن القول ان الكائنات هي اكثر من مجرد مجموعة من المتغيرات حيث أن اغلب الكائنات لها خاصية أخرى تعرف بخاصية التغليف او ما يعرف برمجيا بأسم encapsulation وخاصية التغليف تعني ان خواص الكائن من غير الممكن أن يتم الوصول اليها في كافة انحاء البرنامج ولكن من الممكن ان يتم الوصول اليهم من خلال روتينات فرعية معينة وهذه الروتينات الفرعية في هذه الحالة يطلق عليهم اسم العلاقات او ال Methods وبصورة عامة هذه العلاقات تكون متاحة لكل عمليات الوصول وفي بعض الاحيان يكون الغرض من هذه العلاقات هو أن يتم تقليل الطرق التي من الممكن أن تتبع من أجل تغيير قيم الكائن

Classes

ال object class في لغة البيزل يوفر التطبيق للكائن حيث ان ال class في لغة البيزل يتكون من

1- class methods

والتي هي عبارة عن روتينات تعمل على تأدية وظيفة برمجية تكون خاصة لل class

2- object methods

والتي هي عبارة عن روتينات تعمل على تأدية وظيفة برمجية تكون خاصة للكائن ما بصورة مستقلة

3- package variables

ويحتوي ال class أيضا على متغيرات حزم و لكن هذه المتغيرات في مجال البرمجة كائنية المنحى يطلق عليها أسم class attribute

object class يحتوي على الاقل على علاقة تكون هذه العلاقة هي مسؤولة عن توليد كائنات جديدة تعرف هذه العلاقة بأسم constructor method وهناك بعض ال object class قد تحتوي على علاقة اخرى تعرف هذه العلاقة الاخرى بأسم العلاقة الهدامة التي تعمل على ترتيب الكائنات التي تم تهديمها

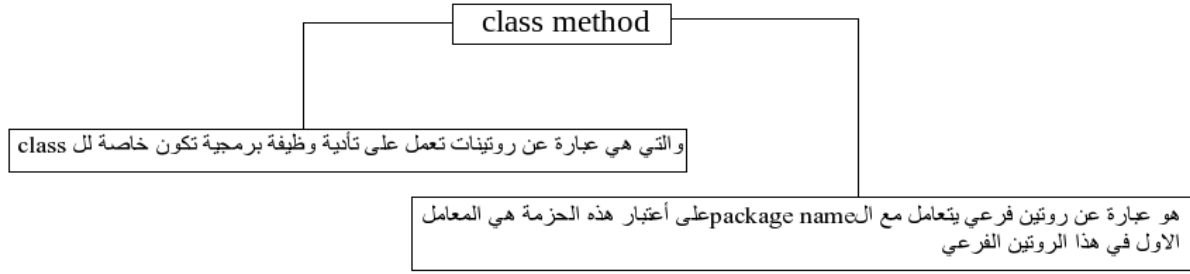
1- class method

هو عبارة عن روتين فرعي يتعامل مع ال package name على اعتبار هذه الحزمة هي المعامل الاول في هذا الروتين الفرعي

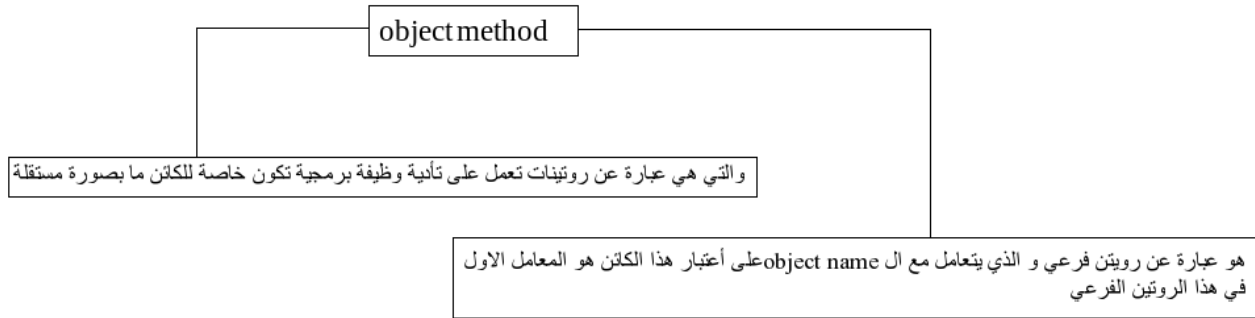
2- object method

هو عبارة عن روتين فرعي و الذي يتعامل مع ال object name على اعتبار هذا الكائن هو المتغير الاول في هذا الروتين الفرعي

ولكن لغة البيزل تعمل على تمرير هذه المتغيرات بصورة أتوماتيكية عندما يقوم المبرمج باستعمال علامة السهم او ما يعرف بال arrow method(->)



Figure(91)



Figure(92)

Creating Objects

كل ال **object classes** تحتوي على الاقل علاقة واحدة تعرف هذه العلاقة بالعلاقة البناءة حيث تعمل هذه العلاقة على بناء وتكوين الكائنات الجديدة ولغة البييرل تسمح للبرمج ان يطلق اي اسم على هذه العلاقة البناءة ولكن بشرط واحد هو ان تكون هذه العلاقة هي عبارة عن روتين فرعي ويكون تكويني الكائن في لغة البييرل كما يلي من خلال المقطع البرمجي الاتي

*Code(114)

```
$object = programming::fr34ks->new();
```

يلاحظ في البرمجي السابق انه تمت عملية التعريف حيث وضع ال **New keyword** بعد اسم الحزمة ولكن لغة البييرل توفر طريقة اخرى للتعامل مع هذه الحالة وهي كما يلي من خلال المقطع البرمجي الاتي

*Code(115)

```
$object = new programming::fr34ks();
```

تجدد الاشارة الى أن من خلال المقطع البرمجي السابق المعني البرمجي هو ذات المعنى ولكن الطريقة هي التي قد اختلفت أي في الطريقة الثانية تم تعريف ال **new keyword** قبل اسم الحزمة ولكن في المقطع البرمجي الاسبق تمت عملية التعريف بعد اسم الحزمة

Using Objects

مع الكائنات في لغة البييرل يتم التعامل مع علامة السهم او ما يعرف بال **arrow operator** للوصول الى الكائنات والتعامل معها ولكن تجدد الاشارة الى انه في لغة البييرل انه عندما يكون المبرمج في بيئة البرمجية كائنية المنحى يكون السهم له دلالة و هي التعامل مع الكائنات و الوصول اليها ولكن عندما يكون المبرمج في بيئة أخرى خارج البرمجة كائنية المنحى يكون لها للسهم استعمال ودلالة برمجية أخرى و هي الحالة الاتية

*Code(116)

```
$value = $hashref->{'key'};
```

يشير المقطع البرمجي اعلاه الى ان السهم في الوسط البرمجي الخارج عن نطاق هو التعامل مع المتغيرات التي من نوع هاش و التي تم معاملتها بدورها مع تقنية المرجع او الـ rفرنس

*Code(117)

```
$object_result = $object->method(@args);
$class_result = Class::Name->classmethod(@args);
$value = $object->{'property_name'};
```

Calling class method

ال class method هي عبارة عن روتين فرعي يتم تعريفه في داخل الـ class وغالبا ما يقتصر العمل البرمجي لهذه الـ class method على الـ class بصورة خاصة أكثر من من عمله على الكائن البرمجي

*Code(118)

```
$result = My::Object::Class->classmethod(@args);
```

هكذا يكون التمثيل البرمجي لعملية الـ Class method في لغة البيزل ولكن من الممكن أن يتم تغيير عملية التمثيل البرمجي من هذه الحالة الى حالة اخرى

*Code(119)

```
$result = classmethod My::Object::Class(@args);
```

في الحالة البرمجية الاولى كان الاسلوب البرمجي المتبع هو هو أنتكون الحزمة البرمجية هي الاولى ومن ثم يتن استدعاء الـ class method والحالة البرمجية الثانية كانت العملية معاكسة للعملية الاولى

Calling object method

ال object method عبارة عن روتين فرعي يتم تعريفه داخل الصنف ويكون عمله البرمجي على الكائن بصورة محددة وتتم عملية استدعاء الـ object method من خلال علامة السهم او الـ arrow operator

*Code(120)

```
$result = $object->method(@args);
```

يلاحظ في الفقرة البرمجية اعلاه انه تم وضع الكائن في بداية السطر البرمجية ومن ثم بعد ذلك وضع الروتين الفرعي مع المعاملات التي يحتويها هذا الروتين الفرعي

How To ref

ال keyword "ref" هي عبارة عن دالة برمجية من الدوال المبنية في النظام في لغة البيزل ويكون اسلوب التمثيل البرمجي الخاص بها كما يلي من خلال المقطع البرمجي الاتي

*Code(121)

```
$perl_var="perl programmer";
$perl_ref = \$a;
if (ref($perl_ref) eq "SCALAR"){
print "ok, it`s a ref a now";
}
else {
print "Bad, it`s not a ref any more";
}
```

و الان عندما يتم تنفيذ المقطع البرمجي اعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل التوضيحي الاتي

```
ok, it`s a ref a now
```

Figure(93)

*Code(122)

```
$perl_array=("Spawn","Storm","Striker");
$perl_ref = \@perl_array;
if (ref($perl_ref) eq "ARRAY"){
print "ok, it`s a ref a now";
}
else {
print "Bad, it`s now a ref any more";
}
```

والان عندما يتم تنفيذ المقطع البرمجي اعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

```
ok, it`s a ref a now
```

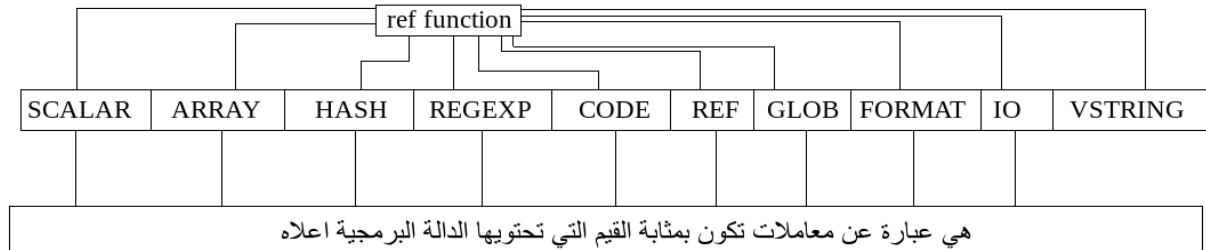
Figure(94)

من خلال كل من المقطعين البرمجين السابقين تم استعمال نوعين من المتغيرات وهما

1- scalar

2- array

حيث تم تحويلهم من صيغتهم البرمجية البسيطة الى صيغة المرجع اي تم تحويلهم الى مرجع ومن ثم استعمال دالة ال ref لكي يتم التأكد من أن هذين المتغيرين تم معاملتهم بتقنية المرجع بنجاح



Figure(95)

How To bless

تعتبر دالة ال bless من الدوال البرمجية المبنية في لغة البيرل ويكون اسلوب التمثيل البرمجي لهذه الدالة في لغة البيرل كما يلي من خلال المقطع البرمجي الاتي

*Code(123)

```
bless 1- <reference> ,2- <packages>
```

الدالة البرمجية اعلاه كما هو موضح تتعامل مع نوعين من المعاملات النوع الاول من المعاملات هو

1- reference

2- packages

أما عن طريقة التمثيل البرمجي لهذه الدالة فهي تكون كما يلي من خلال المقطع البرمجي الآتي

*Code(124)

```
$a= " We are perl programmers";  
$a_ref = \ $a;  
print '$a_ref is a ',ref $a_ref, " reference", "\n";  
bless ($a_ref, "P3rL ProGrammErs");  
print '$a_ref is a ',ref $a_ref, " reference", "\n";
```

الآن عندما يتم تنفيذ المقطع البرمجي اعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الآتي

```
$a_ref is a SCALAR reference  
$a_ref is a P3rL ProGrammErs reference
```

Figure(96)

الآن وبعد أن تم تنفيذ المقطع البرمجي اعلاه فأن ما قد حصل هو انه قد تم تحويل المتغير الذي يحمل الاسم \$a_ref الى مرجع او reference ولكن هذا المرجع الآن يعلم ماهي الحزمة البرمجية التي يعود إليها أي انه لو تم استدعاء method على هذا المتغير فأن لغة البيزل الآن سوف تقوم بالبحث عن هذه العلاقة في داخل الحزمة التي تحمل الاسم P3rL ProGrammErs

Constructors

الجزء الاهم من أي كائن هو العلاقة البناءة التي يحتويها وهي العلاقة التي يتم استخدامها في بناء وتكوين كائنات اخرى جديدة و العلاقة و الوحيدة للكائن التي تستخدم في عملية بناء الكائنات الجديدة هي `new`. ويكون تكوين الكائنات الجديدة من خلال المقطع البرمجي الاتي الطريقة الاولى

*Code(125)

```
$object = new My::Object::Class;
$object = new My::Object::Class('initial', 'data', 'for', 'object');
```

الطريقة الثانية

*Code(126)

```
$object = My::Object::Class->new();
$object = My::Object::Class->new('initial', 'data', 'for', 'object');
```

الطريقة البرمجية الاولى هي الطريقة التي يتم أتباعها عادة في ال Object oriented syntax أما الطريقة البرمجية الثانية هي الطريقة التي يتم أتباعها في ال class method call والمعرف `new` هو عبارة عن روتين فرعي يقوم ببناء وتكوين الكائنات البرمجية و يكون غالبا استعمال المعرف `new` متبوعا بال (`->`) arrow operator وتعتبر الدالة `bless` هي قلب اي عملية بناء اي قلب اي عملية `object constructing` وكما قد ذكر اعلاه فان الدالة `bless` تتعامل مع نوعين من المتغيرات وهذين النوعين هما

1- references

2- packages

ولكن تجدر الاشارة الى ان الدالة `bless` من الممكن ان تتعامل مع نوع واحد من المتغيرات وهذا النوع هو المرجع ويتم اعتماد الحزمة الافتراضية كموقع لهذا المرجع

*Code(127)

```
use strict;
sub new {
  $self = {};          # create a reference to a hash
  bless $self;        # mark reference as object of this class
  return $self;      # return it.
}
```

على المبرمج الذي يتعامل مع هكذا موقف من المواقف التي يتم استعمال نمط واحد مع دالة ال `bless` فهذا يعني أن هذا النمط لايملك القدرة على التعامل مع تقنية الوراثة وهذا ما يطلق عليه بأسم ال simple constructor لذا لا بد من استعمال النمط الكامل من هذه الدالة لكي يكون استعمال الوراثة ممكنا

The new method

لقد ذكر في الموضوع السابق ان المعرف `new` هو عبارة عن روتين فرعي اذن فهذا يعني انه هذا الروتين الفرعي لديه تعريف خاص به وهذا التعريف الخاص بهذا المعرف هو كما يلي من خلال المقطع البرمجي

*Code(128)

```
sub new {
  my $self = {};
  bless $self;
  return $self;
}
```

المقطع البرمجي اعلاه هو عبارة عن احد التعاريف التي تم استعمالها في المعرف `new` ولكن اذا ما تم استعمال هذا

المقطع البرمجي فهذا يعني أن المقطع البرمجي لا يمكن ان يتم استعمال تقنية الوراثة لانه دالة ال `bless` تم تحديد معامل واحد لها كمعرف لذا لا بد من استعمال مقطع برمجي أخر لهذا المعرف ويسمح باستعمال تقنية الوراثة وهو

*Code(129)

```
sub new {
my $class = shift;
my $self = {@_};
bless($self, $class);
return $self;
}
```

*Code(130)

```
package programmers;
sub new {
my $class = shift;
my $self = {@_};
bless($self, $class);
return $self;
}
my $object = programmers::new("programmers",
Spawn =>"Perl programmer",
Storm =>"C programmer",
Striker =>"Python programmer",
Website =>"programming-fr34ks"
);
print $object->{Striker}, "\n";
```

والان عندما يتم تنفيذ المقطع البرمجي المذكور اعلاه فأن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

Python programmer

Figure(97)

الان كما يلاحظ من المقطع البرمجي اعلاه فأن البيانات يتم خزنها في ال `hash reference` حيث يتم التعامل مع هذه البيانات على أنها معاملات للعلاقة البناءة و الان عندما يتم استدعاء العلاقة البناءة فأن لغة البيزل سوف ترى البيانات التي تم خزنها في ال `hash reference` كما يلي

*Code(131)

```
"Spawn", "Perl programmer",
"Storm", "C programmer",
"Striker", "Python programmer",
"Website", "programming-fr34ks"
```

ثم تقوم العلاقة البناءة باستعمال الاسم ال `Class` كما قد ذكر سابقا ثم ما قد تبقى من المعامل البرمجي سوف يكون شكله كما يلي من خلال المقطع البرمجي الاتي

*Code(132)

```
@_=(
"Spawn","Perl programmer",
"Storm","C programmer",
"Striker","Python programmer",
"Website","programming-fr34ks"
);
```

وعندها عند نهاية البرنامج يتم طباعة ناتج العملية
ومن الممكن ان يتم صياغة المقطع البرمجي 131 بطريقة اخرى حيث يكون تمثيله البرمجي كما يلي من خلال
المقطع البرمجي الاتي

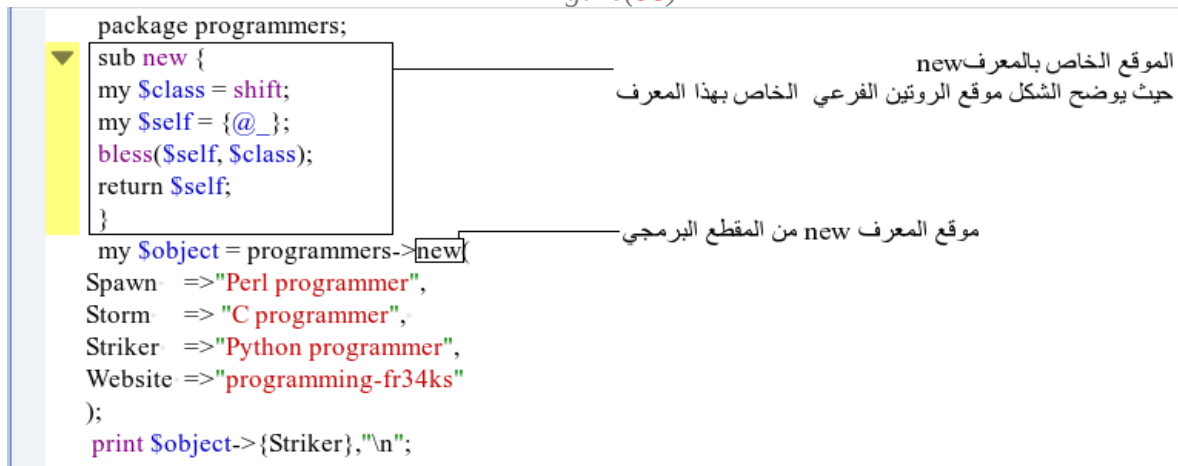
*Code(133)

```
package programmers;
sub new {
my $class = shift;
my $self = {@_};
bless($self, $class);
return $self;
}
my $object = programmers->new(
Spawn =>"Perl programmer",
Storm =>"C programmer",
Striker =>"Python programmer",
Website =>"programming-fr34ks"
);
print $object->{Striker}, "\n";
```

وايضا عندما يتم تنفيذه فان الناتج من عملية التنفيذ سوف يكون كما يلي من خلال الشكل الاتي

Python programmer

Figure(98)



Figure(99)

Creating methods

لقد ذكر على في الصفحات السابقة ان المعرف `new` هو عبارة عن علاقة وأن هذه العلاقة هي عبارة عن روتين فرعي اذن الممكن ان يقوم المبرمج بكتابة اي علاقة برمجية يرغب بها عن طريق كتابة روتين فرعي يقوم بهذه العملية ومن الممكن عندها ان يتم معاملة هذا الروتين الفرعي على أنه عبارة عن علاقة برمجية وحينها يتحول الى معرف يعامل معاملة المعرف `new` ويكون التمثيل البرمجي لهذه العملية في لغة البيرل كما يلي من خلال المقطع البرمجي الاتي

*Code(134)

```
package programmers;
sub new {
my $class = shift;
my $self = {@_};
bless($self, $class);
return $self;
}
sub get_name {
my $self = shift;
return $self->{Perl_programmer};
}
my $object = programmers->new(
Perl_programmer    =>"Spawn",
C_programmer       =>"Storm",
python_programmer =>"Striker",
programming_fr34ks =>"website"
);
print $object->get_name,"\n";
```

الان عندما يتم تنفيذ المقطع البرمجي المذكور اعلاه فإن الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

Spawn

Figure(100)

```

1 package programmers;
2 sub new {
3   my $class = shift;
4   my $self = {@_};
5   bless($self, $class);
6   return $self;
7 }
8 sub get_name {
9   my $self = shift;
10  return $self->{Perl_programmer};
11 }
12 my $object = programmers->new(
13   Perl_programmer => "Spawn",
14   C_programmer => "Storm",
15   python_programmer => "Striker",
16   programming_fr34ks => "website"
17 );
18 print $object->get_name, "\n";

```

موقع المعرفة `get_name`
حيث يوضح الشكل موقع الروتين الفرعي الخاص بهذا المعرفة

موقع المعرفة `get_name`
من المقطع البرمجي

Figure(101)

Inheritance

توفر تقنية الوراثة في لغة البيزل للمبرمج خاصية مميزة وهذه الخاصية تكون في اعادة استعمال المقاطع البرمجية لكي لا يكون البرنامج مليء بالمقاطع البرمجية التي من الممكن ان يتم التحلي عنها يكون التمثيل البرمجي للوراثة في لغة البيزل من خلال استعمال متغير خاص الذي يوفر هذه الخاصية وهو المتغير

*Code(135)

```
@ISA
```

هذه المصفوفة الخاصة هي المتغير الذي تتعامل مع تقنية الوراثة ويكون التمثيل البرمجي لهذه العملية كما يلي من خلال المقطع الاتي

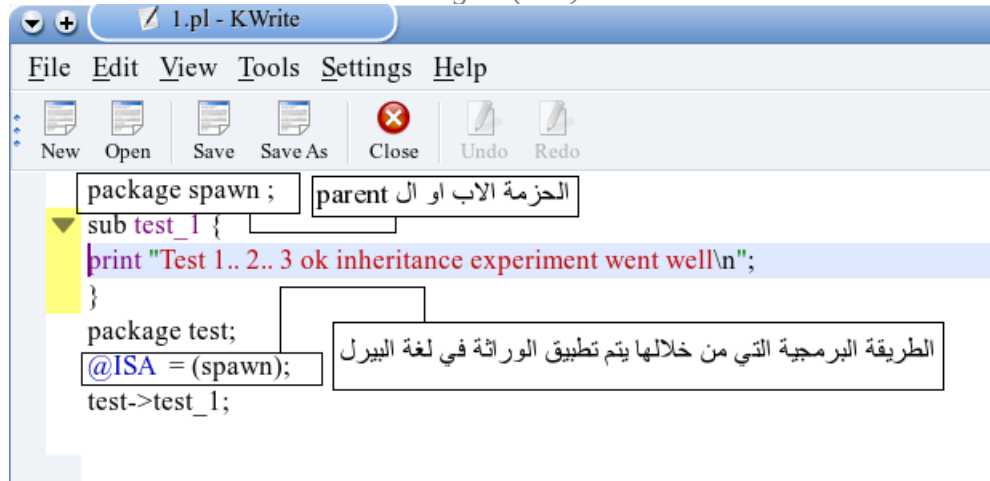
*Code(136)

```
package spawn ;
sub test_1 {
print "Test 1.. 2.. 3 ok inheritance experimnt went well\n";
}
package test;
@ISA = (spawn);
test->test_1;
```

اذا تم تنفيذ المقطع البرمجي السابق فإن الناتج من عملية تنفيذه سوف تكون كما يلي من خلال الشكل الاتي

```
Test 1.. 2.. 3 ok inheritance experimnt went well
```

Figure(102)

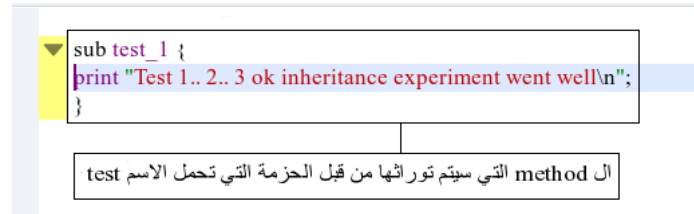


Figure(103)

يتضح من خلال المقطع البرمجي السابق ان عملية الوراثة قد تمت حيث ان الحزمة التي تحمل الاسم test الان سوف تقوم بالتوارث من الحزمة التي تحمل الاسم spawn حيث أن الحزمة test سوف ترث العملية التي تحمل الاسم

test_1

- 1- parent package (spawn)
- 2- child package (test)
- 3- method (test_1)



Figure(104)

اسم الحزمة التي سيتم الوراثة منها
 @ISA = (spawn); انها تحتوي على الحزمة ال parent

Figure(105)

يلاحظ من الشكل الذي يحمل الرقم (105) ان مصفوفة الوراثة تتعامل مع متغير واحد فقط وهو حزمة التي تحمل الاسم spawn هذا النوع من الوراثة يعرف بأسم الوراثة الفردية او single inheritance او ما تعرف بأختصارا ب (SI) وهي تعني الوراثة التي تتعامل مع عنصر واحد فقط
 واذا حاول المبرمج ان يقوم بتنفيذ المقطع البرمجي 136 بالطريقة الاتية

*Code(137)

```

package spawn ;
sub test_1 {
  print "Test 1.. 2.. 3 ok inheritance experiment went well\n";
}
package test;
test->test_1;

```

يلاحظ من المقطع البرمجي أنه سيتم تنفيذه من دون الاستعانة بمصفوفة الوراثة فأن الناتج عندما يتم التنفيذ سيكون كما يلي من خلال الشكل الاتي

Can't locate object method "test_1" via package "test" at - line 6.

Figure(106)

@ISA

أولا : تجدر الإشارة الى أن هذه المصفوفة ISA يتم لفظها بالطريقة الاتية IS-A
 ثانيا : هذا المتغير يجب التعامل معه دائما بالطريقة ال Global ولا يجب ان يتم معاملته مع الدالة my

Overriding the methods

حالات ال overriding في لغة البيزل البيزل هي تلك الحالة البرمجية التي يكون المقطع البرمجي الذي يتم التعامل معه يحتوي على two methods تكون العلاقة الاولى تعود الى base class و الاخرى تعود الى derived class ويكون التمثيل البرمجي لهذه في لغة البيزل كما يلي من خلال المقطع البرمجي الاتي

*Code(138)

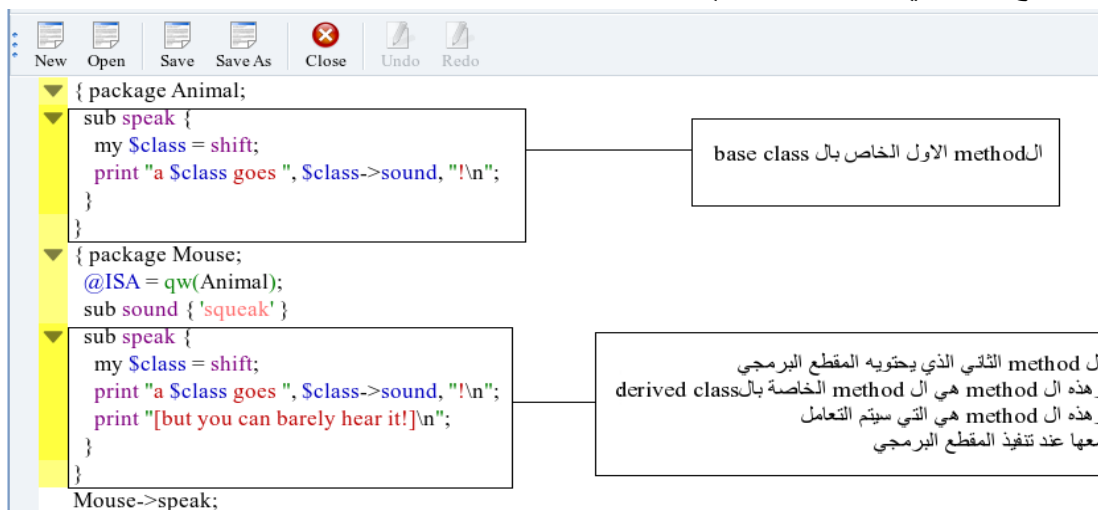
```
{ package Animal;
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
  }
}
{ package Mouse;
  @ISA = qw(Animal);
  sub sound { 'squeak' }
  sub speak {
    my $class = shift;
    print "a $class goes ", $class->sound, "!\n";
    print "[but you can barely hear it!]\n";
  }
}
Mouse->speak;
```

الان عندما يتم تنفيذ المقطع البرمجي اعلاه فان الناتج من عملية التنفيذ سوف تكون كما يلي من خلال الشكل الاتي

```
a mouse goes squeak!
[but you can barely hear it!]
```

Figure(107)

يلاحظ من المقطع البرمجي اعلاه انه يحتوي كما ذكر على two methods



Figure(108)

***License**

This book made under the terms of the (GPL) license and you are free to do what ever you wanna do without any prior permission

***Thanks**

Thanks for the people who stand by me all the time and give me the support I need

***Greetings**

Storm(Twistedjoker),Striky,Sofy

www.programming-fr34ks.net

*Wrote BY:-
M_SpAwn
P3rL WaRRiOr*