

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِیْمِ

إبدأ مع لغة أوبجكت باسكال

```
Source Editor
Hello.lpr 38
1 program Hello;
.
. (smode objfpc)(SH+)
.
5 uses
. (SIFDEF UNIX)(SIFDEF UseCThreads)
. cthreads,
. (SENDIF)(SENDIF)
. Classes
10 ( you can add units after this );
.
. begin
13 Writeln('Hello world');
. end.
15
```

تأليف: معترز عبدالعظيم الطاهر

Object Pascal

كود لبرمجيات الكمبيوتر

مقدمة الكتاب

بسم الله الرحمن الرحيم، والصلاة والسلام على أشرف الأنبياء والمرسلين، نبينا محمد وعلى آله وصحبه أجمعين.

أما بعد فهذا الكتاب موجه لمن يريد أن يتعلم لغة باسكال الكائنية (Object Pascal) لاستخدامها مع مترجم فري باسكال Free Pascal أو أداة التطوير دلفي Delphi. فهو يصلح للمبرمج الجديد الذي ليست لديه خبرة في البرمجة شرط أن يكون لديه خبرة ومعرفة في علوم الحاسوب. كذلك فهو يصلح لمن كانت لديه خبرة في لغة برمجة أخرى ويريد تعلم هذه اللغة. وبالنسبة للمبرمج الجديد فهذا الكتاب يُساعد أيضاً على فهم وتعلم البرمجة عموماً حيث أنه يساعد المبرمج على فهم كيفية حل المشاكل والحيل البرمجية التي يستخدمها المبرمج في التطبيقات المختلفة.

لغة أوبجكت باسكال

أول ظهور للغة باسكال تدعم البرمجة الكائنية الموجهة (Object Oriented Programming) كان في عام 1983 في شركة أبل للكمبيوتر. بعد ذلك تلتها تيربو باسكال 5.5 الشهيرة التابعة لسلسلة تيربو باسكال التي أنتجتها شركة بورلاند، وقد أضاف المبرمج أندرس هجلبيرغ البرمجة الكائنية لخط تيربو باسكال في عام 1989.

لغة باسكال الكائنية هي لغة تدعم البرمجة الهيكلية (Structured Programming) كما وتدعم البرمجة الكائنية (Object Oriented Programming).

لغة باسكال هي لغة متعددة الأغراض تصلح لعدد كبير من التطبيقات والاحتياجات، فبدايةً من تعلم البرمجة لسهولةها ووضوحها، مروراً بالألعاب، والبرامج الحاسوبية، والبرامج التعليمية، وتطبيقات الإنترنت، وبرامج الاتصالات، ولغات البرمجة، فمثلاً بيئة تطوير دلفي مطورة بالدلفي نفسها، كذلك فري باسكال و لازاراس طُورت باستخدام لغة باسكال، وانتهاءً بنظم التشغيل مثل الإصدارات الأولى من نظام تشغيل ماكنتوش كان يستخدم فيها لغة باسكال الكائنية و كذلك نواة نظام التشغيل Toro OS المستخدم فيه مترجم Free Pascal.

بعد النجاح التي حققته أداة التطوير تيربو باسكال التي كانت تُستخدم لإنتاج برامج تعمل في نظام DOS، أنتجت شركة بورلاند أداة التطوير دلفي في عام 1995 لتستهدف نظام التشغيل وندوز 16 بت ثم وندوز 32 بت في إصدارة دلفي 2 ليعمل مع إصدارة وندوز 95. وقد لاقت دلفي نجاحاً كبيراً حيث أنها كانت تنتج برامج أسرع بأضعاف المرات التي كانت تنتجها نظيرتها فيجوال بيسك، وكانت البرامج التي تنتجها لدلفي لا تحتاج لمكتبات إضافية أو ما يُعرف بالـ **Run-time libraries** حين توزيع وتشغيل البرامج في أجهزة أخرى لا تحتوي على دلفي.

دلفي Delphi

أداة التطوير دلفي التي أنتجتها شركة بورلاند كما سبق ذكره هي عبارة عن أداة تطوير سريعة للبرامج (**Rapid Application Development Tool**) ، أما اللغة المستخدمة في هذه الأداة فهي لغة باسكال الكائنية.

حدث تطوير كبير للغة باسكال الكائنية من قبل فريق دلفي حتى أصبحت لغة منافسة لمثيلاتها. وأصبحت لغة ذات إمكانات عالية ومكتبات غنية.

بلغ عدد المبرمجين الذين يستخدمون دلفي أكثر من مليون ونصف مبرمج حسب إحصائية الشركة في عام 2008. و أنتجت بها برامج كثيرة، نذكر منها Skype, Morfik, Age of wonder وغيرها من البرامج المهمة، بالإضافة إلى البرامج الموجهة للشركات Enterprise Applications.

فري باسكال Free Pascal

بعد توقف شركة بورلاند عن إنتاج خط تيربو باسكال الذي كان يستخدم نظام DOS إلى عام 1993، أنتج فريق فري باسكال نسخة شبيهة بتيربو باسكال ليكون بديل حر ومفتوح المصدر. لكن هذه المرة مع إضافة مهمة وهي استهداف منصات جديدة مثل: لينكس، و ماكنتوش، و ARM، و IOS و Android، وغيرها بالإضافة إلى وندوز 32 بت و وندوز 64 بت.

فريق فري باسكال كانت إحدى أهدافه هي التوافقية مع لغة باسكال الكائنية المستخدمة في دلفي. النسخة الأولى من مترجم فري باسكال صدرت في يوليو عام 2000.

لازاراس Lazarus

بعد نجاح مترجم فري باسكال وتفوقه على مترجم تيربو باسكال، وإنتاج نسخة تعمل في عدد من المنصات التشغيلية، كانت الحلقة الناقصة هي أداة التطوير المتكاملة. لازاراس هي أداة التطوير المستخدمة مع فري باسكال، أو هي أداة التطوير التي تستخدم فري باسكال كمترجم. وتحتوي على مكتبة ضخمة للكائنات **class library**، وبهذه الطريقة نكون قد حولنا أداة باسكال إلى أداة معتمدة على التطوير باستخدام المكونات أو الكائنات **component driven development** مماثلة للدلفي بالإضافة لكون لازاراس محرر للكود ومصمم للبرنامج. فهي بذلك تحقق كونها أداة تطوير سريعة **RAD Rapid Application Development**. بدأ مشروع لازاراس عام 1999 أُصدرت النسخة رقم 1 منه في أواخر أغسطس من العام 2012، لكن كُتب عدد كبير من البرامج بواسطة النسخ السابقة للنسخة رقم 1 كما كُتب عدد من الكتب حوله. في هذا الكتاب سوف نستخدم لازاراس وفري باسكال في كتابة وشرح البرامج، إلا أن نفس البرامج يمكن تطبيقها باستخدام دلفي مع بعض التعديلات.

مميزات لغة باسكال

تتميز لغة باسكال الهدفية بسهولة تعلمها، وإمكاناتها العالية، وسرعة مترجماتها والبرامج التي تنتج عنها. لذلك فهي تعطي المبرمج فرصة إنتاج برامج ذات كفاءة واعتمادية عاليتين في وقت وجيز، باستخدام بيئة تطوير متكاملة وواضحة دون الدخول في تعقيدات اللغات وأدوات التطوير الصعبة. وهذا يحقق الإنتاجية العالية.

الهؤلف: رعتز عبد العظيم الطاهر

تخرجت في جامعة السودان للعلوم والتكنولوجيا عام 1999م الموافق 1420 هجرية. وقد بدأت دراسة لغة باسكال في العام الأول في الجامعة عام 1995 كلغة ثانية بعد تعلم لغة بيسك الذي بدأت تعلمه في المرحلة المتوسطة. ومنذ بداياتي مع لغة باسكال ما زلت أستخدمها إلى الآن. وقد بدأت استخدام أداة التطوير دلفي في عام 1997م. والآن استخدم فري باسكال و لازاراس لكتابة عدد من البرامج خصوصاً برامج سطح المكتب أو بعض البرامج الخدمية. بالإضافة للغة باسكال استخدم لغة جافا ولغة Go.

ترخيص الكتاب:

ترخيص الكتاب هو رخصة الإبداع العامة

Creative Commons CC BY-SA 3.0

بيئة التعليم المستخدمة مع هذا الكتاب:

سوف نستخدم في هذا الكتاب إن شاء الله بيئة لازاراس وفري باسكال كما سبق ذكره، ويمكن الحصول عليه من هذا الموقع: lazarus.freepascal.org. أو يمكن الحصول عليه من داخل نظام لينكس عن طريق برامج إضافة التطبيقات، حيث نجده في قسم أدوات التطوير، أو عن طريق استخدام yum install lazarus في فيدورا أو ما يشابهها من توزيعات لينكس، أو باستخدام apt-get install Lazarus في توزيعة Ubuntu أو ما يشابهها. و لازاراس هي أداة تطوير حرة ومفتوح المصدر كما ذكرنا، وتوجد في أكثر من منصة نظام تشغيل. الكود والبرامج المكتوبة بواسطتها يمكن نقلها لإعادة ترجمتها وربطها (Compilation and linking) في أي منصة يريد المبرمج، إلا أن البرنامج الناتج (الملف الثنائي executables) لا يمكن نقله إلى نظام مختلف، فهو مرتبط فقط بالمنصة التي ترجم المبرمج برنامجها فيها حيث أن لازاراس وفري باسكال ينتج عنهما ملفات ثنائية تنفيذية تعمل مباشرة على نواة نظام التشغيل ومكتباتها و لا تحتاج

لوسيط مثل برامج جافا ودوت نت، لذلك فهو يتفوق على هذه اللغات بالسرعة في التنفيذ وعدم الحاجة لمكتبات إضافية في الأجهزة التي سوف يُثبت البرامج المُنتج بواسطة لازاراس بها.

استخدام البيئة النصية

الفصول الأولى من هذا الكتاب تستخدم إمكانيات الإدخال والإخراج البسيطة والأساسية التي تسمى ببرامج سطر الأوامر command line أو برامج البيئة النصية **Text mode**، وذلك لبساطتها وتوفرها في كل أنظمة التشغيل وسهولة فهمها.

ربما يمل الدارس منها ويتمنى أن يصنع برامج تستخدم الشاشة الرسومية بما تحتويه من أزرار ونوافذ وقوائم ومربعات نصية وغيرها، إلا أننا أحببنا أن لانشغل الدارس بشيء ويركز على المفاهيم الأساسية التي سوف يستخدمها في البرامج النصية البسيطة وسوف يستخدمها في البرامج الرسومية المعقدة إن شاء الله. بهذه الطريقة يكون الفهم سهلاً وسريعاً لأساسيات الباسكال والبرمجة. وفي الفصول المتقدمة سوف نستخدم إن شاء الله البيئة الرسومية ذات الواجهة المحببة للمستخدم العادي.

الأمثلة

معظم الأمثلة المستخدمة يف هذا الكتاب سوف نجدها في الموقع <http://code.sd> في الصفحة التي تحتوي على هذا الكتاب. وعلى الدارس محاولة تصميم الأمثلة خطوة بخطوة، حتى تترسخ له البرمجة واستخدام هذه الأدوات. فإذا تعذر له ذلك أو حدثت مشكلة يمكن مقارنة البرنامج الذي كتبه بنفسه مع المثال الموجود في الصفحة.

المحتويات

2	مقدمة الكتاب
2	لغة أوبجكت باسكال
3	دلفي Delphi
3	فري باسكال Free Pascal
4	لازاراس Lazarus
4	مميزات لغة باسكال
5	المؤلف: معتر عبد العظيم الطاهر
5	ترخيص الكتاب
5	بيئة التعليم المستخدمة مع هذا الكتاب
6	استخدام البيئة النصية
6	الأمثلة

الفصل الأول

أساسيات اللغة

12	البرنامج الأول
14	شاشة المخرجات
18	تعددية الأنظمة portability
20	تجارب أخرى
22	المتغيرات Variables
26	الأنواع الفرعية
28	التفرعات المشروطة Conditional Branching
28	عبارة الشرط If condition
28	برنامج مكيف الهواء
31	برنامج الأوزان
33	عبارة الشرط Case .. of
33	برنامج المطعم
34	برنامج المطعم باستخدام عبارة if
35	برنامج درجة الطالب
35	برنامج لوحة المفاتيح
37	الحلقات loops
37	حلقة for

38	جدول الضرب باستخدام for loop
39	برنامج المضروب Factorial
40	حلقة Repeat Until
40	برنامج المطعم باستخدام Repeat Until
42	حلقة while do
43	برنامج المضروب باستخدام حلقة while do
44	المقاطع strings
47	الدالة Copy
48	الإجراء Insert
49	الإجراء Delete
49	الدالة Trim
51	الدالة StringReplace
53	المصفوفات arrays
56	السجلات Records
58	الملفات files
60	الملفات النصية text files
60	برنامج قراءة ملف نصي
63	برنامج إنشاء وكتابة ملف نصي
66	الإضافة إلى ملف نصي
66	برنامج الإضافة إلى ملف نصي
67	ملفات الوصول العشوائي Random access files
67	الملفات ذات النوع typed file
67	برنامج تسجيل درجات الطلاب
68	برنامج قراءة ملف الدرجات
69	برنامج إضافة درجات الطلاب
71	برنامج إنشاء وإضافة درجات الطلاب
72	برنامج سجل السيارات
74	نسخ الملفات Files copy
74	برنامج نسخ الملفات عن طريق البايت
76	الملفات غير محددة النوع untyped files
76	برنامج نسخ الملفات باستخدام الملفات غير محددة النوع
79	برنامج عرض محتويات رموز الملف
81	التاريخ والوقت Date and Time
83	مقارنة التواريخ والأوقات

85.....	مسجل الأخبار
86.....	الثوابت constants
86.....	برنامج استهلاك الوقود
88.....	Ordinal types الأنواع العددية
90.....	sets المجموعات
92.....	Exception handling معالجة الاعتراضات
92.....	try except عبارة
94.....	try finally عبارة
95.....	raise an exception رفع الاستثناءات

الفصل الثاني

البرمجة الهيكلية

Structured Programming

98.....	مقدمة
98.....	procedures الإجراءات
99.....	Parameters المدخلات
100.....	برنامج المطعم باستخدام الإجراءات
102.....	functions الدوال
103.....	repeat بحلقة باستخدام الدوال
104.....	Local Variables المتغيرات المحلية
106.....	برنامج قاعدة بيانات الأخبار
109.....	الدالة كمدخل
110.....	مخرجات الإجراءات و الدوال
111.....	calling by reference النداء باستخدام المرجع
113.....	units الوحدات
115.....	الوحدات وبنية لازاراس وفري باسكال
116.....	الوحدات التي يكتبها المبرمج
117.....	وحدة التاريخ الهجري
120.....	Procedure and function Overloading تحميل الإجراءات و الدوال
121.....	default parameters القيم الافتراضية للمدخلات
122.....	sorting ترتيب البيانات
122.....	bubble sort خوارزمية ترتيب الفقاعة

125.....	برنامج ترتيب درجات الطلاب :
127.....	خوارزمية الترتيب الاختياري Selection Sort
129.....	خوارزمية الترتيب Shell
131.....	ترتيب المقاطع
131.....	برنامج ترتيب الطلاب بالأسماء
132.....	مقارنة خوارزميات الترتيب

الفصل الثالث

الواجهة الرسومية

Graphical User Interface

137.....	مقدمة
138.....	البرنامج ذو الواجهة الرسومية الأول
144.....	البرنامج الثاني: برنامج إدخال الاسم
145.....	برنامج الـ ListBox
147.....	برنامج محرر النصوص Text Editor
149.....	برنامج الأخبار
151.....	برنامج الفورم الثاني
151.....	برنامج المحول الهجري

الفصل الرابع

البرمجة الكائنية

Object Oriented Programming

156.....	مقدمة
156.....	المثال الأول، برنامج التاريخ والوقت
161.....	الكبسلة Encapsulation
163.....	برنامج الأخبار بطريقة كائنية
169.....	برنامج الصفوف
175.....	الملف الكائني Object Oriented File
175.....	برنامج نسخ الملفات بواسطة TFileStream
176.....	الوراثة Inheritance

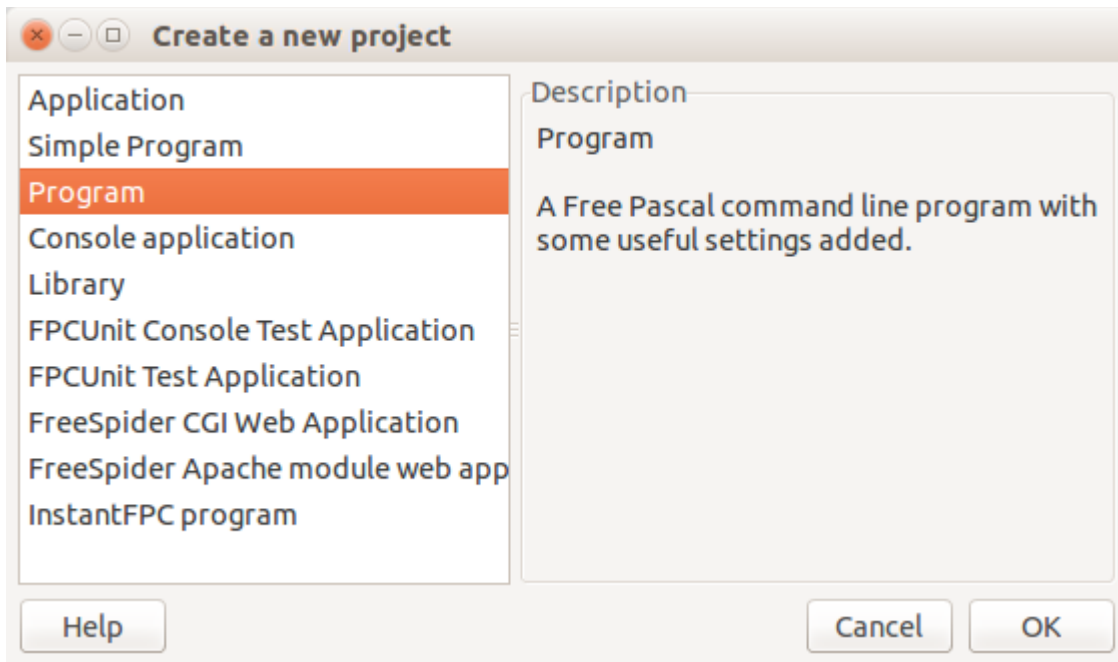
الفصل الأول

أساسيات اللغة

البرنامج الأول

بعد تثبيت لازاراس وتشغيله، نُنشئ برنامج جديد باستخدام الأمر التالي من القائمة الرئيسية:

Project/New Project/Program



لنحصل على الكود التالي في ال Source Editor:

```
program Project1;  
  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes  
  { you can add units after this };  
  
begin  
end.
```

بعد ذلك نحفظ البرنامج عن طريق *File/Save* من القائمة الرئيسية، في مجلد معروف بالنسبة لنا، ثم نسميه مثلاً *first*

في حالة استخدام نظام لينكس نكتب الأمر التالي بين عبارتي *.begin end*.

```
Writeln('This is Free Pascal and Lazarus');
```

أما في حالة وندوز فنكتب الأسطر التالية:

```
Writeln('This is Free Pascal and Lazarus');  
Writeln('Press enter key to close');  
Readln;
```

ليصبح البرنامج الكامل كالآتي:

```
program first;  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes  
  { you can add units after this };  
  
begin  
  Writeln('This is Free Pascal and Lazarus');  
  
  // For Windows only:  
  Writeln('Press enter key to close');  
  Readln;  
end.
```

عبارة *Writeln* تكتب نص في الشاشة. أما عبارة *Readln* فتوقف شاشة المخرجات حتى يتسنى قراءتها، وبعد الضغط على زر *Enter* يتوقف البرنامج ونرجع إلى حالة كتابة الكود.

بعد ذلك ننفذ البرنامج عن طريق المفتاح *F9* أو بالضغط على الزر:



لنحصل على المخرجات التالية:

```
This is Free Pascal and Lazarus
```

بعد تشغيل البرنامج والضغط على مفتاح الإدخال، ينتهي البرنامج ونرجع إلى الحالة الأولى، وهي حالة كتابة الكود. إذا كنا نستخدم نظام لينكس نجد على القرص ملف باسم *first*، وإذا كنا نستخدم وندوز نجد ملف باسم *first.exe* وهو ملفات الثنائية تنفيذية يمكن تشغيله في أي جهاز آخر لا يحتوي بالضرورة على لازاراس أو فري باسكال. هذه البرامج تكون عبارة عن برامج تطبيقية طبيعية بالنسبة لنظام التشغيل (Native Applications).

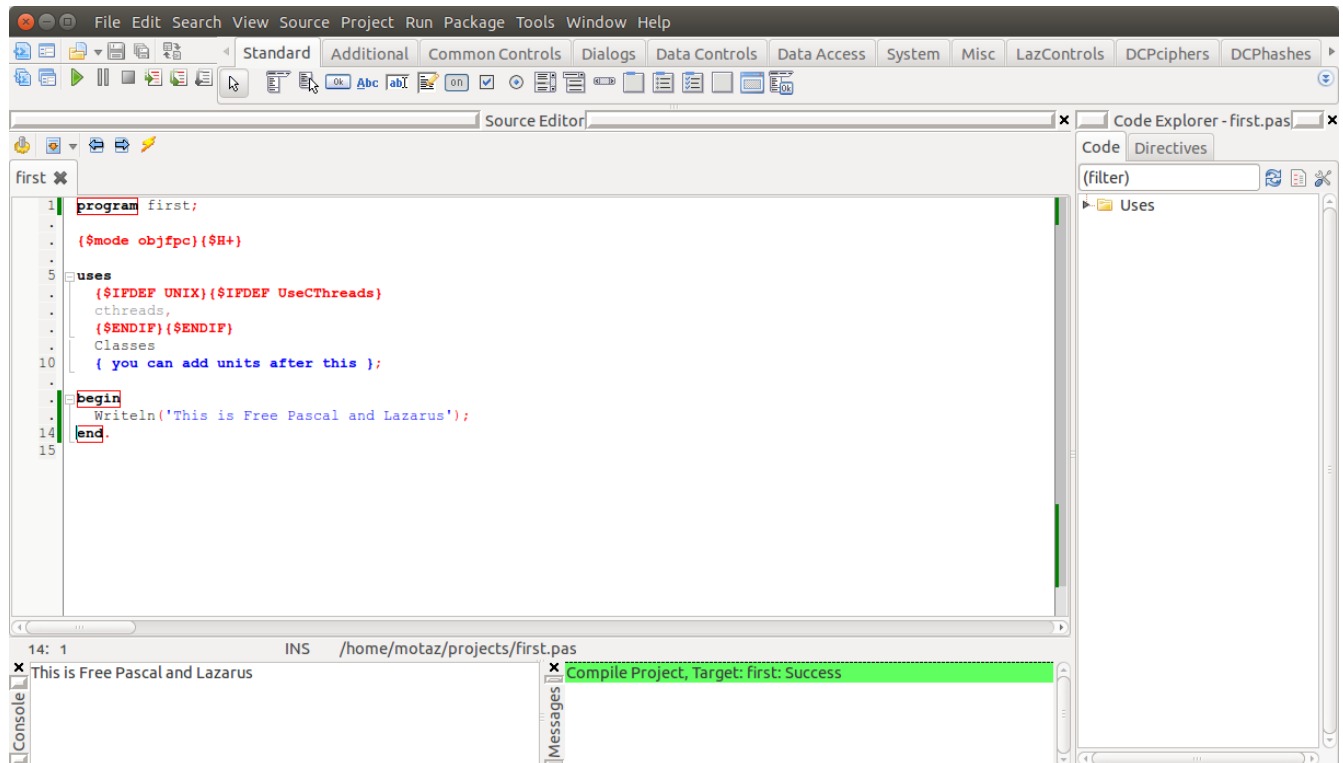
بالنسبة لنظام لينكس عند تشغيل البرنامج الجديد يكون في وضع Debug Mode لذلك لا تظهر شاشة المخرجات

شاشة المخرجات

في نظام لينكس تظهر مخرجات البرنامج السابق في شاشة تسمى terminal output

يمكن إظهارها بالضغط على مفاتيح Alt+Ctrl+O

يمكننا تحريك هذه الشاشة (terminal output) في مكان ظاهر لأننا سوف نستخدمها بكثرة لإظهار نتائج البرامج. وهي تظهر في الشاشة التالية في اليسار في الركن الأسفل.

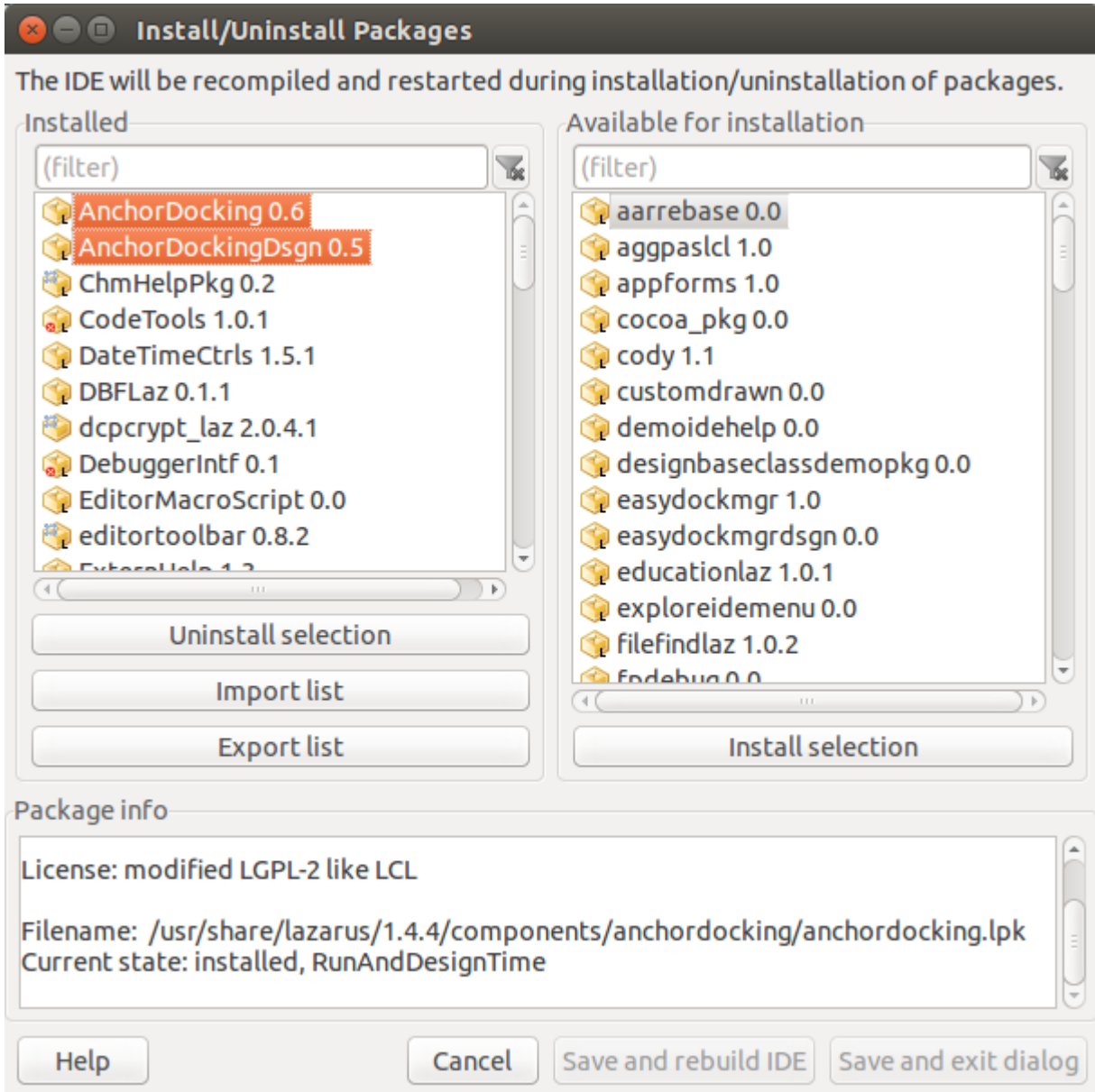


```
1 program first;
.
. {$mode objfpc} {$H+}
.
5 uses
. {$IFDEF UNIX} {$IFDEF UseCThreads}
. cthreads,
. {$ENDIF} {$SENDIF}
. Classes
10 ( you can add units after this );
.
. begin
. Writeln('This is Free Pascal and Lazarus');
14 end.
15
```

14: 1 INS /home/motaz/projects/first.pas
This is Free Pascal and Lazarus
Compile Project, Target: first: Success

لإظهار شاشة لازاراس بالشكل السابق والتي تُسمى (Docked IDE) نختار
packages/Install/Uninstall packages

ثم اختيار Anchor Docking التي تظهر باليمين ثم نضغط على الزر install selection لتظهر على
اليسار، ثم نضغط على زر Save and Rebuild IDE كما في الشكل التالي:



أما في نظام وندوز فلا بد من إضافة هذين السطرين في نهاية كل برنامج نصي:

```
Writeln('Press enter to close');  
Readln;
```

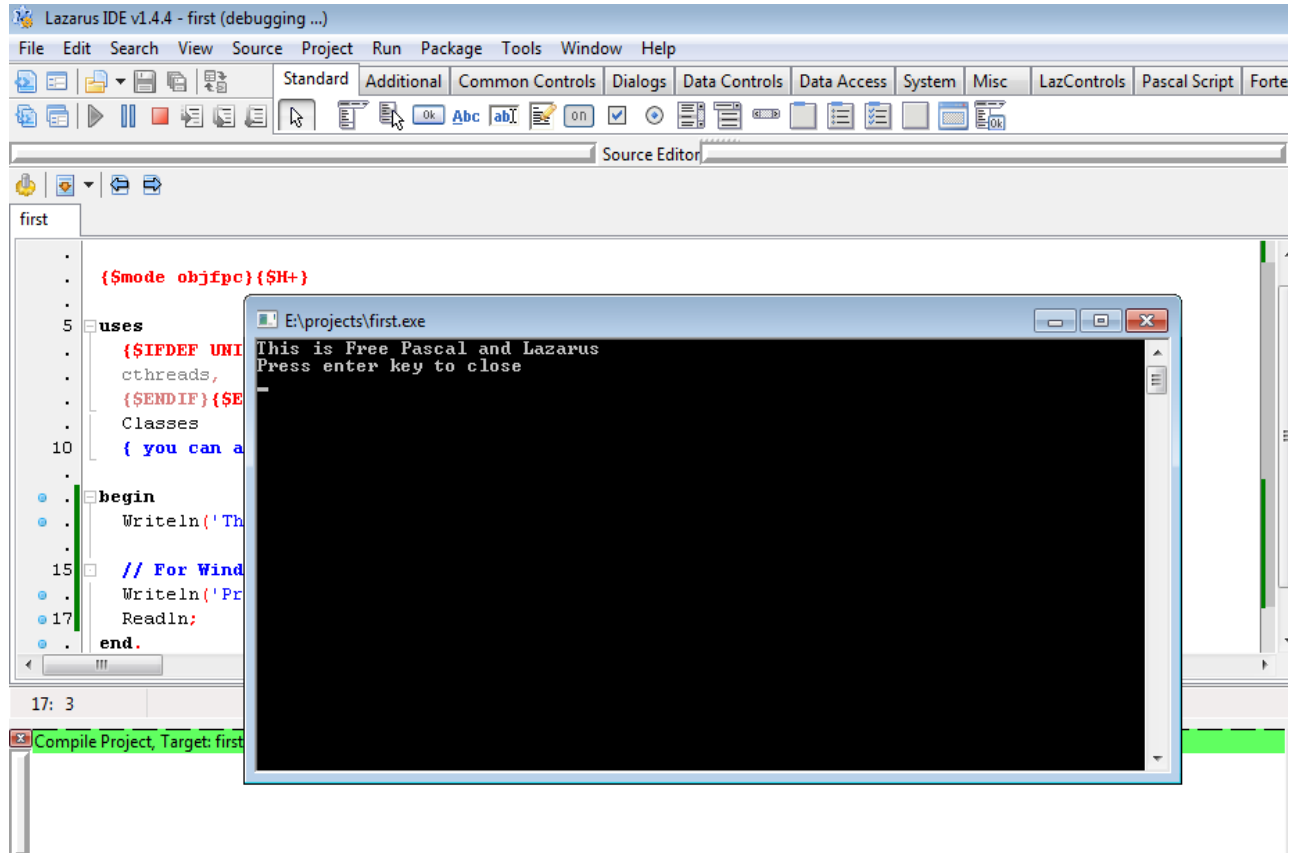
هذه الأسطر موجودة في كل الأمثلة النصية لكنها معطلة بواسطة // أي أنها لا تعمل إلا بإزالة هذين الخططين:

```
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;
```

عند إزالتها يصبحان كالتالي:

```
//For Windows, please uncomment below lines  
Writeln('Press enter key to continue..');  
Readln;
```

ومهمتها الإبقاء على شاشة ال terminal مفتوحة حتى نستطيع قراءة المخرجات. حيث تظهر الشاشة كالتالي:



ملحوظة:

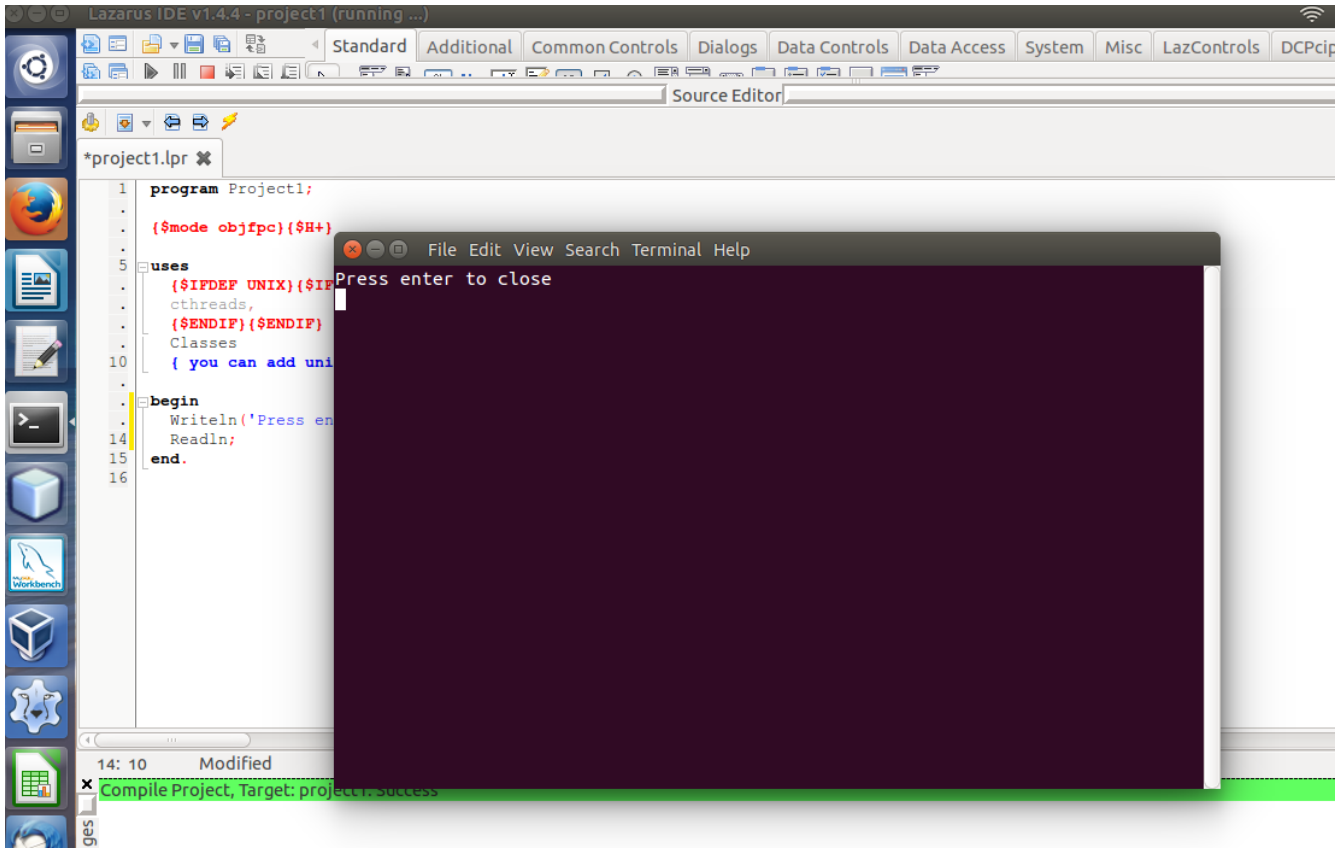
يمكن استخدام نفس الطريقة في نظام لينكس، وذلك بتعطيل المنقح *debugger* من

tools/options/Debugger

ثم اختيار *None* في خيار *Debugger type and path*

عندها يمكننا استخدام *readln* لتثبيت المخرجات لقراءتها ثم الضغط على زر Enter بنفس طريقة

وندوز:



تعددية الأنظمة portability

إذا كُنّا في بيئة لينكس مثلاً فإن الملفات التنفيذية الناتجة عن مترجم فري باسكال يُمكن توزيعها لتشغيلها فقط في نظام لينكس، أما إذا أردنا إصدار ملف تنفيذي لتوزيعه في بيئة وندوز أو ماكنتوش فعلياً نقل مصدر البرنامج source code إلى النظام الذي نريده ثم نُعيد ترجمته بواسطة لازاراس في تلك البيئة أو نظام التشغيل لإصدار ملفات تنفيذية لذلك النظام، لذلك فإن شعار فري باسكال/لازاراس هو

Write once and compile anywhere



أي نكتب الكود مرة واحدة ثم نعيد ترجمته في أي نظام تشغيل نريده.

هذه الحال نفسها مطبقة في لغة سي وسي ++، لذلك نجد أن البرامج المكتوبة بها توجد منها عدة نُسخ، مثلاً نجد في موقع موزيلا تحميل نسخة فيرفوكس لنظام وندوز، وأخرى لنظام لينكس ونسخة لنظام ماكنتوش. غالباً يتعرف المتصفح على نظام التشغيل الذي نستخدمه، فنجد مباشرة نسخة البرنامج المناسب لنظام التشغيل الذي نتصفح به النت.

بالنسبة لبيئة لازاراس توجد طرق لإصدار برامج تنفيذية لُنظم تشغيل غير المستخدم لتطوير البرامج، مثلاً يُمكن إصدار برامج تنفيذية لنظام وندوز من بيئة لينكس، وهذه العملية تُسمى cross-compilation و هي تحتاج إعدادات إضافية. بالإضافة لأن إعادة ترجمة البرنامج في البيئات الأخرى يتيح تجربته كافية لأنه أحياناً تكون هناك اختلافات بسيطة في سلوك البرامج من نظام تشغيل إلى نظام آخر، مثلاً حجم الخطوط قد لا يكون مطابق، وشكل الأزرار يختلف من بيئة لبيئة.

هناك موضوع آخر مهم يوضع في الحُسبان بالنسبة للملفات التنفيذية الناتجة، وهي معمارية النظام، هل هي 32 بت أم 64 بت. حيث أن نظام التشغيل 32 بت لا تعمل به برامج ذات 64 بت، وبعض أنظمة التشغيل 64 بت لا تعمل بها برامج 32 بت لنفس النظام. لذلك من الأفضل إنتاج برامج لكل هذه المعماريات.

في نظام لينكس يمكننا معرفة أي نوع ملف، خصوصاً الملفات التنفيذية باستخدام الأمر *file* كمثال:

```
file SMSTest
SMSTest: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.4.0, not
stripped
```

وهو يخبرنا بأن الملف هو ملف تنفيذي لنظام *GNU/Linux* ومعمارية *x68-64* أي معمارية 64 بت

أما هذا المثال:

```
file SMSTest.exe
SMSTest.exe: PE32 executable (GUI) Intel 80386, for MS Windows
```

فهو يخبرنا بأن الملف *SMSTest.exe* هو ملف تنفيذي لنظام وندوز ومعمارية *Intel 80386* وهي تعني 32 بت.

تجارب أخرى

في نفس البرنامج السابق نغير هذا السطر:

```
Writeln('This is Free Pascal and Lazarus');
```

بالسطر التالي:

```
Writeln('This is a number: ', 15);
```

ثم نشغل البرنامج لنشاهد المخرجات التالية:

```
This is a number: 15
```

نستبدل السطر السابق بالأسطر التالية ثم ننفذ البرنامج في كل مرة:

الكود:

```
Writeln('This is a number: ', 3 + 2);
```

المخرجات:

```
This is a number: 5
```

الكود:

```
Writeln('5 * 2 = ', 5 * 2);
```

المخرجات:

```
5 * 2 = 10
```

الكود:

```
Writeln('This is a real number: ', 7.2);
```

المخرجات:

```
7.20000000000000E+0000
```

الكود:

```
Writeln('One, Two, Three : ', 1, 2, 3);
```

المخرجات:

```
One, Two, Three : 123
```

الكود:

```
Writeln(10, ' * ', 3, ' = ', 10 * 3);
```

المخرجات:

```
10 * 3 = 30
```

يمكن كتابة أي قيم بأشكال مختلفة في عبارة *Writeln* لنرى ماهي النتائج.

المتغيرات Variables

المتغيرات هي عبارة حاويات للبيانات. فمثلاً في الطريقة الرياضية عندما نقول أن $s = 5$ فهذا يعني أن s هي متغير وهي في هذه اللحظة تحمل القيمة 5. كذلك يمكن إدخالها في عبارات رياضية، حيث أن قيمة s مضروبة في 2 ينتج عنها 10:

$$s = 5$$

$$s * 2 = 10$$

تتميز لغة باسكال بأنها تلتزم بنوع المتغير Strong Typed language، فحسب البيانات التي سوف نضعها لابد من تحديد نوع المتغير. وهذا المتغير سوف يحمل نوع واحد فقط من البيانات طوال تشغيل البرنامج، فمثلاً إذا عرّفنا متغير من النوع الصحيح، فإنه يمكننا فقط إسناد أرقام صحيحة له، و لا يمكننا إسناد عدد كسري مثلاً.

كذلك يجب التعريف عن المتغير قبل استخدامه كما في المثال التالي:

```
program FirstVar;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x: Integer;
begin
  x:= 5;
  Writeln(x * 2);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

فعند تنفيذه نحصل على القيمة 10.

في المثال أعلاه كتبنا الكلمة المحجوزة *Var* والتي تفيد بأن الأسطر القادمة عبارة عن تعريف للمتغيرات. وعبارة:

```
x: Integer;
```

تفيد شيئين: أولهما أن اسم المتغير الذي سوف نستخدمه هو x وأن نوعه *Integer* وهو نوع العدد الصحيح الذي يقبل فقط أرقام صحيحة لا تحتوي على كسور ويمكن أن تكون موجبة أو سالبة.

أما عبارة

```
x:= 5;
```

فهي تعني وضع القيمة 5 في المتغير الصحيح x

يمكن إضافة متغير آخر للبرنامج نسميه y مثلاً كما في المثال التالي:

```
var
  x, y: Integer;
begin
  x:= 5;
  y:= 10;
  Writeln(x * y);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نجد أن مخرجات البرنامج السابق هي:

```
50
```

في المثال التالي نختبر نوع جديد من المتغيرات، وهو متغير رمزي *character*

```
var
  c: Char;
begin
  c:= 'M';
  Writeln('My first letter is: ', c);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ أن المتغير الرمزي لابد من إحاطته بعلامة التنصيص ' ويُمكنه تخزين حرف أو رقم أو أي رمز آخر.

أما المثال التالي فهو لنوع الأعداد الحقيقية التي يمكن أن تحتوي على كسور:

```
var
  x: Single;
begin
  x:= 1.8;
  Writeln('My Car engine capacity is ', x, ' liters');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

لنتمكن من كتابة برامج أكثر تفاعلاً لابد من ذكر طريقة إدخال قيمة المتغيرات من المستخدم بدلاً من كتابتها في البرنامج. والطريقة البسيطة هي استخدام عبارة *Readln* التي تُمكن المستخدم من إدخال مدخلات حسب نوع المتغيرات كما في المثال التالي:

```
var
  x: Integer;
begin
  Write('Please input any number:');
  Readln(x);
  Writeln('You have entered: ', x);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

في هذه الحالة أصبح تخصيص القيمة للمتغير x هو عن طريق لوحة المفاتيح. ملحوظة:

لابد من التأكد أن المؤشر موجود في شاشة *terminal output* لإدخال القيمة x ثم نضغط على *enter* البرنامج التالي يحسب جدول الضرب لرقم يُختاره المستخدم:

```
program MultTable;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x: Integer;
begin
  Write('Please input any number:');
```



```
Readln(x);
Writeln(x, ' * 1 = ', x * 1);
Writeln(x, ' * 2 = ', x * 2);
Writeln(x, ' * 3 = ', x * 3);
Writeln(x, ' * 4 = ', x * 4);
Writeln(x, ' * 5 = ', x * 5);
Writeln(x, ' * 6 = ', x * 6);
Writeln(x, ' * 7 = ', x * 7);
Writeln(x, ' * 8 = ', x * 8);
Writeln(x, ' * 9 = ', x * 9);
Writeln(x, ' * 10 = ', x * 10);
Writeln(x, ' * 11 = ', x * 11);
Writeln(x, ' * 12 = ', x * 12);
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

الملاحظة المهمة في المثال السابق أن أي عبارة تُكتب بين علامتي تنصيص أحادية تكتب كما هي مثلاً:

```
' * 1 = '
```

أما ما يُكتب بدون تنصيص فإن قيمته هي التي تظهر. يمكن تجربة العبارتين التاليتين حتى يكون الفرق أوضح بن استخدام علامة التنصيص وعدم استخدامها:

```
Writeln('5 * 3');
Writeln(5 * 3);
```

فالعبارة تُكتب كما هي:

```
5 * 3
```

أما ناتج العبارة الثانية فيكون حاصل العملية الحسابية:

```
15
```

في المثال التالي سوف نجري عملية حسابية ونضع الناتج في متغير ثالث ثم نظهر قيمة هذا المتغير:

```
var
  x, y: Integer;
  Res: Single;
begin
  Write('Input a number: ');
  Readln(x);
  Write('Input another number: ');
```

```
Readln(y);
Res:= x / y;
Writeln(x, ' / ', y, ' = ', Res);
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

فبما أن العملية الحسابية هي قسمة وربما ينتج عنها عدد كسري (حقيقي) لذلك يرفض مترجم الباسكال وضع النتيجة في متغير صحيح، لذلك لابد أن يكون هذا المتغير *Res* عدد حقيقي. ونوع المتغير *Single* يُستخدم لتعريف متغيرات كسرية ذات دقة عشرية أحادية.

الأنواع الفرعية

توجد أنواع كثيرة للمتغيرات، فمثلاً الأعداد الصحيحة توجد منها *Byte, SmallInt, Integer, LongInt, Word*. وتختلف عن بعضها في مدى الأرقام وهي أصغر وأكبر عدد يمكن إسناده لها. كذلك تختلف في عدد خانات الذاكرة (البايت) المطلوبة لتخزين تلك المتغيرات.

النوع	الحجم بالبايت	أصغر قيمة	أكبر قيمة
Byte	1	0	255
ShortInt	1	-128	127
SmallInt	2	-32768	32767
Word	2	0	65535
Integer	4	-2,147,483,648	2,147,483,647
LongInt	4	-2,147,483,648	2,147,483,647
Cardinal	4	0	4,294,967,295
Int64	8	-9,223,372,036,854,780,000	9,223,372,036,854,780,000

يمكن معرفة مدى هذه الأنواع وعدد خانات الذاكرة التي تتطلبها باستخدام الدوال: *Low, High, SizeOf*. كما في المثال التالي:

```
program Types;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes;

begin
  Writeln('Byte: Size = ', SizeOf(Byte), ', Minimum value = ', Low(Byte),
    ', Maximum value = ', High(Byte));

  Writeln('Integer: Size = ', SizeOf(Integer),
    ', Minimum value = ', Low(Integer), ', Maximum value = ', High(Integer));
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

التفرعات المشروطة Conditional Branching

من أهم ما يُميز برامج الحاسوب أو أي جهاز إلكتروني آخر، هو إمكانية تنفيذ إجراء معين عند حدوث شرط معين. مثلاً نجد أن في بعض السيارات تتأمن أبوابها تلقائياً عند بلوغها سرعة معينة وهي غير مؤمنة. ففي هذه الحالة الشرط مركب هو وصول سرعة معينة في وجود أبواب غير مؤمنة، أما التفرع أو الإجراء فهو عملية تأمين الأبواب. أما في حالة عدم توفر الشرط وهو (بلوغ سرعة معينة مع وجود حالة الأبواب غير مؤمنة) فإن الإجراء (تأمين الأبواب) لا يُنفذ.

معظم السيارات والمصانع والغسالات الآلية و ما شابهها من الأجهزة تعمل بمعالج صغير أو ما يُسمى micro controller وهي دائرة مدمجة (IC) يمكن برمجتها بواسطة لغة أسمبلي أو لغة سي. كذلك فإن بعض أنواع هذه المعالجات التي تعمل في مثل هذه الأنظمة المدمجة embedded systems مثل المعالج آرم ARM يمكن برمجتها بواسطة فري باسكال، وهي توجد في الأجهزة المحمولة واللوحية وبعض أجهزة الألعاب، كذلك فإن أجهزة اللوح الواحد single board computer مثل جهاز راسبيري باي، فهو يستخدم معالج ARM ويمكن عمل برامج له بواسطة لازاراس/فري باسكال.

عبارة الشرط If condition

عبارة *if* الشرطية في لغة باسكال هي عبارة سهلة وواضحة. في المثال التالي سوف نستقبل من المستخدم درجة الحرارة ثم نحكم هل نُشغل جهاز التكييف أم لا:

برنامج مكيف الهواء:

```
var
  Temp: Single;
begin
  Write('Please enter Temperature of this room :');
  Readln(Temp);
  if Temp > 28 then
    Writeln('Please turn on air-condition')
  else
    Writeln('Please turn off air-condition');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نجد أن العبارات الجديدة هي: *if then else* وهي تعني: إذا كانت درجة الحرارة أكبر من 28 أكتب السطر الأول (*Please turn on air-condition*) وإذا لم يتحقق الشرط (رقم أصغر أو يساوي 28) في هذه الحالة أكتب السطر الثاني (*Please turn off air-condition*).

يمكن كتابة شروط متتالية كما في المثال التالي:

```
var
  Temp: Single;
begin
  Write('Please enter Temperature of this room :');
  Readln(Temp);
  if Temp > 28 then
    Writeln('Please turn on air-condition')
  else
    if Temp < 25 then
      Writeln('Please turn off air-condition')
    else
      Writeln('Do nothing');
```

نُفذ البرنامج أعلاه عدة مرات و نُدخل قيم تتراوح بين 20 و 30 ثم نُشاهد النتائج.

يمكن تعقيد البرنامج السابق ليصبح أكثر واقعية كما في المثال أدناه:

```
var
  Temp: Single;
  ACIsOn: Byte;
begin
  Write('Please enter Temperature of this room : ');
  Readln(Temp);
  Write('Is air condition on? if it is (On) write 1, if it is (Off) write 0 : ');
  Readln(ACIsOn);

  if (ACIsOn = 1) and (Temp > 28) then
    Writeln('Do no thing, we still need cooling')
  else
    if (ACIsOn = 1) and (Temp < 25) then
      Writeln('Please turn off air-condition')
    else
      if (ACIsOn = 0) and (Temp < 25) then
        Writeln('Do nothing, it is still cold')
      else
        if (ACIsOn = 0) and (Temp > 28) then
          Writeln('Please turn on air-condition')
        else
          Writeln('Please enter valid values');
```

```
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;  
end.
```

نجد في المثال السابق وجود عبارة *and* وهي تعني إذا تحقق الشرطين الأول والثاني نفذ العبارة التالية.

وهذا هو تفسير البرنامج السابق:

-إذا كان المكيف يعمل ودرجة الحرارة أكبر من 28 أكتب: *Do no thing, we still need cooling*. أما إذا لم ينطبق هذا الشرط (*else*) نذهب إلى الإجراء التالي:

-إذا كان المكيف يعمل ودرجة الحرارة أقل من 25 أكتب: *Please turn off air-condition*. أما إذا لم ينطبق الشرط نذهب للإجراء التالي:

-إذا كان المكيف مغلق ودرجة الحرارة أقل من 25 أكتب: *Do nothing, it is still cold*. أما إذا لم ينطبق الشرط نذهب للإجراء التالي:

-إذا كان المكيف مغلق ودرجة الحرارة أكبر من 28 أكتب: *Please turn on air-condition*.

- أما إذا لم ينطبق هذا الشرط فهو يعني أن المستخدم أدخل قيمة غير ال 0 أو 1 في المتغير *ACIsOn*. وفي هذه الحالة نُنبهه بإدخال قيم صحيحة: *Please enter valid values*.

إذا افترضنا أن جهاز الحاسوب أو أي جهاز آخر يمكن تشغيله بالاسكال به موصل بالمكيف وأن هناك إجراء لفتح المكيف وآخر لغلاقه في هذه الحال يمكننا استبدال إجراء *Writeln* بإجراءات حقيقية تُنفذ الفتح والإغلاق. في هذه الحالة يجب زيادة الشروط لتصبح أكثر تعقيداً تحسباً للمدة الزمنية التي عمل بها المكيف، كذلك يمكن الأخذ في الاعتبار هل الوقت في الصباح أم المساء.

برنامج الأوزان

في المثال التالي يُدخل المستخدم طوله بالأمتار ووزنه بالكيلو جرام، ثم يحسب البرنامج الوزن المثالي بناءً على طول المستخدم. ويقارن الوزن المثالي بالوزن الحالي للمستخدم، ويطبع النتائج حسباً لفرق الوزن:

```
program Weight;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  Height: Double;
  Weight: Double;
  IdealWeight: Double;
begin
  Write('What is your height in meters (e.g. 1.8 meter) : ');
  Readln(Height);
  Write('What is your weight in kilos : ');
  Readln(Weight);
  if Height >= 1.4 then
    IdealWeight:= (Height - 1) * 100
  else
    IdealWeight:= Height * 20;

  if (Height < 0.4) or (Height > 2.5) or (Weight < 3) or
    (Weight > 200) then
    begin
      Writeln('Invalid values');
      Writeln('Please enter proper values');
    end
  else
    if IdealWeight = Weight then
      Writeln('Your weight is suitable')
    else
      if IdealWeight > Weight then
        Writeln('You are under weight, you need to gain ',
          Format('%.2f', [IdealWeight - Weight]), ' Kilos')
      else
        Writeln('You are over weight, you need to lose ',
          Format('%.2f', [Weight - IdealWeight]), ' Kilos');
        //For Windows, please uncomment below lines
        //writeln('Press enter key to continue..');
        //readln;
end.
```

في المثال السابق نجد عدة أشياء جديدة:

النوع *Double*: وهو مشابه للنوع *Single* فكلاهما عدد حقيقي، أو ما يُسمى بالرقم ذو الفاصلة العائمة (floating point number). والنوع *Double* هو رقم حقيقي ذو دقة مضاعفة، وهو يحتل مساحة ضعف النوع *single* حيث أن الأخير يحتل مساحة 4 بايت من الذاكرة أما الـ *Double* فهو يحتاج إلى 8 بايت من الذاكرة لتخزينه.

الشيء الثاني هو استخدامنا للكلمة *Or*، وهي بخلاف *And* ومن معناها تفيد تحقيق أحد الشروط. مثلاً إذا تحقق الشرط الأول ($Height < 0.4$) فإن البرنامج يُنفذ إجراء الشرط، ولو لم يتحقق سوف يُختبر الشرط الثاني، فإذا تحقق، يُنفذ إجراء الشرط، وإذا لم يتحقق الشرط الثاني فإن البرنامج يختبر الشرط الثالث وهكذا..

الشيء الثالث هو استخدامنا لعبارتي *begin* و *end* بعد عبارة *if* وذلك لأن إجراء الشرط لابد أن يكون عبارة واحدة، وفي هذه الحالة احتجنا لأن ننفذ عبارتين وهما:

```
Writeln('Invalid values');  
Writeln('Please enter proper values');
```

لذلك استخدمنا *begin end* لتحويل العبارتين إلى كتلة واحدة أو عبارة واحدة تصلح لأن تكون إجراء الشرط:

```
if (Height < 0.4) or (Height > 2.5) or (Weight < 3) or  
(Weight > 200) then  
begin  
  Writeln('Invalid values');  
  Writeln('Please enter proper values');  
end
```

الشيء الرابع هو استخدام الدالة *Format*. ولا يمكن استخدام هذه الدالة إلا بعد إضافة الوحدة *SysUtils* في عبارة *Uses*، والوحدة *SysUtils* هي عبارة عن مكتبة تحتوي على إجراء *Format*، وسوف نفصل الوحدات أو المكتبات و الإجراءات في فصل البرمجة الهيكلية إن شاء الله. هذه الدالة تُظهر المتغيرات بشكل معين، مثلاً في المثال السابق تُظهر العدد الحقيقي في شكل رقم يحتوي على فاصلة عشرية ذات خانيتين كما يظهر في التنفيذ. مثلاً

```
What is your height in meters (e.g. 1.8 meter) : 1.8
```



```
What is your weight in kilos : 60.2  
You are under weight, you need more 19.80 Kilos
```

ملحوظة: حساب الوزن المثالي في البرنامج السابق ربما تكون غير دقيقة. ولمزيد من الدقة يمكن البحث عن الموضوع في الإنترنت. والمقصود بهذا البرنامج كيفية وضع حلول برمجية لمثل هذه الأمثلة. فلا بد للمبرمج أن يعرف كيف يضع حل لمسألة معينة، لأن المعرفة بالبرمجة وأدواتها لا تكفي لأن يصبح المبرمج قادر على تحليل وتصميم برامج يعتمد عليها، لذلك لابد من الدخول في تفاصيل الموضوع أو المشكلة وتحلل قبل وضع البرنامج المناسب.

عبارة الشرط of .. Case

توجد طريقة أخرى للتفرع المشروط وهو استخدام عبارة *Case .. of*. مثلاً نريد استقبال طلب للزبون في مطعم، والوجبات مرقمة في قائمة رئيسة، ونريد من الزبون اختيار رقم الطلب كما في المثال التالي:

برنامج المطعم

```
var  
Meal: Byte;  
begin  
  
Writeln('Welcome to Pascal Restaurant. Please select your order');  
Writeln('1 - Chicken      (10 Geneh)');  
Writeln('2 - Fish          (7 Geneh)');  
Writeln('3 - Meat            (8 Geneh)');  
Writeln('4 - Salad           (2 Geneh)');  
Writeln('5 - Orange Juice (1 Geneh)');  
Writeln('6 - Milk            (1 Geneh)');  
Writeln;  
Write('Please enter your selection: ');  
Readln(Meal);  
  
case Meal of  
1: Writeln('You have ordered Chicken, this will take 15 minutes');  
2: Writeln('You have ordered Fish, this will take 12 minutes');  
3: Writeln('You have ordered meat, this will take 18 minutes');  
4: Writeln('You have ordered Salad, this will take 5 minutes');  
5: Writeln('You have ordered Orange juice, ',  
          'this will take 2 minutes');  
6: Writeln('You have ordered Milk, this will take 1 minute');  
else  
Writeln('Wrong entry');
```

```
end;  
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;  
end.
```

إذا أردنا عمل نفس البرنامج باستخدام عبارة `if` فإن البرنامج سوف يكون أكثر تعقيداً ويحتوي على تكرار.

برنامج المطعم باستخدام عبارة `if`

```
var  
Meal: Byte;  
begin  
  Writeln('Welcome to Pascal restaurant, please select your meal');  
  Writeln('1 - Chicken      (10 Geneh)');  
  Writeln('2 - Fish         (7 Geneh)');  
  Writeln('3 - Meat           (8 Geneh)');  
  Writeln('4 - Salad          (2 Geneh)');  
  Writeln('5 - Orange Juice (1 Geneh)');  
  Writeln('6 - Milk            (1 Geneh)');  
  Writeln;  
  Write('Please enter your selection: ');  
  Readln(Meal);  
  
  if Meal = 1 then  
    Writeln('You have ordered Chicken, this will take 15 minutes')  
  else  
    if Meal = 2 then  
      Writeln('You have ordered Fish, this will take 12 minutes')  
    else  
      if Meal = 3 then  
        Writeln('You have ordered meat, this will take 18 minutes')  
      else  
        if Meal = 4 then  
          Writeln('You have ordered Salad, this will take 5 minutes')  
        else  
          if Meal = 5 then  
            Writeln('You have ordered Orange juice, ',  
              'this will take 2 minutes')  
          else  
            if Meal = 6 then  
              Writeln('You have ordered Milk, this will take 1 minute')  
            else  
              Writeln('Wrong entry');  
            //For Windows, please uncomment below lines  
            //writeln('Press enter key to continue..');  
            //readln;  
          end.  
        end.  
      end.  
    end.  
  end.
```

يمكن كذلك استخدام مدى للأرقام بعبارة *case*. في المثال التالي يُقيم البرنامج درجة الطالب:

برنامج درجة الطالب

```
var
  Mark: Integer;
begin
  Write('Please enter student mark: ');
  Readln(Mark);
  Writeln;

  case Mark of
    0 .. 39 : Writeln('Student grade is: F');
    40 .. 49: Writeln('Student grade is: E');
    50 .. 59: Writeln('Student grade is: D');
    60 .. 69: Writeln('Student grade is: C');
    70 .. 84: Writeln('Student grade is: B');
    85 .. 100: Writeln('Student grade is: A');
  else
    Writeln('Wrong mark');
  end;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ في البرنامج السابق أننا استخدمنا مدى مثل 85 .. 100 أي أن الشرط هو أن تكون الدرجة بين هاتين القيمتين.

عبارة *case* لا تعمل إلا مع المتغيرات المعدودة مثل الأعداد الصحيحة أو الحروف، لكنها لا تعمل مع بعض الأنواع الأخرى مثل الأعداد الحقيقية أما المقاطع فهي تعمل فقط مع مترجم فري باسكال ولا تعمل مع مترجم دلفي.

برنامج لوحة المفاتيح

في البرنامج التالي نستخدم نوع المتغيرات الحرفية *char* وذلك لاستقبال حرف من لوحة المفاتيح ومعرفة موقعه:

```
var
  Key: Char;
```

```
begin
  Write('Please enter any English letter: ');
  Readln(Key);
  Writeln;

  case Key of
    'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p':
      Writeln('This is in the second row in keyboard');
    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l':
      Writeln('This is in the third row in keyboard');
    'z', 'x', 'c', 'v', 'b', 'n', 'm':
      Writeln('This is in the fourth row in keyboard');
  else
    Writeln('Unknown character');
  end;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ في المثال السابق أننا لم نستخدم المدى لكن استخدمنا تعدد الخيارات، مثلاً إذا اخترنا العبارة الأخيرة:

```
'z', 'x', 'c', 'v', 'b', 'n', 'm':
```

فهي تعني إذا كانت قيمة *key* هي *z* أو *x* أو *c* أو *v* أو *b* أو *n* أو *m* نفذ إجراء الشرط.

يمكن كذلك المزج بين صيغة المدى وتعدد الخيارات مثل:

```
'a' .. 'd', 'x', 'y', 'z':
```

وهي تعني اختبار الحرف إذا كان في المدى من الحرف *a* إلى الحرف *d* أو كان هو الحرف *x* أو *y* أو *z* نفذ الشرط.

الحلقات loops

الحلقات هي من المواضيع المهمة والعملية في البرمجة، فهي تعني الاستمرار في تنفيذ جزء معين من العبارات بوجود شرط معين. وعندما ينتهي أو ينتفي هذا الشرط تتوقف الحلقة عن الدوران.

حلقة for

يمكن تكرار عبارة معينة بعدد معين باستخدام *for* كما في المثال التالي:

```
var
  i: Integer;
  Count: Integer;
begin
  Write('How many times? ');
  Readln(Count);
  for i:= 1 to Count do
    Writeln('Hello there');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نستخدم في حلقات *for* متغير صحيح يسمى متغير الحلقة، وهو في هذا المثال المتغير *i*، وقيمه تبدأ في الدورة الأولى بالقيمة الابتدائية التي حددها المبرمج، في هذه الحالة هو الرقم *1* ثم يزيد هذا المتغير في كل دورة حتى ينتهي في الدورة الأخيرة بالقيمة الأخيرة، وهي التي يحددها المستخدم بإدخاله لقيمة *Count*

يمكن إظهار قيمة متغير الحلقة كما في التعديل التالي للمثال السابق:

```
var
  i: Integer;
  Count: Integer;
begin
  Write('How many times? ');
  Readln(Count);
  for i:= 1 to Count do
    begin
      Writeln('Cycle number: ', i);
      Writeln('Hello there');
    end;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
```

```
//readln;  
end.
```

نلاحظ أن حلقة for تُنفذ عبارة واحدة فقط، وهي العبارة التي تليها، لكن هذه المرة احتجنا لتنفيذ عبارتين، لذلك حولناهما إلى عبارة واحدة باستخدام *.begin end*.

جدول الضرب باستخدام for loop

لو قارنا بين برنامج جدول الضرب السابق والتالي الذي سوف نستخدم فيه عبارة for سوف نجد أن الأخير ليس فيه تكرار كالأول:

```
program MultTableWithForLoop;  
  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes  
  { you can add units after this };  
  
var  
  x, i: Integer;  
begin  
  Write('Please input any number: ');  
  Readln(x);  
  for i:= 1 to 12 do  
    Writeln(x, ' * ', i, ' = ', x * i);  
  //For Windows, please uncomment below lines  
  //writeln('Press enter key to continue..');  
  //readln;  
end.
```

نجد أننا بدلاً عن كتابة إجراء إظهار حاصل الضرب 12 مرة فقد تمت كتابته مرة واحدة فقط وتولت حلقة for تكرار هذا الإجراء 12 مرة.

يمكن جعل الحلقة تدور بالعكس، من القيمة الكبرى إلى القيمة الصغرى وذلك باستخدام *downto* وذلك بتغيير سطر واحد في المثال السابق لجدول الضرب:

```
for i:= 12 downto 1 do
```

برنامج المضروب Factorial

المضروب هو مجموع حاصل ضرب الرقم مع الرقم الذي يسبقه إلى الرقم واحد:

مضروب 3 يساوي $1 * 2 * 3$ وتنتج عنه القيمة: 6

```
var
  Fac, Num, i: Integer;
begin
  Write('Please input any number: ');
  Readln(Num);
  Fac:= 1;
  for i:= Num downto 1 do
    Fac:= Fac * i;
  Writeln('Factorial of ', Num , ' is ', Fac);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

حلقة Repeat Until

بخلاف حلقة for loop التي تتميز بأن دوراتها تكون محدودة بعدد معين فإن *Repeat loop* تدور إلى أن يتحقق شرط معين، فمادام هذا الشرط غير مُحقق فهي تعيد الدوران، فإذا تحقق الشرط فسوف يخرج مؤشر التنفيذ من هذه الحلقة و يُنفذ ما بعدها. لذلك نجد أن هذه الحلقة غير محددة بعدد معين من الدورات.

```
var
  Num : Integer;
begin
  repeat
    Write('Please input a number: ');
    Readln(Num);
  until Num <= 0;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

في البرنامج السابق يدخل البرنامج للحلقة أولاً ثم يسأل المستخدم أن يكتب عدداً، ثم في نهاية الحلقة يفحص قيمة هذا العدد فإذا ساوي الصفر أو عدد أقل منه فإن الشرط يكون قد تحقق ثم تنتهي الحلقة، أما إذا أدخلنا فيها أرقام أكبر من الصفر فإن الحلقة تستمر في الدوران.

برنامج المطعم باستخدام Repeat Until

```
var
  Selection: Char;
  Price: Integer;
  Total: Integer;
begin
  Total:= 0;
  repeat
    Writeln('Welcome to Pascal Restaurant. Please select your order');
    Writeln('1 - Chicken      (10 Geneh)');
    Writeln('2 - Fish          (7 Geneh)');
    Writeln('3 - Meat            (8 Geneh)');
    Writeln('4 - Salad           (2 Geneh)');
    Writeln('5 - Orange Juice   (1 Geneh)');
    Writeln('6 - Milk            (1 Geneh)');
    Writeln('X - nothing');
    Writeln;
    Write('Please enter your selection: ');
```



```
Readln(Selection);
Price:= 0;

case Selection of
  '1': begin
    Writeln('You have ordered Chicken, ',
            'this will take 15 minutes');
    Price:= 10;
  end;
  '2': begin
    Writeln('You have ordered Fish, ',
            ' this will take 12 minutes');
    Price:= 7;
  end;
  '3': begin
    Writeln('You have ordered meat, ',
            ' this will take 18 minutes');
    Price:= 8;
  end;
  '4': begin
    Writeln('You have ordered Salad, '
            ' this will take 5 minutes');
    Price:= 2;
  end;
  '5': begin
    Writeln('You have ordered Orange juice, ',
            'this will take 2 minutes');
    Price:= 1;
  end;
  '6': begin
    Writeln('You have ordered Milk, ',
            'this will take 1 minute');
    Price:= 1;
  end;
else
begin
  Writeln('Wrong entry');
end;
end;

Total:= Total + Price;

until (Selection = 'x') or (Selection = 'X');
Writeln('Total price          = ', Total);
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

في البرنامج السابق هناك عدة أشياء جديدة وهي:

1. أضفنا *begin end case* وذلك لأن تنفيذ الشرط لابد أن يكون عبارة واحدة، وكما سبق ذكره فإن

begin end تحوّل عدة عبارات إلى عبارة واحدة. فبعد عبارة *WriteIn('You have ordered ..* أضفنا عبارة جديدة لها وهي *price:= 2* وهي لمعرفة قيمة الطلب لإضافته لاحقاً لمعرفة الحساب الكلي.

1. استخدمنا المتغير *selection* من النوع الحرفي *char* وهو يتميز بإمكانيته في استقبال أي رمز من لوحة المفاتيح، فمثلاً يمكن أن يستقبل رقم (خانة واحدة فقط) أو حرف. وقد استخدمنا الحرف *x* للدلالة على نهاية الطلبات.

2. وضعنا صفر في المتغير *Total* والذي سوف نُجمع قيمة الطلبات فيه ثم نعرضه في النهاية. وقد استخدمنا هذه العبارة للتجميع:

```
Total:= Total + Price;
```

3. وضعنا خيارين لإدخال الحرف *x* فإذا أدخل المستخدم *x* كبيرة *X* أو صغيرة *x* فإن الحلقة تتوقف. علماً بأن قيمة الحرف *x* تختلف عن قيمته بالصيغة الكبيرة *X*.

حلقة while do

تتشابه حلقة *while* مع حلقة *repeat* بارتباط الاستمرار في الدوران مع شرط معين، إلا أن *while* تختلف اختلافين:

1. يُفحص الشرط قبل الدخول إلى الحلقة
2. حلقة *repeat* تُنفذ مرة واحدة على الأقل، وبعد الدورة الأولى يُفحص الشرط، أما *while* فهي تمنع البرنامج من دخول الحلقة إذا كان الشرط غير صحيح من البداية.
3. إذا كانت الحلقة تحتوي على أكثر من عبارة فلا بد من استخدام *begin end*. أما حلقة *repeat* فإنها لا تحتاج لذلك، لأن الحلقة حدودها تبدأ من كلمة *repeat* وتنتهي بـ *until*.

مثال:

```
var
  Num: Integer;
begin
  Write('Input a number: ');
  Readln(Num);
  while Num > 0 do
```

```
begin
  Write('From inside loop: Input a number : ');
  Readln(Num);
end;
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;

end.
```

يمكننا إعادة كتابة برنامج المضروب باستخدام `while do`

برنامج المضروب باستخدام حلقة `while do`

```
var
  Fac, Num, i: Integer;
begin
  Write('Please input any number: ');
  Readln(Num);
  Fac:= 1;
  i:= Num;
  while i > 1 do
  begin
    Fac:= Fac * i;
    i:= i - 1;
  end;
  Writeln('Factorial of ', Num , ' is ', Fac);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

بما أن حلقة `while` ليس لديها متغير حلقة مثل `for loop` فقد استخدمنا المتغير `i` ليكون متغير الحلقة. لذلك أنقصنا قيمة المتغير `i` حتى يصل إلى واحد وهو شرط عدم الدخول مرة أخرى في الحلقة.

المقاطع strings

المقاطع هي عبارة عن نص (text) أو مجموعة من الحروف والأرقام والرموز، حيث أنه يمكن للمتغير المقطعي أن يستقبل اسم مستخدم مثلاً أو رقم جواز أو رقم لوحة سيارة تحتوي على حروف وأرقام.

المثال التالي يوضح كيفية استقبال وطباعة قيمة متغير مقطعي:

```
var
  Name: string;
begin
  Write('Please enter your name : ');
  Readln(Name);
  Writeln('Hello ', Name);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

في المثال التالي نستخدم عدة متغيرات لتخزين معلومات شخص معين:

```
var
  Name: string;
  Address: string;
  ID: string;
  DOB: string;
begin
  Write('Please enter your name : ');
  Readln(Name);
  Write('Please enter your address : ');
  Readln(Address);
  Write('Please enter your ID number : ');
  Readln(ID);
  Write('Please enter your date of birth : ');
  Readln(DOB);
  Writeln;
  Writeln('Card:');
  Writeln('-----');
  Writeln(' | Name      : ', Name);
  Writeln(' | Address  : ', Address);
  Writeln(' | ID       : ', ID);
  Writeln(' | DOB     : ', DOB);
  Writeln('-----');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

يمكن تجميع المقاطع لتكوين مقطع أكبر، مثلاً إذا افترضنا أن لدينا ثلاث متغيرات مقطعية، أحدها يحمل إسم الشخص والثاني يحمل اسم والده والثالث اسم الجد، فيمكن تجميعها لتصبح مقطعاً واحداً:

```
var
  YourName: string;
  Father: string;
  GrandFather: string;
  FullName: string;
begin
  Write('Please enter your first name : ');
  Readln(YourName);
  Write('Please enter your father name : ');
  Readln(Father);
  Write('Please enter your grand father name : ');
  Readln(GrandFather);
  FullName:= YourName + ' ' + Father + ' ' + GrandFather;
  Writeln('Your full name is: ', FullName);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ في المثال السابق أننا أضفنا مسافة ' ' بين كل اسم والآخر حتى لا تلتصق الأسماء وتكون غير مقروءة. وهذه المسافة أيضاً عبارة عن رمز يمثل وحدة من وحدات المقطع.

يمكن إجراء عدة عمليات على المقاطع، مثل البحث فيها عن مقطع معين أو نسخها لمتغير مقطعي آخر، أو تحويلها إلى حروف إنجليزية كبيرة أو صغيرة. وكمثال يمكننا تجربة الدوال الخاصة بالحروف الكبيرة والصغيرة في اللغة الإنجليزية:

يمكننا إضافة سطر للمثال السابق قبل طباعة الاسم كاملاً. وإضافة هي:

```
FullName:= UpperCase(FullName);
```

أو التحويل للحروف الصغيرة:

```
FullName:= LowerCase(FullName);
```

في المثال التالي نريد البحث عن الحرف *a* في اسم المستخدم، وذلك باستخدام الدالة *Pos* والتي تُرجع رقم الحرف في المقطع، مثلاً الحرف رقم 1 أو 2، وهكذا، وإذا لم يكن موجود فإن هذه الدالة ترجع 0 وهو يعني عدم وجود هذا الحرف في المقطع المعني:

```
var
  YourName: string;
begin
  Write('Please enter your name : ');
  Readln(YourName);
  If Pos('a', YourName) > 0 then
    Writeln('Your name contains a')
  else
    Writeln('Your name does not contain a letter');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ بعد تشغيل البرنامج السابق أن الاسم إذا احتوى على A كبيرة فإنها لا تساوي الـ a الصغيرة ويعتبر البرنامج أنها غير موجودة، ولجعل البرنامج يتجاهل حالة الحرف يمكننا إضافة الدالة *LowerCase* أثناء البحث:

```
If Pos('a', LowerCase(YourName)) > 0 then
```

كذلك يمكن معرفة موضع الحرف في الاسم وذلك بتعديل البرنامج إلى:

```
var
  YourName: string;
begin
  Write('Please enter your name : ');
  Readln(YourName);
  If Pos('a', LowerCase(YourName)) > 0 then
    begin
      Writeln('Your name contains a');
      Writeln('a position in your name is: ',
        Pos('a', LowerCase(YourName)));
    end
  else
    Writeln('Your name does not contain a letter');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ أن الاسم إذا احتوى على أكثر من حرف a فإن الدالة *Pos* تُرجع رقم أول ظهور لهذا الحرف فقط.

يمكن معرفة طول المقطع (عدد حروفه) باستخدام الدالة *length* كما في المثال التالي:

```
Writeln('Your name length is ', Length(YourName), ' letters');
```

كذلك يمكن معرفة الحرف الأول باستخدام:

```
Writeln('Your first letter is ', YourName[1]);
```

والحرف الثاني:

```
Writeln('Your second letter is ', YourName[2]);
```

والحرف الأخير:

```
Writeln('Your last letter is ', YourName[Length(YourName)]);
```

ويمكن كتابة كافة الحروف مفردة ، كل حرف في سطر كالاتي:

```
for i:= 1 to Length(YourName) do  
  Writeln(YourName[i]);
```

الدالة Copy

يمكن نسخ جزء معين من المقطع مثلاً إذا كان المقطع يحتوى على الكلمة 'hello world' فيمكننا استخراج كلمة 'world' إذا عرفنا موضعها بالضبط في المقطع الأصلي وذلك كما في المثال التالي:

```
var  
  Line: string;  
  Part: string;  
begin  
  Line:= 'Hello world';  
  
  Part:= Copy(Line, 7, 5);  
  
  Writeln(Part);  
  //For Windows, please uncomment below lines  
  //writeln('Press enter key to continue..');  
  //readln;  
end.
```

نلاحظ أننا استخدمنا الدالة Copy بالصيغة التالية

```
Part:= Copy(Line, 7, 5);
```

ولشرحها من الشمال إلى اليمين:

Part =: وهي تعني أن نضع المخرجات النهائية للدالة *Copy* في المتغير المقطعي *part*

Line وهو المقطع المصدر الذي نريد النسخ منه

7 وهو رقم الحرف الذي نريد النسخ ابتداءً منه، وهو في هذا المثال يمثل الحرف *w*

5 وهو طول المقطع الذي نريد نسخه، وهي في هذه الحالة يمثل طول المقطع الجزئي *world*

في المثال التالي نطلب من المستخدم إدخال اسم شهر مثل February فيطبع البرنامج اختصار الشهر في هذه الصيغة : Feb وهي تمثل الحروف الثلاث الأولى من اسم الشهر:

```
var
  Month: string;
  ShortName: string;
begin
  Write('Please input full month name e.g. January : ');
  Readln(Month);

  ShortName:= Copy(Month, 1, 3);

  Writeln(Month, ' is abbreviated as : ', ShortName);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

الإجراء Insert

الدالة *insert* تضيف مقطع داخل مقطع آخر، بخلاف استخدام علامة الجمع + التي تلتصق مقطعين مع بعضهما، فإن الدالة *insert* يمكنها إدخال مقطع أو حرف داخل أو وسط مقطع آخر.

في المثال التالي تُدخل كلمة *Pascal* داخل عبارة *hello world* لتصبح *Hello Pascal World*

```
var
  Line: string;
begin
```



```
Line:= 'Hello world';  
  
Insert('Pascal ', Line, 7);  
  
Writeln(Line);  
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;  
end.
```

ومدخلات الإجراء هي كالتالي:

'Pascal' وهو المقطع الذي نريد إدخاله على العبارة
Line وهو المقطع الأصلي الذي نريد تغييره وإدخال مقطع آخر في وسطه
7 المكان بالأحرف الذي نريد الإضافة فيه. وهذا المكان أو الفهرس هو موضع في المقطع *line*

الإجراء Delete

يستخدم الإجراء *delete* لحذف حرف أو مقطع من مقطع آخر، مثلاً في المثال التالي نحذف حرفي الـ **L** في عبارة **Hello World** لتصبح **heo World**. وللقيام بهذه العملية يجب معرفة موقع الحرف الذي نريد الحذف من عنده وطول الجزء المحذوف. وهي في هذه الحالة 3 و 2 على التوالي.

```
var  
  Line: string;  
begin  
  Line:= 'Hello world';  
  Delete(Line, 3, 2);  
  Writeln(Line);  
  //For Windows, please uncomment below lines  
  //writeln('Press enter key to continue..');  
  //readln;  
end.
```

الدالة Trim

الدالة Trim تحذف المسافات فقط في بداية ونهاية مقطع معين. مثلاً إذا كان لدينا هذا المقطع ' hello ' فإنه يصبح 'hello' بعد استخدام هذه الدالة. لكن لا يمكن إظهار المسافة في أمثلتنا الحالية إلا إذا كتبنا حرف أو رمز قبل وبعد الكلمة المطبوعة:

```
program MultTableWithForLoop;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

{$IFDEF WINDOWS}{$R first.rc}{$ENDIF}

var
  Line: string;
begin
  Line:= ' Hello ';

  Writeln('<', Line, '>');

  Line:= Trim(Line);

  Writeln('<', Line, '>');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ أننا أضفنا الوحدة أو المكتبة *SysUtils* وهي التي توجد فيها هذه الدوال.

توجد دوال أخرى هي *TrimLeft* و *TrimRight* وهي تحذف المسافات من جهة واحدة يمكن تجربتها في المثال السابق.

الدالة StringReplace

الدالة *StringReplace* تُبدل مقاطع أو حروف معينة من مقطع ما، ثم وضع النتيجة في مقطع جديد، كما في المثال التالي:

```
program $trReplace;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  Line: string;
  Line2: string;
begin
  Line:= 'This is a test for string replacement';
  Line2:= StringReplace(Line, ' ', '-', [rfReplaceAll]);
  Writeln(Line);
  Writeln(Line2);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

مدخلات الدالة هي كالتالي:

1. *Line*: وهو المقطع المصدر الذي نريد إصدار نسخة معدلة منه
2. ' ': وهو المقطع الذي نريد البحث عنه وإبداله، وفي هذه الحالة يمثل حرف واحد وهو المسافة
3. '-': المقطع البديل، وهو المقطع الذي سوف يُبدل
4. *[rfReplaceAll]*: وهي طريقة الإبدال، في هذه الحالة سوف تُبدل كل المسافات.

يمكن استخدام مقطع واحد *Line* و الاستغناء عن المتغير *Line2* مع الحصول على نفس النتيجة كالتالي:

```
var
  Line: string;
begin
  Line:= 'This is a test for string replacement';
  Writeln(Line);
  Line:= StringReplace(Line, ' ', '-', [rfReplaceAll]);
  Writeln(Line);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

المصفوفات arrays

المصفوفة هي عبارة عن صف أو مجموعة من متغيرات ذات نوع واحد، مثلاً إذا قلنا أن لدينا مصفوفة من الأعداد الصحيحة تحتوي على 10 عناصر فإن تعريفها يكون كالتالي

```
Numbers: array [1 .. 10] of Integer;
```

ويمكننا وضع قيمة في المتغير الأول في المصفوفة كالتالي:

```
Numbers[1]:= 30;
```

ويمكننا وضع قيمة في المتغير الثاني في المصفوفة كالتالي:

```
Numbers[2]:= 315;
```

في المثال التالي نخزن درجات عشرة طلاب ثم نعرضها:

```
var
  Marks: array [1 .. 10] of Integer;
  i: Integer;
begin
  for i:= 1 to 10 do
  begin
    Write('Input student number ', i, ' mark: ');
    Readln(Marks[i]);
  end;

  for i:= 1 to 10 do
  begin
    Write('Student number ', i, ' mark is : ', Marks[i]);
    if Marks[i] >= 40 then
      Writeln(' Pass')
    else
      Writeln(' Fail');
    end;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نجد أننا استخدمنا حلقة *for* لإدخال وطباعة الدرجات. كذلك قارنا نتيجة كل طالب داخل الحلقة لنعرف الناجح وغير الناجح.

يمكن تعديل البرنامج السابق لنعرف أكبر وأصغر درجة في الدرجات كالتالي:

```
var
Marks: array [1 .. 10] of Integer;
i: Integer;
Max, Min: Integer;
begin

for i:= 1 to 10 do
begin
Write('Input student number ', i, ' mark: ');
Readln(Marks[i]);
end;

Max:= Marks[1];
Min:= Marks[1];

for i:= 1 to 10 do
begin
// Get if current Mark is maximum mark or not
if Marks[i] > Max then
Max:= Marks[i];

// Check if current value is minimum mark or not
if Marks[i] < Min then
Min:= Marks[i];

Write('Student number ', i, ' mark is : ', Marks[i]);
if Marks[i] >= 40 then
Writeln(' Pass')
else
Writeln(' Fail');
end;

Writeln('Max mark is: ', Max);
Writeln('Min mark is: ', Min);
end.
```

نلاحظ أننا افترضنا أن الرقم الأول هو أكبر درجة لذلك وضعناه في المتغير *Max* كذلك اعتبرنا أنه أصغر درجة فوضعناه في المتغير *Min* إلى أن يثبت العكس في كلا الحالتين.

```
Max:= Marks[1];
Min:= Marks[1];
```

وفي داخل الحلقة عند طباعة الأرقام العشرة، قارنا كل رقم مع القيم *max* و *min* فإذا وجدنا رقم أكبر من *max* استبدلنا قيمة *max* بقيمة الدرجة الحالية، وفعّلنا نفس الشيء مع القيمة *min*.

نلاحظ في المثال السابق أننا استخدمنا تعليق مثل:

```
// Get if current Mark is maximum mark or not
```

وقد بدأنا السطر بالعلامة // وهي تعني أن باقي السطر عبارة عن تعليق لا يُترجم ضمن البرنامج، إنما يستفيد منه المبرمج كشرح حتى يصبح البرنامج مقروء له وللمن يريد الإطلاع على البرنامج. هذه الطريقة تصلح للتعليق القصير، أما إذا كان التعليق أكثر من سطر يمكن استخدام الأقواس المعكوفة {} أو الأقواس والنجمة (**)

مثلاً:

```
for i:= 1 to 10 do
begin
  { Get if current Mark is maximum mark or not
  check if Mark is greater than Max then put
  it in Max }
  if Marks[i] > Max then
    Max:= Marks[i];

  (* Check if current value is minimum mark or not
  if Min is less than Mark then put Mark value in Min
  *)
  if Marks[i] < Min then
    Min:= Marks[i];

  Write('Student number ', i, ' mark is : ', Marks[i]);
  if Marks[i] >= 40 then
    Writeln(' Pass')
  else
    Writeln(' Fail');
end;
```

كذلك يمكن الاستفادة من هذه الخاصية بتعطيل جزء من الكود مؤقتاً كالتالي:

```
Writeln('Max mark is: ', Max);
// Writeln('Min mark is: ', Min);
```

في هذا المثال عطلنا إجراء طباعة أصغر درجة.

السجلات Records

كما لاحظنا أن المصفوفات تحتوي على مجموعة متغيرات من نوع واحد، فإن السجلات تجمع بين مجموعة من أنواع مختلفة تسمى حقول *Fields*، ولكنها تمثل كيان واحد. مثلاً إذا افترضنا أننا نريد تسجيل معلومات سيارة، فنجد أن هذه المعلومات هي:

1. نوع السيارة: متغير مقطعي
2. سعة المحرك: حدد حقيقي (كسري)
3. سنة التصنيع: متغير صحيح

فلا يمكن التعبير عن هذه الأنواع المختلفة كوحدة واحدة إلا باستخدام السجل كما في المثال التالي:

```
program Cars;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TCar = record
    ModelName: string;
    Engine: Single;
    ModelYear: Integer;
  end;

var
  Car: TCar;

begin
  Write('Input car Model Name: ');
  Readln(Car.ModelName);
  Write('Input car Engine size: ');
  Readln(Car.Engine);
  Write('Input car Model year: ');
  Readln(Car.ModelYear);

  Writeln;
  Writeln('Car information: ');
  Writeln('Model Name : ', Car.ModelName);
  Writeln('Engine size : ', Car.Engine);
  Writeln('Model Year : ', Car.ModelYear);
```



```
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;
```

end.

في المثال السابق عرّفنا نوع جديد باستخدام الكلمة المفتاحية *type* :

```
type  
  TCar = record  
    ModelName: string;  
    Engine: Single;  
    ModelYear: Integer;  
end;
```

وسمينا هذا النوع الجديد *TCar* والحرف *T* هو من كلمة *Type* حتى نفرق بينه وبين المتغيرات. وهذا النوع الجديد الذي يمثل سجل *Record* يحتوي على ثلاث أنواع كما يظهر في المثال.

وعندما نريد استخدام هذا النوع الجديد لابد من تعريف متغير يمثل هذا النوع، حيث لا يمكننا استخدام النوع *TCar* مباشرةً كما لا يمكننا استخدام النوع *Integer* مباشرةً إلا بعد تعريف متغير مثلاً / أو *Num*. لذلك عرّفنا المتغير *Car* من النوع *TCar* في بند المتغيرات:

```
var  
  Car: TCar;
```

وعندما نريد إدخال قيم للمتغيرات أو طباعتها نستخدم هذه الطريقة للوصول لمتغير ما في سجل:

```
Car.ModelName
```

في درس الملفات ذات الوصول العشوائي إن شاء الله سوف نستفيد فائدة مباشرة من السجلات التي تمثل ركن أساسي في قواعد البيانات.

الملفات files

الملفات هي من أهم العناصر في نظام التشغيل والبرامج عموماً، ونظام التشغيل نفسه هو عبارة عن ملفات بأشكال مختلفة. والمعلومات والبيانات هي عبارة عن ملفات، مثل الصور والكتب والبرامج والنصوص البسيطة، كلها عبارة عن ملفات. ويجب على نظام التشغيل توفير إمكانية لإنشاء الملفات ولقراءتها وكتابتها ولتحريرها ولحذفها.

تنقسم الملفات إلى عدة أنواع بناءً على عدة أوجه نظر. فيمكن تقسيم الملفات إلى ملفات تنفيذية وملفات بيانات، حيث أن الملفات التنفيذية هي التي تمثل البرامج وأجزائه التنفيذية مثل الملف الثنائي الذي يحتوي على كود يفهمه نظام التشغيل و مثال لها الملفات التنفيذية التي تصدر عن مترجم الفري باسكال أو مترجمات لغة سي، مثل ال *gcc*. هذه الملفات التنفيذية يمكن نقلها في عدد من الأجهزة التي تحتوي على نفس نظام التشغيل ثم تشغيلها بالنقر عليها أو بكتابة اسمها في شاشة الطرفية *console*. وكمثال لها برنامج الآلة الحاسبة، محرر النصوص، برامج الألعاب، إلخ.

أما النوع الثاني فهي ملفات البيانات التي لا تحتوي على كود ولا تمتلك إمكانية التشغيل، إنما تحتوي على بيانات بسيطة أو معقدة يمكن قراءتها مثل النصوص والملفات المصدرية للغات البرمجة مثل *first.lpr*، أو الصور وملفات الصوت التي هي عبارة عن بيانات يمكن عرضها باستخدام برامج معينة مثل برامج تحرير الصور وبرامج تعدد الوسائط *multimedia*. وكمثال لهذه الملفات ملفات الصوت .mp3 وملفات الكتب .pdf.

يمكن ذكر نوع ثالث وهي ملفات الأوامر *scripts* وهي ملفات نصية من حيث أنها تحتوي على نص يمكن قراءته، وهي تحتوي على أوامر لنظام تشغيل مثل برامج ال *shell scripts* في نظام لينكس، أو *batch command* في نظام وندوز، أو برامج لغة *php* أو *Python* يمكن اعتبارها تنفيذية لأنها تحتوي على أوامر ويمكن تشغيلها لتنفيذ برنامج معين. لكنها في الحقيقة ملفات نصية وملفات بيانات تستخدمها مفسرات لغات البرمجة تلك لتعمل بطريقة معينة، أي أن برنامج بايثون النصي عند تشغيله إنما يُنفذه حقيقة مفسر أو مترجم البايثون والذي هو ملف تنفيذي يمكن أن يكون قد تمت كتابته بلغة سي مثلاً. أي أن مترجم بايثون هو ملف تنفيذي ثنائي، أما البرنامج المحتوي على أوامر لغة بايثون فهي ملفات نصية.

يمكننا كذلك تقسيم الملفات من حيث نوع البيانات إلى قسمين:

1. **ملفات نصية** بسيطة يمكن إنشائها وقراءتها عن طريق أدوات بسيطة في نظام التشغيل مثل *cat* التي تعرض محتويات ملف نصي في نظام لينكس، أو *type* التي تطبع محتويات ملف نصي في وندوز، و *nano* التي تحرر و تُنشئ الملفات النصية في معظم أنظمة لينكس مثل دبيان لينكس.

2. **ملفات بيانات ثنائية** وهي يمكن أن تحتوي على رموز غير مقروءة وهذه الملفات لا تصلح لأن تُحرر و تُقرأ بواسطة المستخدم مباشرة، بل يجب عليه استخدام برامج محددة لهذه العمليات. مثل ملفات الصور إذا حاول المستخدم قراءتها باستخدام *cat* مثلاً فإن يحصل على رموز لا تُفهم، لذلك يجب فتحها باستخدام برامج عرض أو تحرير مثل المتصفح أو برنامج *Gimp*. كذلك يمكن أن يكون مثال لهذه الأنواع ملفات قواعد البيانات البسيطة مثل *paradox, dBase* فهي عبارة عن ملفات ثنائية تُخزن فيها سجلات تحتوي على معلومات منظمة بطريقة معينة.

النوع الآخر من التقسيم للملفات هو **طريقة الوصول** والكتابة، حيث يوجد نوعين من الملفات:

1. **ملفات ذات وصول تسلسلي sequential access files**: ومثال لها الملفات النصية، وتتميز بأن طول السجل أو السطر فيها غير ثابت، لذلك لا يمكن معرفة موقع سطر معين في الملف. ويجب فتح الملف للقراءة فقط أو الكتابة فقط، ولا يمكن الجمع بين الـ (الكتابة والقراءة) ولا يمكن تعديلها بسهولة إلا بقراءتها كاملة في مخزن مؤقت ثم تعديل أسطر معينة فيها ثم مسح الملف وكتابتها من جديد. كذلك فإن القراءة والكتابة تحدث بتسلسل وهو من بداية الملف إلى نهايته، حيث لا يمكن الرجوع سطر إلى الورا بالطرق العادية.

2. **ملفات ذات وصول عشوائي random access files**: وهي ملفات ذات طول سجل ثابت، حين أن السجل يمثل أصغر وحدة يتعامل معها البرنامج في القراءة، الكتابة والتعديل. ويمكن الجمع بين الكتابة والقراءة في نفس اللحظة، مثلاً يمكن قراءة السجل الخامس، ثم نسخ محتوياته في السجل الأخير. ويمكن الوصول مباشرة إلى أي سجل دون التقيد بمكان القراءة أو الكتابة الحالي. فمثلاً إلى كان طول السجل هو 5 حروف أو بايت، فإن السجل الخامس يبدأ في الموقع رقم 50 في هذا الملف.

الملفات النصية text files

الملفات النصية كما سبق ذكرها هي ملفات بسيطة يمكن قراءتها بسهولة. ومن ناحية الوصول هي ملفات تسلسلية **sequential files** حيث يجب القراءة فقط في اتجاه واحد وهو من البداية للنهاية ويجب الكتابة فيها بنفس الطريقة.

لعمل برنامج بسيط لقراءة محتويات ملف، علينا أولاً إنشاء ملفات نصية أو البحث عنها ثم وضعها في الدليل أو المجلد الذي يوجد فيه البرنامج ثم تنفيذ البرنامج التالي:

برنامج قراءة ملف نصي

```
program ReadFile;

{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, sysUtils
  { you can add units after this };

var
  FileName: string;
  F: TextFile;
  Line: string;
begin
  Write('Input a text file name: ');
  Readln(FileName);
  if FileExists(FileName) then
    begin
      // Link file variable (F) with physical file (FileName)
      AssignFile(F, FileName);

      Reset(F); // Open file, set open mode for read, file should be exist

      // while file has more lines that does not read yet do the loop
      while not Eof(F) do
        begin
          Readln(F, Line); // Read a line from text file
          Writeln(Line); // Display this line in user screen
        end;
      CloseFile(F); // Release F and FileName connection
    end
  else // else if FileExists..
    Writeln('File does not exist');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
```

```
//readln;  
end.
```

بعد تنفيذ البرنامج السابق تُدخل اسم ملف موجود في نفس مكان البرنامج أو بكتابة الاسم كاملاً مثلاً
يمكن إدخال أسماء هذه الملفات في نظام لينكس:

```
/etc/resolv.conf
```

أو

```
/proc/cpuinfo
```

أو هذا الملف في نظام وندوز:

```
c:\windows\win.ini
```

نلاحظ أننا استخدمنا دوال وإجراءات وأنواع جديدة في هذه البرنامج وهي:

.1

```
F: TextFile;
```

وهي تعريف المتغير *F* بأن نوعه *TextFile* وهو النوع الذي يتعامل مع الملفات النصية أو ما يُعرف بالـ *file handle*، وفي البرنامج سوف نستخدم هذا المتغير للتعبير عن الملف بدلاً عن اسم الملف في القرص.

.2

```
if FileExists(FileName) then
```

وهي دالة موجودة في المكتبة *SysUtils* وهي تختبر وجود هذا الملف في القرص، فإذا كان موجود تحقق الشرط وإن لم يكن موجود فإن عبارة *else* هي التي سوف تُنفَّذ.

.3

```
AssignFile(F, FileName);
```

بعد التأكد من أن الملف موجود يربط إجراء *AssignFile* اسم الملف الفعلي بالمتغير *F*، والذي عن طريقه يمكننا التعامل مع الملف من داخل باسكال.

.4

```
Reset(F); // Open file, Set open mode for read, file should be exist
```

وهي العبارة التي تفتح الملف النصي للقراءة فقط، وهي تخبر نظام التشغيل أن هذا الملف محجوز

للقراءة، فإذا حاول برنامج آخر فتح الملف للكتابة أو حذفه فإن نظام التشغيل يمنعه برسالة مفادها أن الملف مفتوح بواسطة تطبيق آخر أو أن السماحية غير متوفرة `access denied`

.5

```
Readln(F, Line); // Read a line from text file
```

وهو إجراء القراءة من الملف، حيث أن هذا الإجراء يقرأ سطر واحد فقط من الملف المفتوح `F` ثم يضع هذا السطر في المتغير `Line`.

.6

```
while not Eof(F) do
```

كما ذكرنا في الإجراء `Readln` فإنه يقرأ سطر واحد فقط، وبالنسبة للملف النصي فإننا لا يمكننا معرفة كم سطر يحتوي هذا الملف قبل الانتهاء من قراءته كاملاً، لذلك استخدمنا هذه الدالة `eof` والتي تعني وصول مؤشر القراءة إلى نهاية الملف `End Of File`. وقد استخدمناها مع الحلقة `while` وذلك للاستمرار في القراءة حتى الوصول إلى نهاية الملف وتحقيق شرط `end of file`.

.7

```
CloseFile(F); // Release F and FileName connection
```

بعد الفراغ من قراءة الملف، يجب إغلاقه وذلك لتحريره من جهة نظام التشغيل حتى يمكن لبرامج أخرى التعامل معه بحرية، ولا بد من تنفيذ هذا الإجراء فقط بعد فتح الملف بنجاح بواسطة `reset` مثلاً. فإذا فشل فتح الملف أصلاً بواسطة `reset` عندها لا يجب إغلاقه.

في المثال التالي سوف نُنشئ ملف نصي جديد والكتابة فيه:

برنامج إنشاء وكتابة ملف نصي

```
var
  FileName: string;
  F: TextFile;
  Line: string;
  ReadyToCreate: Boolean;
  Ans: Char;
  i: Integer;
begin
  Write('Input a new file name: ');
  Readln(FileName);

  // Check if file exists, warn user if it is already exist
  if FileExists(FileName) then
    begin
      Write('File already exist, did you want to overwrite it? (y/n)');
      Readln(Ans);
      if upcase(Ans) = 'Y' then
        ReadyToCreate:= True
      else
        ReadyToCreate:= False;
    end
  else // File does not exist
    ReadyToCreate:= True;

  if ReadyToCreate then
    begin
      // Link file variable (F) with physical file (FileName)
      AssignFile(F, FileName);

      Rewrite(F); // Create new file for writing

      Writeln('Please input file contents line by line, '
        , 'when you finish write % then press enter');
      i:= 1;
      repeat
        Write('Line # ', i, ':');
        Inc(i);
        Readln(Line);
        if Line <> '%' then
          Writeln(F, Line); // Write line into text file
      until Line = '%';

      CloseFile(F); // Release F and FileName connection, flush buffer
    end
  else // file already exist and user does not want to overwrite it
    Writeln('Doing nothing');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
```

```
//readln;  
end.
```

في البرنامج السابق استخدمنا عدة أشياء وهي:

.1

```
ReadyToCreate: Boolean;
```

النوع *boolean* يمكن لمتغيراته أن تحمل إحدى قيمتين فقط: True/False . وهذه القيم يمكن استخدامها مع عبارة *if condition* كما في المثال، كذلك يمكن استخدامها مع حلقة *while* وحلقة *repeat*. حيث أن الشرط في النهاية يتحول إلى إحدى هتتين القيمتين، فكما في أمثلة سابقة استخدمنا هذه العبارة الشرطية:

```
if Marks[i] > Max then
```

فهي إما أن تكون النتيجة فيتحول الشرط إلى *True* أو تكون خاطئة فيتحول الشرط إلى *False*. فعندما تتحول إلى *True* يُنفذ الشرط:

```
if True then
```

وإذا كانت قيمتها النهائية *False* لا يُنفذ إجراء الشرط وإنما يُنفذ إجراء *else* إن وجد.

.2

```
if upcase(Ans) = 'Y' then
```

هذه العبارة تُنفذ في حالة وجود الملف، لذلك فإن البرنامج يُنبه المستخدم بوجود الملف، وسؤاله إذا كان يرغب في حذف محتوياته والكتابة عليه من جديد (overwrite). فإذا أدخل الحرف *y* صغيرة أو *Y* كبيرة فإن الشرط يُنفذ في الحالتين، حيث أن الدالة *upCase* تحوّل الحرف إلى حرف كبير لمقارنته مع الحرف الكبير 'Y'. أما إذا أدخل المستخدم حرفاً كبيراً *Y* فإن الدالة *upCase* لا تغير به وترجع الحرف كما هو *Y*.

.3

```
Rewrite(F); // Create new file for writing
```


الإجراء Rewrite يُنشئ ملف جديد و يحذف محتوياته إذا كان موجود. كذلك فهو يفتح الملف للكتابة فقط في حالة الملف النصي.

.4

```
WriteLn(F, Line); // Write line into text file
```

الإجراء *WriteLn(F)* يكتب المقطع *Line* في الملف ثم يُضيف علامة نهاية السطر وهي *CR/LF*. وهي عبارة عن رموز الواحد منها يمثل بايت وقيمتها هي كالآتي :

CR: Carriage Return = 13
LF: Line Feed = 10

وهذه الرموز من الأحرف الغير مرئية، حيث لا تُكتب في الشاشة، إنما يظهر فقط مفعولها، وهي الانتقال إلى سطر جديد.

.5

```
Inc(i);
```

الإجراء *Inc* يُضيف واحد إلى قيمة المتغير الصحيح، في هذه الحالة *i* وهو يعادل هذه العبارة:

```
i := i + 1;
```

.6

```
CloseFile(F); // Release F and FileName connection, flush buffer
```

كما ذكرنا سابقاً فإن الإجراء *CloseFile* يُغلق الملف و يُوقف عملية الكتابة أو القراءة من الملف، كذلك فإنه يحرره من الحجز من نظام التشغيل ليُسمح للبرامج الأخرى باستخدامه. إلا أن له وظيفة إضافية في حالة الكتابة. فكما نعلم أن الكتابة على القرص الصلب هي عملية بطيئة نسبياً مقارنة بالكتابة في الذاكرة، مثل إعطاء قيم للمتغيرات، أو الكتابة في مصفوفة. لذلك من غير المنطقي كتابة كل حرف أو سطر على حده في القرص الصلب مباشرة أو أي وسيط تخزين آخر، لذلك يُخزن البرنامج عدد معين من الأسطر في الذاكرة بعملية تسمى الـ *Buffering*، فكلما استخدمنا عبارة *Write* أو *WriteLn* لكتابة سطر فإن البرنامج يكتبه تلقائياً في الذاكرة إلى أن يمتلئ هذا الوعاء (*Buffer*) في الذاكرة فيكتب البرنامج فعلياً على القرص الصلب ثم يحذف المحتويات من الذاكرة (*Buffer*)، وهذه العملية تسمى *Flushing* وبهذه الطريقة نضمن السرعة في كتابة الملف، باعتبار أن التكلفة الزمنية مثلاً لكتابة سطر واحد ربما تساوي تقريباً تكلفة كتابة 10 أسطر في القرص الصلب دفعة واحدة. وتكمن خطورة هذه الطريقة في

انقطاع الطاقة عن الحاسوب قبل الكتابة الفعلية، فيجد المستخدم أن الأسطر الأخيرة التي كتبها قد ضاعت. ويمكن إجبار البرنامج بالقيام بالكتابة الفعلية على القرص باستخدام الإجراء *Flush*. كذلك فإن عملية الـ *Flushing* تحدث أيضاً عند إغلاق الملف.

الإضافة إلى ملف نصي

في هذا المثال سوف نفتح ملف نصي يحتوي على بيانات ثم نُضيف إليه أسطر بدون حذف الأسطر القديمة، وذلك باستخدام الإجراء *AppendFile*

برنامج الإضافة إلى ملف نصي:

```
var
  FileName: string;
  F: TextFile;
  Line: string;
  i: Integer;
begin
  Write('Input an existed file name: ');
  Readln(FileName);
  if FileExists(FileName) then
    begin
      // Link file variable (F) with physical file (FileName)
      AssignFile(F, FileName);

      Append(F); // Open file for appending

      Writeln('Please input file contents line by line',
        'when you finish write % then press enter');
      i:= 1;
      repeat
        Write('Line # ', i, ' append :');
        Inc(i);
        Readln(Line);
        if Line <> '%' then
          Writeln(F, Line); // Write line into text file
        until Line = '%';
        CloseFile(F); // Release F and FileName connection, flush buffer
      end
    else
      Writeln('File does not exist');
```

```
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

بعد تنفيذ البرنامج يمكننا كتابة اسم ملف نصي موجود وإضافة بضعة أسطر عليه، بعد ذلك يمكننا أن نستعرض الملف عن طريق الأمر *cat* في لينكس، أو عن طريق تصفح الدليل الذي يوجد فيه البرنامج و الضغط عليه بالماوس لفتحه.

ملفات الوصول العشوائي Random access files

كما سبق ذكره فإن النوع الثاني من الملفات من حيث الوصول هي ملفات الوصول العشوائي أو كما تسمى أحياناً بملفات الوصول المباشر. وهي تتميز بطول معروف للسجل ويمكن الكتابة والقراءة من الملف في آن واحد، كذلك يمكن الوصول مباشرة إلى أي سجل بغض النظر عن موقع القراءة أو الكتابة الحالي.

توجد طريقتين للتعامل مع الملفات ذات الوصول العشوائي في لغة باسكال: الطريقة الأولى هي استخدام الملفات ذات النوع المحدد *typed files*، أما الطريقة الثانية فهي استخدام الملفات الغير محددة النوع *untyped files*.

الملفات ذات النوع typed file

في هذه الطريقة يكون الملف المراد قراءته أو كتابته مرتبط بنوع معين، والنوع المعين يمثله سجل، فمثلاً يمكن أن يكون ملف من النوع الصحيح *Byte*، في هذه الحال نقول أن السجل هو عبارة عن عدد صحيح *Byte* وفي هذه الحالة يكون طول السجل 1 بايت.

المثال التالي يوضح طريقة كتابة ملف لأعداد صحيحة:

برنامج تسجيل درجات الطلاب

```
var
  F: file of Byte;
```

```
Mark: Byte;
begin
AssignFile(F, 'marks.dat');
Rewrite(F); // Create file
WriteLn('Please input students marks, write 0 to exit');

repeat
Write('Input a mark: ');
ReadLn(Mark);
if Mark <> 0 then // Don't write 0 value
Write(F, Mark);
until Mark = 0;
CloseFile(F);
end.
```

نلاحظ في البرنامج أننا عرفنا نوع الملف بهذه الطريقة:

```
F: file of Byte;
```

وهي تعني أن الملف هو من نوع *Byte* أو أن سجلاته عبارة هي قيم للنوع *Byte* والذي يحتل في الذاكرة وفي القرص 1 بايت، ويمكنه تخزين القيم من 0 إلى 255.

كذلك أنشأنا الملف وجهازناه للكتابة باستخدام الأمر:

```
Rewrite(F); // Create file
```

وقد استخدمنا الإجراء *Write* للكتابة في الملف:

```
Write(F, Mark);
```

حيث أن *WriteLn* لا تصلح لهذا النوع من الملفات لأنها تضيف علامة نهاية السطر *CR/LF* كما سبق ذكره، أما *Write* فهي تكتب السجل كما هو بدون أي زيادات، لذلك نجد أننا في المثال السابق يمكننا معرفة حجم الملف، فإذا أدخلنا 3 درجات فإن عدد السجلات يكون 3 وبما أن طول السجل هو 1 بايت فإن حجم الملف يكون 3 بايت. يمكن تغيير النوع إلى *Integer* الذي يحتل 4 خانات، ورؤية حجم الملف الناتج.

في المثال التالي نقرأ محتويات الملف السابق، فلا ننسى إرجاع نوع الملف في البرنامج السابق إلى النوع *Byte*، لأن القراءة والكتابة لا بد أن تكون لملف من نفس النوع.

برنامج قراءة ملف الدرجات

```
program ReadMarks;
```

```
{ $mode objfpc } { $H+ }

uses
  { $IFDEF UNIX } { $IFDEF UseCThreads }
  cthreads,
  { $ENDIF } { $ENDIF }
  Classes, SysUtils
  { you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    Reset(F); // Open file
    Writeln('Please input students marks, write 0 to exit');

    while not Eof(F) do
    begin
      Read(F, Mark);
      Writeln('Mark: ', Mark);
    end;
    CloseFile(F);
  end
  else
    Writeln('File (marks.dat) not found');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ أن البرنامج السابق أظهر الدرجات الموجودة في الملف. وأظن أن البرنامج واضح ولا يحتاج لشرح.

في البرنامج التالي نفتح الملف السابق ونضيف إليه درجات جديدة بدون حذف الدرجات السابقة:

برنامج إضافة درجات الطلاب

```
program AppendMarks;

{ $mode objfpc } { $H+ }

uses
  { $IFDEF UNIX } { $IFDEF UseCThreads }
  cthreads,
```

```
{ $ENDIF } { $ENDIF }
Classes, SysUtils
{ you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    FileMode:= 2; // Open file for read/write
    Reset(F); // open file
    Seek(F, FileSize(F)); // Go to after last record
    Writeln('Please input students marks, write 0 to exit');

    repeat
      Write('Input a mark: ');
      Readln(Mark);
      if Mark <> 0 then // Don't write 0 value
        Write(F, Mark);
    until Mark = 0;
    CloseFile(F);
  end
  else
    Writeln('File marks.dat not found');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

بعد تنفيذ البرنامج السابق نُشغل برنامج قراءة ملف الدرجات وذلك لرؤية أن قيم السجلات السابقة والجديدة موجودة معاً في نفس الملف.

نلاحظ في هذا البرنامج أننا استخدمنا الإجراء *Reset* للكتابة بدلاً من *Rewrite*. وفي هذا النوع من الملفات يمكن استخدام كليهما للكتابة، والفرق الأساسي بينهما يكمن في أن *Rewrite* تُنشيء الملف إذا لم يكن موجوداً وتحذف محتوياته إذا كان موجوداً، أما *Reset* فهي تفترض وجود الملف، فإذا لم يكن موجوداً حدث خطأ. لكن عبارة *Reset* تفتح الملف حسب قيمة *FileMode* :
فإذا كانت قيمتها 0 فإن الملف يفتح للقراءة فقط، وإذا كانت قيمته 1 يفتح للكتابة فقط، وإذا كانت قيمته 2 -وهي القيمة الافتراضية- فإنه يفتح للقراءة والكتابة معاً:

```
FileMode:= 2; // Open file for read/write
Reset(F); // open file
```

كذلك فإن الدالة *Reset* تضع مؤشر القراءة والكتابة في أول سجل، لذلك إذا باشرنا الكتابة فإن البرنامج

السابق يكتب فوق محتويات السجلات السابقة، لذلك يجب تحريك هذا المؤشر للسجل الأخير، وذلك باستخدام الإجراء *Seek* الذي يُحرّك المؤشر، ومن هنا سُمّي هذا النوع من الملفات بالملفات العشوائية أو ملفات الوصول المباشر، حيث أن الإجراء *Seek* يسمح لنا بالتنقل لأي سجل مباشرة إذا علمنا رقمه، شريطة أن يكون هذا الرقم موجود، فسوف يحدث خطأ مثلاً إذا حاولنا توجيه المؤشر إلى السجل رقم 100 في حين أن الملف يحتوي على عدد سجلات أقل من 100. كذلك استخدمنا الدالة *FileSize* والتي تحسب عدد السجلات الحالية في الملف، والسجل يُمثل وحدة من نوع الملف، فإذا كان ملف من نوع *Integer* مثلاً فإن طول السجل يكون أربعة بايت، فإذا كان حجم الملف 400 بايت فحينها يحتوي الملف على 100 سجل.

```
Seek(F, FileSize(F)); // Go to after last record
```

نلاحظ أن المثال السابق يصلح فقط في حالة وجود ملف الدرجات، أما إذا لم يكن موجود يجب استخدام البرنامج الأول لكتابة الدرجات. يمكننا المزج بين الطريقتين، بحيث أن البرنامج يفضل وجود الملف، فإذا كان موجود يفتحه للإضافة عن طريق *Reset* وإذا لم يكن موجود يُنشئه باستخدام *Rewrite*:

برنامج إنشاء وإضافة درجات الطلاب

```
program ReadWriteMarks;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    FileMode:= 2; // Open file for read/write
    Reset(F); // open file
    Writeln('File already exist, opened for append');
    // Display file records
```

```
while not Eof(F) do
begin
  Read(F, Mark);
  Writeln('Mark: ', Mark);
end
end
else // File not found, create it
begin
  Rewrite(F);
  Writeln('File does not exist, created');
end;

Writeln('Please input students marks, write 0 to exit');
Writeln('File pointer position at record # ', FilePos(f));
repeat
  Write('Input a mark: ');
  Readln(Mark);
  if Mark <> 0 then // Don't write 0 value
    Write(F, Mark);
until Mark = 0;
CloseFile(F);
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

بعد تشغيل البرنامج نجد أن القيم السابقة تُعرض في البداية قبل الشروع في إضافة درجات جديدة. نلاحظ أننا في هذه الحالة لم نستخدم الإجراء *Seek* وذلك لأننا قرأنا كل محتويات الملف، ومن المعروف أن القراءة تُحرك مؤشر الملف إلى الأمام، لذلك بعد الفراغ من قراءة كافة سجلاته يكون المؤشر في الخانة الأخيرة في الملف، لذلك يمكن الإضافة مباشرة. استخدمنا الدالة *FilePos* التي تخبرنا بالموقع الحالي لمؤشر الملفات.

في المثال التالي سوف نستخدم سجل *Record* لتسجيل بيانات سيارة، نلاحظ أننا كتبنا وقرأنا السجل كوحدة واحدة:

برنامج سجل السيارات

```
program CarRecords;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
```



```
{ $ENDIF } { $ENDIF }
Classes, SysUtils
{ you can add units after this };

type
  TCar = record
    ModelName: string[20];
    Engine: Single;
    ModelYear: Integer;
  end;

var
  F: file of TCar;
  Car: TCar;
begin
  AssignFile(F, 'cars.dat');
  if FileExists('cars.dat') then
  begin
    FileMode:= 2; // Open file for read/write
    Reset(F); // open file
    Writeln('File already exist, opened for append');
    // Display file records
    while not Eof(F) do
    begin
      Read(F, Car);
      Writeln;
      Writeln('Car # ', FilePos(F), ' -----');
      Writeln('Model : ', Car.ModelName);
      Writeln('Year : ', Car.ModelYear);
      Writeln('Engine: ', Car.Engine);
    end
  end
  else // File not found, create it
  begin
    Rewrite(F);
    Writeln('File does not exist, created');
  end;

  Writeln('Please input car informaion, ',
    'write x in model name to exit');
  Writeln('File pointer position at record # ', FilePos(f));

  repeat
    Writeln('-----');
    Write('Input car Model Name : ');
    Readln(car.ModelName);
    if Car.ModelName <> 'x' then
    begin
      Write('Input car Model Year : ');
      Readln(car.ModelYear);
      Write('Input car Engine size: ');
      Readln(car.Engine);
      Write(F, Car);
    end;
  end;
```

```
until Car.ModelName = 'x';
CloseFile(F);
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

في البرنامج السابق استخدمنا سجل لمعلومات السيارة، والحقل الأول في السجل *ModelName* هو من النوع المقطعي، إلا أننا في هذه الحالة استخدمنا *short string* والذي يحتوي على طول محدد لا يتجاوز الـ 255 حرف.

```
ModelName: string[20];
```

وبهذه الطريقة يكون طول المقطع معروف ومحدد ويأخذ مساحة معروفة من القرص، أما طريقة استخدام النوع *string* مطلقاً والذي يسمى *AnsiString* فهي طريقة لها تبعاتها في طريقة تخزينها في الذاكرة، وسوف نتكلم عنها في كتاب لاحق إن شاء الله.

نسخ الملفات Files copy

الملفات بكافة أنواعها سواءً كانت ملفات نصية أو ثنائية، صور، برامج، أو غيرها فإنها تشترك في أن الوحدة الأساسية فيها هي البايت *Byte*، حيث أن أي ملف هو عبارة عن مجموعة من البايت، تختلف في محتوياتها، لكن البايت يحتوي على أرقام من القيمة صفر إلى القيمة 255، لذلك إن قرأنا أي ملف فنجد أن رموزه لا تخرج عن هذه الاحتمالات (0 - 255).

عملية نسخ الملف هي عملية بسيطة، فنحن ننسخ الملف حرفاً حرفاً باستخدام متغير يحتل بايت واحد من الذاكرة مثل البايت *Byte* أو الرمز *Char*. في هذه الحالة لا يهم نوع الملف، لأن النسخ بهذه الطريقة يكون الملف المنسوخ صورة طبق الأصل من الملف الأصلي:

برنامج نسخ الملفات عن طريق البايت

```
program FilesCopy;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
```

```
Classes, SysUtils
{ you can add units after this };

var
  SourceName, DestName: string;
  SourceF, DestF: file of Byte;
  Block: Byte;
begin
  Writeln('Files copy');
  Write('Input source file name: ');
  Readln(SourceName);

  Write('Input destination file name: ');
  Readln(DestName);
  if FileExists(SourceName) then
  begin
    AssignFile(SourceF, SourceName);
    AssignFile(DestF, DestName);

    FileMode:= 0; // open for read only
    Reset(SourceF); // open source file
    Rewrite(DestF); // Create destination file

    // Start copy
    Writeln('Copying..');
    while not Eof(SourceF) do
    begin
      Read(SourceF, Block); // Read Byte from source file
      Write(DestF, Block); // Write this byte into new
                          // destination file
    end;
    CloseFile(SourceF);
    CloseFile(DestF);

  end
  else // Source File not found
    Writeln('Source File does not exist');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

عند تشغيل هذا البرنامج يجب كتابة اسم الملف المراد النسخ منه والملف الجديد كاملاً مثلاً في نظام لينكس نكتب:

```
Input source file name: /home/motaz/quran/mishari/32.mp3
Input destination file name: /home/motaz/Alsajda.mp3
```

وفي نظام وندوز:

```
Input source file name: c:\photos\mypphoto.jpg  
Input destination file name: c:\temp\copy.jpg
```

أما إذا كان البرنامج *FileCopy* موجود في نفس الدليل للملف المصدر والنسخة، فيمكن كتابة اسمي الملف بدون كتابة اسم الدليل مثلاً:

```
Input source file name: test.pas  
Input destination file name: testcopy.pas
```

نلاحظ أن برنامج نسخ الملفات يأخذ وقت طويل عند نسخ الملفات الكبيرة مقارنة بنسخها بواسطة نظام التشغيل نفسه، وذلك يعني أن نظام التشغيل يستخدم طريقة مختلفة لنسخ الملفات. فهذه الطريقة بطيئة جداً بسبب قراءة حرف واحد في الدورة الواحدة ثم نسخة في الملف الجديد، فلو كان حجم الملف مليون بايت فإن الحلقة تدور مليون مرة، تتكرر فيها القراءة مليون مرة والكتابة مليون مرة. وكانت هذه الطريقة للشرح فقط، أما الطريقة المثلى لنسخ الملفات فهي عن طريق استخدام الملفات غير محددة النوع *untyped files*.

الملفات غير محددة النوع *untyped files*

وهي ملفات ذات وصول عشوائي، إلا أنها تختلف عن الملفات محددة النوع في أنها لا ترتبط بنوع محدد، كذلك فإن القراءة والكتابة غير مرتبطة بعدد سجلات معين، حيث أن للمبرمج كامل الحرية في تحديد عدد السجلات التي يرغب في كتابتها أو قراءتها في كل مرة.

برنامج نسخ الملفات باستخدام الملفات غير محددة النوع

```
program FilesCopy2;  
  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes, SysUtils  
  { you can add units after this };  
  
var  
  SourceName, DestName: string;
```

```
SourceF, DestF: file;
Block: array [0 .. 1023] of Byte;
NumRead: Integer;
begin
  Writeln('Files copy');
  Write('Input source file name: ');
  Readln(SourceName);

  Write('Input destination file name: ');
  Readln(DestName);

  if FileExists(SourceName) then
  begin
    AssignFile(SourceF, SourceName);
    AssignFile(DestF, DestName);

    FileMode:= 0; // open for read only
    Reset(SourceF, 1); // open source file
    Rewrite(DestF, 1); // Create destination file

    // Start copy
    Writeln('Copying..');
    while not Eof(SourceF) do
    begin
      // Read Byte from source file
      BlockRead(SourceF, Block, SizeOf(Block), NumRead);
      // Write this byte into new destination file
      BlockWrite(DestF, Block, NumRead);
    end;
    CloseFile(SourceF);
    CloseFile(DestF);

  end
  else // Source File not found
    Writeln('Source File does not exist');

  Writeln('Copy file is finished. ');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

في المثال السابق نجد أن هناك أشياء جديدة وهي:

1. طريقة تعريف الملفات، وهي بتعريفها أن المتغير مجرد ملف:

```
SourceF, DestF: file;
```

2. الوعاء الذي يستخدم في نسخ البيانات بين الملفين:

```
Block: array [0 .. 1023] of Byte;
```

نجد أنه في هذه الحالة عبارة عن مصفوفة من نوع البايت تحتوي على كيلو بايت، ويمكن تغييرها إلى أي رقم يريده المبرمج تتناسب مع الذاكرة وحجم الملف.

3. طريقة فتح الملف اختلفت قليلاً:

```
Reset(SourceF, 1); // open source file  
Rewrite(DestF, 1); // Create destination file
```

فقد زاد مُدخل جديد وهو طول السجل، وفي حالة نسخ ملفات هذه يجب أن يكون دائماً يحمل القيمة واحد وهو يعني أن طول السجل واحد بايت. والسبب يكمن في أن الرقم واحد يقبل القسمة على جميع قيم حجم الملفات، مثلاً يمكن أن يكون حجم الملف 125 بايت، أو 23490 بايت وهكذا. أما إذا حددنا طول السجل بـ 2 مثلاً فإنه يحدث خطأ عند قراءة آخر سجل في الملفات التي حجمها رقم فردي مثلاً ملف ذو 123 بايت.

4. طريقة القراءة:

```
BlockRead(SourceF, Block, SizeOf(Block), NumRead);
```

يستخدم الإجراء *BlockRead* مع الملفات غير محددة النوع، حيث يعتبر أن القيمة المراد قراءتها هي عبارة عن كومة أو رزمة غير معلومة المحتويات. والمدخلات لهذا الإجراء هي:

SourceF: وهو متغير الملف المراد القراءة منه.

Block: وهو المتغير أو المصفوفة التي يراد وضع محتويات القراءة الحالية فيها.

SizeOf(Block): وهو عدد السجلات المراد قراءتها في هذه اللحظة، ونلاحظ أننا استخدمنا الدالة *SizeOf* التي تحسب سعة أو حجم المتغير من حيث عدد البايت، وفي هذه الحالة هو الرقم 1024.

NumRead: عندما نقول أننا نريد قراءة 1024 بايت ربما ينجح الإجراء بقراءتها جميعاً في حالة أن تكون هناك بيانات متوفرة في الملف، أما إذا كانت محتويات الملف أقل من هذه القيمة أو أن مؤشر القراءة وصل قرب نهاية الملف، ففي هذه الحالة ربما لا يستطيع قراءة 1024 بايت، وتكون القيمة التي قرأها أقل من ذلك وتخزن القيمة في المتغير *NumRead*. فمثلاً إذا كان حجم الملف 1034 بايت، يقرأ الإجراء 1024 بايت في المرة الأولى، أما في المرة الثانية فيقرأ 10 بايت فقط وترجع هذه القيمة في

المتغير *NumRead* حتى يتسنى استخدامها مع الإجراء *BlockWrite*.

5. طريقة الكتابة:

```
BlockWrite(DestF, Block, NumRead);
```

وأظن أنها واضحة، والمتغير الأخير *NumRead* في هذه المرة عدد البايت أو الرموز المراد كتابتها في الملف المنسوخ، وهي تعني عدد الرموز من بداية المصفوفة *Block*. وعند تشغيل هذا الإجراء فإن المتغير *NumRead* يحمل قيمة عدد الرموز التي تمت قراءتها عن طريق *BlockRead* وعند تشغيل البرنامج سوف نلاحظ السرعة الكبيرة في نسخ الملفات، فمثلاً إذا كان طول الملف مليون بايت، فيلزم حوالي أقل من ألف دورة فقط للقراءة ثم الكتابة في الملف الجديد.

في المثال التالي، يعرض البرنامج محتويات الملف بالطريقة المستخدمة في التخزين في الذاكرة أو الملف، فكما سبق ذكره فإن الملف هو عبارة عن سلسلة من الحروف أو الرموز.

برنامج عرض محتويات رموز الملف

```
program ReadContents;
{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  FileName: string;
  F: file;
  Block: array [0 .. 1023] of Byte;
  i, NumRead: Integer;
```

```
begin
  Write('Input source file name: ');
  Readln(FileName);

  if FileExists(FileName) then
    begin
      AssignFile(F, FileName);

      FileMode:= 0; // open for read only
      Reset(F, 1);

      while not Eof(F) do
        begin
          BlockRead(F, Block, SizeOf(Block), NumRead);
          // display contents in screen
          for i:= 0 to NumRead - 1 do
            Writeln(Block[i], ':', Chr(Block[i]));
          end;
          CloseFile(F);
        end
      else // File does not exist
        Writeln('Source File does not exist');
        //For Windows, please uncomment below lines
        //writeln('Press enter key to continue..');
        //readln;
    end.
end.
```

بعد تنفيذ البرنامج يمكن للمستخدم كتابة اسم ملف نصي حتى نعرض محتوياته. نلاحظ أن بعد كل سطر نجد علامة الـ *Line Feed LF* وقيمتها كرقم هي 10 في نظام لينكس، أما في نظام وندوز فنجد علامة *Carriage Return/Line Feed CRLF* وقيمتها على التوالي 13 و 10، وهي الفاصل الموجود بين السطور في الملف النصي. يمكن كذلك استخدام البرنامج في فتح ملفات من نوع آخر لمعرفة طريقة تكوينها. استخدمنا في البرنامج الدالة *Chr* التي تحول البايت إلى حرف، مثلاً نجد أن البايت الذي قيمته 97 يمثل الحرف *a* وهكذا.

التاريخ والوقت Date and Time

التاريخ والوقت من الأساسيات المهمة في البرامج، فهي من الخصائص المهمة المرتبطة بالبيانات والمعلومات، فمعظم المعلومات والبيانات تحتاج لتاريخ يمثل وقت إدراج هذه المعلومة أو تعديلها. فمثلاً إذا كانت هناك معاملة بنكية أو صرف شيك فلا بد من تسجيل الوقت الذي حدثت فيه هذه المعاملة نسبة لأهميتها في المراجعة لاحقاً عندما يطلب العميل كشف الحساب مثلاً، أو عندما يحصر البنك المعاملات التي حدثت في شهر ما مثلاً.

يُخزن التاريخ والوقت في متغير واحد من نوع `TDateTime` وهو عبارة عن عدد حقيقي (كسري) ذو دقة مضاعفة `Double` وهو يحتل ثماني خانات من الذاكرة. العدد الصحيح منه يمثل الأيام، والجزء الكسري يمثل الوقت. والقيمة `صفر` لمتغير من هذا النوع تمثل الأيام التي انقضت منذ يوم 30/12/1899 ميلادية.

في البرنامج التالي نستخدم الدالة `Now` والتي تُرجع قيمة التاريخ والزمن الحاليين:

```
program DateTime;  
  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes , SysUtils  
  { you can add units after this };  
  
begin  
  Writeln('Current date and time: ', DateTimeToStr(Now));  
end.
```

نلاحظ أننا استخدمنا الدالة `DateToStr` الموجودة في المكتبة `SysUtils` والتي تُفسر قيمة التاريخ والوقت وتحولهما لمقطع مقروء حسب إعدادات المستخدم في طريقة إظهار التاريخ. فإذا لم نستخدمها سوف نحصل على عدد حقيقي لا يمكن قراءته بسهولة:

```
Writeln('Current date and time: ', Now);
```

كذلك توجد دوال لإظهار قيمة التاريخ فقط وأخرى لإظهار قيمة الوقت فقط مثلاً:

```
Writeln('Current date is ', DateToStr(Now));  
Writeln('Current time is ', TimeToStr(Now));
```

كذلك توجد دالتان مشابهتان للدالة **Now** لكنهما يرجعان التاريخ فقط والأخرى الزمن فقط وهما **Date** و **Time**

```
Writeln('Current date is ', DateToStr(Date));  
Writeln('Current time is ', TimeToStr(Time));
```

هذه الدوال تحذف الجزء الآخر أو تحوله إلى صفر، فالدالة **Date** تُرجع فقط التاريخ و تُرجع صفر في الجزء الكسري الذي يمثل الزمن. والقيمة صفر بالنسبة للزمن تمثل الساعة 12 صباحاً. والدالة **Time** تُرجع الزمن وتضعه القيمة صفر في الجزء الصحيح الذي يمثل التاريخ والذي قيمته كما ذكرنا **30/12/1899**. يمكن ملاحظة هذه القيم في المثال التالي عند استخدام الدالة **DateTimeToStr** مع الدوال الجديدة **Date**, **Time**

```
Writeln('Current date is ', DateTimeToStr(Date));  
Writeln('Current time is ', DateTimeToStr(Time));
```

يمكن أن يتضح المثال السابق أكثر باستخدام الدالة **FormatDateTime** والتي تعرض هيئة التاريخ والوقت بحسب الشكل الذي يريده المبرمج بغض النظر عن إعدادات المستخدم في حاسوبه:

```
Writeln('Current date is ',  
FormatDateTime('yyyy-mm-dd hh:nn:ss', Date));  
Writeln('Current time is ',  
FormatDateTime('yyyy-mm-dd hh:nn:ss', Time));
```

في المثال التالي سوف نعامل التاريخ كعدد حقيقي فنضيف أو نطرح قيم منه.

```
begin  
Writeln('Current date and time is ',  
FormatDateTime('yyyy-mm-dd hh:nn:ss', Now));  
Writeln('Yesterday time is ',  
FormatDateTime('yyyy-mm-dd hh:nn:ss', Now - 1));  
Writeln('Tomorrow time is ',  
FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1));  
Writeln('Today + 12 hours is ',  
FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1/2));  
Writeln('Today + 6 hours is ',  
FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1/4));  
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;  
end.
```

نلاحظ عندما زدنا الرقم 1 أو طرحناه من قيمة التاريخ الحالي **Now** فإننا نُزيد يوم أو نُنقص يوم على التوالي. وعندما زدنا نصف يوم $\frac{1}{2}$ فهي تعني 12 ساعة، في هذه الحالة أثرتنا على الجزء الكسري في التاريخ الذي يمثل الوقت.

في المثال التالي نكون تاريخ معين من السنين، والشهور، والأيام:

```
var
  ADate: TDateTime;
begin
  ADate:= EncodeDate(1975, 11, 16);
  Writeln('My date of birth is: ', FormatDateTime('yyyy-mm-dd', ADate));
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ في المثال السابق أننا استخدمنا الدالة **EncodeDate** والتي مدخلاتها هي السنة، الشهر ثم اليوم، وهي تحوّل وتجمّع هذه القيم في متغير واحد من النوع **TDateTime**.

في المثال التالي سوف نستخدم دالة مشابهة لكنها تختص بالوقت:

```
var
  ATime: TDateTime;
begin
  ATime:= EncodeTime(19, 22, 50, 0);
  Writeln('Almughrib prayer time is: ', FormatDateTime('hh:nn:ss', ATime));
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

مقارنة التواريخ والأوقات

يمكن مقارنة التواريخ والأوقات بنفس طريقة مقارنة الأعداد الحقيقية، فالرقم 9.3 أكبر من الرقم 5.1 . فكما علمنا يمثل العدد الصحيح الأيام أو التاريخ، ويمثل العدد الكسري الوقت. فبالتركيب قيمة **Now + 1**

والتي تعني غداً تكون دائماً أكبر من *Now* و قيمة $Now + 1/24$ والتي تعني بعد ساعة من الآن تكون أكبر من $Now - 2/24$ والتي تعني قبل ساعتين من الآن.

في البرنامج التالي نضع تاريخ معين وهو عام 2020 مثلاً، و اليوم الأول من شهر يناير، فنقارنه بالتاريخ الحالي لنعرف هل تجاوزنا هذا التاريخ أم ليس بعد.

```
var
  Year2020: TDateTime;
begin
  Year2020:= EncodeDate(2020, 1, 1);

  if Now < Year2020 then
    Writeln('Year 2020 is not coming yet')
  else
    Writeln('Year 2020 has already passed');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

يمكن إضافة معلومة جديدة في البرنامج وهي عدد الأيام المتبقية لهذا التاريخ أو الأيام التي انقضت بعده في المثال التالي:

```
var
  Year2020: TDateTime;
  Diff: Double;
begin
  Year2020:= EncodeDate(2020, 1, 1);
  Diff:= Abs(Now - Year2020);

  if Now < Year2020 then
    Writeln('Year 2020 is not coming yet, there are ',
      Format('%0.2f', [Diff]), ' days Remaining ')
  else
    Writeln('First day of January 2020 is passed by ',
      Format('%0.2f', [Diff]), ' Days');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

في المثال السابق سوف يحتوي المتغير *Diff* قيمة فرق التاريخين بالأيام. ونلاحظ أننا استخدمنا الدالة *Abs* والتي تُرجع القيمة المطلقة لعدد (أي حذف السالب إن وجد).

مسجل الأخبار

البرنامج التالي يستخدم الملفات النصية لتسجيل عناوين الأخبار المكتوبة و يُضيف الزمن الذي تمت كتابة الخبر عنده باعتباره وقت حدوث الخبر. عند إغلاق البرنامج وتشغيله مرة أخرى يعرض البرنامج الأخبار السابقة التي تمت كتابتها مع عرض الزمن الذي كتبت فيه.

```
program news;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  Title: string;
  F: TextFile;
begin
  AssignFile(F, 'news.txt');
  if FileExists('news.txt') then
  begin      // Display old news
    Reset(F);
    while not Eof(F) do
    begin
      Readln(F, Title);
      Writeln(Title);
    end;
    CloseFile(F); // reading is finished from old news
    Append(F);    // open file again for appending
  end
  else
    Rewrite(F);

  Write('Input current hour news title: ');
  Readln(Title);
  Writeln(F, DateTimeToStr(Now), ', ', Title);
  CloseFile(F);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

الثوابت constants

الثوابت تشبه المتغيرات من حيث أن كل يمثل قيمة معينة مثل قيمة لعدد صحيح، حقيقي أو مقطع. إلا أن الثوابت تبقى قيمتها ثابتة لا تتغير، بل أن قيمتها محددة في وقت ترجمة البرنامج. بالتالي نستطيع أن نقول أن المترجم يعرف قيمة الثابت سلفاً أثناء قيامه بترجمة البرنامج وتحويله إلى لغة آلة. مثال لقيمة ثابتة:

```
WriteLn(5);  
WriteLn('Hello');
```

نجد أن البرنامج عند تنفيذ هذا السطر يكتب الرقم 5 في كل الأحوال ولا يمكن أن تتغير هذه القيمة، كذلك فإن المقطع *Hello* لا يتغير.

عندما نتكلم عن الثوابت فإننا نقصد الأرقام المحددة أو المتغيرات المقطعية أو أي نوع آخر تمت كتابته في البرنامج بالطريقة السابقة أو بالطريقة التالية:

برنامج استهلاك الوقود

```
program PetrolConsumption;  
  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes, SysUtils  
  { you can add units after this };  
  
const GallonPrice = 6.5;  
  
var  
  Payment: Integer;  
  Consumption: Integer;  
  Kilos: Single;  
begin  
  Write('How much did you pay for your car's petrol: ');  
  ReadLn(Payment);  
  Write('What is the consumption of your car? (Kilos per Gallon): ');
```

```
Readln(Consumption);  
  
Kilos:= (Payment / GallonPrice) * Consumption;  
  
Writeln('This petrol will keep your car running for : ',  
Format('%0.1f', [Kilos]), ' Kilometers');  
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;  
end.
```

البرنامج السابق يحسب عدد الكيلومترات التي سوف تقطعها السيارة بناءً على عدة عوامل:

1. معدل استهلاك السيارة للوقود، وقد استخدمنا المتغير *Consumption* حتى نضع فيه عدد الكيلومترات التي تقطعها السيارة بالجالون الواحد من الوقود
2. قيمة النقود التي دفعها سائق السيارة لشراء الوقود، وقد استخدمنا المتغير *Payment*
3. سعر الجالون، وقد استخدمنا الثابت *GallonPrice* لتخزين قيمة سعر الجالون والتي تعتبر قيمة ثابتة نسبياً فثبتناها في البرنامج حتى لا نطلب من المستخدم إدخالها في كل مرة عند الدفع *payment* ولقراءة الاستهلاك *Consumption*. فإذا تغير سعر الوقود - وهذا من المفترض أن يحدث نادراً - ما علينا إلا الذهاب إلى أعلى البرنامج وتغيير قيمة هذه الثابت ثم إعادة ترجمة البرنامج، أما إذا كانت هذه القيم تتغير كل فترة مثلاً فالأفضل عدم تثبيتها في البرنامج.

يفضل استخدام هذه الطريقة للثوابت عندما نستخدمها لأكثر من مرة في البرنامج، فتغيير القيمة في التعريف تغني عن التغيير في كل أجزاء البرنامج.

القيم المناسبة لاستخدامها كثابت هي القيم التي لا تتغير، مثل قيمة Pi في الرياضيات والقيم المستخدمة للتحويلات، مثلاً معامل التحويل من متر إلى ذراع، ومن درجة حرارة فهرنهايت إلى درجة مئوية، فكلها قيم ثابتة سابقة التعريف لا تتغير، لذلك الأفضل وضعها كثوابت بدلاً من متغيرات.

Ordinal types الأنواع العددية

الأنواع العددية هي عبارة عن نوع يحمل قيم عددية صحيحة ذات أسماء، ونعاملها بأسمائها لتسهيل قراءة الكود وفهم البرنامج. مثلاً في البرنامج التالي نريد أن نخزن اللغة التي يريد استخدامها مستخدم البرنامج، وقد خیرناه بين اللغة العربية والإنجليزية، وكان من الممكن أن نستخدم متغير صحيح ونضع فيه قيم مثل 1 للغة العربية و 2 للغة الإنجليزية. لكننا استخدمنا الأنواع العددية ليكون البرنامج أكثر وضوحاً للمبرمج:

```
program OrdinaryTypes;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TLanguageType = (ltArabic, ltEnglish);

var
  Lang: TLanguageType;
  AName: string;
  Selection: Byte;
begin
  Write('Please select Language: 1 (Arabic), 2 (English)');
  Readln(Selection);

  if Selection = 1 then
    Lang:= ltArabic
  else
    if selection = 2 then
      Lang:= ltEnglish
    else
      Writeln('Wrong entry');

  if Lang = ltArabic then
    Write('ما هو اسمك: ')
  else
    if Lang = ltEnglish then
      Write('What is your name: ');

  Readln(AName);
```



```
if Lang = ltArabic then
begin
  Writeln('مرحباً بك ', AName);
  Write('الرجاء الضغط على مفتاح إدخال لإغلاق البرنامج');
end
else
if Lang = ltEnglish then
begin
  Writeln('Hello ', AName);
  Write('Please press enter key to close');
end;
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

الأرقام الصحيحة والحروف تعتبر من الأنواع العددية، أما الأعداد الكسرية والمقاطع فلا تعتبر من الأنواع العددية.

الهجوعات sets

المجموعات هي عبارة عن نوع من المتغيرات يستطيع جمع عدد من القيم أو الخصائص، شرط أن تكون من الأنواع العددية، فمثلاً عندما نريد تصنيف برامج من حيث أنظمة التشغيل التي تعمل فيها نلجأ إلى الآتي:

1. تعريف نوع محدود يمثل أنظمة التشغيل، كما في المثال التالي *TApplicationEnv*
2. تُعرّف متغيرات للبرامج المراد تصنيفها بالشكل التالي:

```
AppName: set of TApplicationEnv;
```

3. تُسند أنواع أنظمة التشغيل في شكل مجموعة كالتالي:

```
AppName:= [aeLinux, aeWindows];
```

والمثال هو:

```
program Sets;
{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
type
  TApplicationEnv = (aeLinux, aeMac, aeWindows);
var
  Firefox: set of TApplicationEnv;
  SuperTux: set of TApplicationEnv;
  Delphi: set of TApplicationEnv;
  Lazarus: set of TApplicationEnv;
begin
  Firefox:= [aeLinux, aeWindows];
  SuperTux:= [aeLinux];
  Delphi:= [aeWindows];
  Lazarus:= [aeLinux, aeMac, aeWindows];

  if aeLinux in Lazarus then
    Writeln('There is a version for Lazarus under Linux')
  else
```

```
Writeln('There is no version of Lazarus under linux');  
  
if aeLinux in SuperTux then  
  Writeln('There is a version for SuperTux under Linux')  
else  
  Writeln('There is no version of SuperTux under linux');  
  
if aeMac in SuperTux then  
  Writeln('There is a version for SuperTux under Mac')  
else  
  Writeln('There is no version of SuperTux under Mac');  
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;  
end.
```

يمكن كذلك استخدام صيغة المجموعات مع الأنواع العددية أو الصحيحة كالآتي:

مثال لاستخدام متغير صحيح: Month

```
if Month in [1, 3, 5, 7, 8, 10, 12] then  
  Writeln('This month contains 31 days');
```

كذلك يمكننا استخدام الحروف كالآتي:

```
if Char in ['أ', 'إ', 'أ', 'ا'] then  
  Writeln('هذا حرف الألف');
```

معالجة الاعتراضات Exception handling

يوجد نوعان من الأخطاء: أخطاء أثناء الترجمة compilation errors مثل استخدام متغير بدون تعريفه في قسم var، أو كتابة عبارة بطريقة خاطئة. فهذه الأخطاء لا يمكن تجاوزها، وتحول دون ترجمة وربط البرامج، يُنبه المترجم عن وجودها في نافذة Messages.

أما النوع الثاني من الأخطاء فهي الاعتراضات التي تحدث بعد تنفيذ البرنامج، مثلاً إذا طلب البرنامج من المستخدم إدخال رقمين لإجراء القسمة عليهما، ثم أدخل المستخدم صفرًا في المقسوم عليه، ففي هذه الحالة يحدث الخطأ المعروف بالقسمة على الصفر Division by Zero وهي من الأشياء الممنوعة في الرياضيات. أو مثل محاولة فتح ملف غير موجود، أو محاولة الكتابة في ملف ليس للمستخدم صلاحية عليه (من ناحية نظام التشغيل). فمثل هذه الأخطاء تسمى أخطاء في وقت التنفيذ run-time errors. وليس من السهولة أن يتوقعها المترجم، لذلك فمعالجة هذه الاستثناءات تُركت على عاتق المبرمج، حيث أن على المبرمج أن يكتب برنامج أو كود شديد الاعتمادية reliable يستطيع التغلب على معظم الأخطاء التشغيلية التي يمكن أن تحدث.

معالجة الاعتراضات لها عدة طرق، منها حماية جزء معين من الكود باستخدام عبارة try except أو try finally

عبارة try except

تكون بهذه الطريقة:

```
try
    // Start of protected code
    CallProc1;
    CallProc2;
    // End of protected code

except
on e: exception do // Exception handling
begin
    Writeln('Error: ' + e.message);
end;
end;
```

مثال لبرنامج قسمة عددين باستخدام except:

```
program ExceptionHandling;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, sysutils
  { you can add units after this };

var
  x, y: Integer;
  Res: Double;
begin
  try
    Write('Input x: ');
    Readln(x);
    Write('Input y: ');
    Readln(y);
    Res:= x / y;
    Writeln('x / y = ', Res);

  except
    on e: exception do
      begin
        Writeln('An error occurred: ', e.message);
      end;
    end;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ أن هناك قسمين للكود: الكود الموجود بين عبارتي `try .. except` وهو الكود المحمي، والكود الموجود بين عبارتي `except .. end` وهو يمثل قسم معالجة الاستثناءات، و يُنفذ فقط عند حدوث خطأ.

عندما تُدخل القيمة **صفر** للمتغير `y` فإن الكود في قسم معالجة الاستثناءات `except` يُنفذ. ومؤشر التنفيذ في البرنامج عند حدوث خطأ يقفز إلى قسم معالجة الاستثناءات ويترك تنفيذ باقي الكود المحمي، ففي هذه الحالة الكود التالي لا يُنفذ عند حدوث خطأ أثناء القسمة، مثل أن تكون قيمة `y` صفراً:

```
Writeln('x / y = ', Res);
```

أما في حالة عدم حدوث خطأ، أي في الحالات العادية يُنفذ كل الكود المحمي ولا يُنفذ الكود الموجود

في قسم معالجة الأخطاء باللون الأحمر.

عبارة try finally

```
try
  // Start of protected code
  CallProc1;
  CallProc2;
  // End of protected code

finally
  Writeln('This line will be printed in screen for sure');
end;
```

مثال لبرنامج قسمة عددين باستخدام finally:

```
program ExceptionHandling;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x, y: Integer;
  Res: Double;
begin
  try
    Write('Input x: ');
    Readln(x);
    Write('Input y: ');
    Readln(y);
    Res:= x / y;
    Writeln('x / y = ', Res);

  finally
    Writeln('Finished');
  end;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
```

```
//readln;  
end.
```

في هذه المرة فإن الكود في قسم `finally` باللون الأخضر يُنفذ في كافة الأحوال، في حال حدوث خطأ أو في حال عدم حدوث خطأ. وبهذه الطريقة نضمن تنفيذ كود معين في كافة الأحوال.

رفع الاستثناءات `raise an exception`

يمكن للمبرمج رفع استثناء، وذلك عند حدوث شرط معين، فمثلاً إذا طلبنا من المستخدم الالتزام بمدى معين لمُدخل ما، و تجاوز المستخدم هذا المدى، حينئذٍ يمكن افعال استثناء مصحوب برسالة خطأ معين. مثلاً البرنامج التالي:

```
program RaiseExcept;  
  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes, SysUtils  
  { you can add units after this };  
  
var  
  x: Integer;  
begin  
  Write('Please input a number from 1 to 10: ');  
  Readln(X);  
  try  
  
    if (x < 1) or (x > 10) then // rais exception  
      raise exception.Create('X is out of range');  
    Writeln(' X * 10 = ', x * 10);  
  
  except  
  on e: exception do // Catch my exception  
  begin  
    Writeln('Error: ' + e.Message);  
  end;  
end;  
end;
```

```
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;  
end.
```

نلاحظ أنه إذا تجاوز المستخدم المدى -والذي هو محدد من 1 إلى 10- فإن الخطأ التالي سوف يحدث في البرنامج *x is out of range* وبما أننا في نفس البرنامج قد حمينا الكود بعباراة *try except* فإن هذا الخطأ سوف يظهر للمستخدم بطريقة لا تؤثر على استمرارية البرنامج.

الفصل الثاني

البرمجة الهيكلية

Structured

Programming

مقدمة

البرمجة الهيكلية هي طريقة للبرمجة ظهرت الحاجة لها بعد التوسع الذي حدث في البرامج، حيث أصبح من الصعب كتابة وتنقيح وتطوير البرامج الكبيرة المكتوبة في شكل ملف واحد كتلة واحدة. البرمجة الهيكلية هي إتباع طريقة تقسيم البرامج إلى وحدات أو مكتبات وإجراءات ودوال في شكل منظم و مجزأ يمكن الاستفادة منها أكثر من مرة مع سهولة القراءة وتتبع الأخطاء في الأجزاء التي بها مشاكل في البرامج. فنجد أن من أهم أهداف البرمجة الهيكلية هي:

1. تقسيم البرنامج إلى وحدات برمجية أصغر تُسهل كتابة و تطوير البرامج
2. إمكانية استخدام (نداء) هذه الإجراءات في أكثر من موضع في البرنامج الواحد أو عدد من البرامج باستخدام الوحدات. وهذه الخاصية تسمى بإعادة استخدام الكود code re-usability
3. إمكانية مشاركة تصميم وكتابة برنامج واحد لأكثر من مبرمج، حيث يمكن لأي مبرمج كتابة وحدة منفصلة أو إجراء منفصل ثم تُجمع في برنامج واحد.
4. سهولة صيانة البرنامج، حيث تسهل معرفة الأخطاء وتصحيحها بالتركيز على الدالة أو الإجراء الذي ينتج عنه الخطأ.
5. إمكانية تقسيم البرامج إلى أقسام أو طبقات منطقية تجعل البرنامج قابل للتوسعة.

الإجراءات procedures

لعلنا استخدمنا البرمجة الهيكلية من قبل، فقد مر علينا كثير من الدوال والإجراءات والوحدات التي هي عبارة عن أجزاء من مكتبات فري باسكال، لكن الفرق الآن أن المبرمج في هذه المرحلة سوف نكتب إجراءات ودوال خاصة به تمثل خصوصية البرنامج الذي يكتبه.

في المثال التالي كتبنا إجراءان: *SayHello* و *SayGoodbye*

```
program Structured;  
{$mode objfpc}{$H+}  
uses
```

```
{ $IFDEF UNIX } { $IFDEF UseCThreads }
cthreads,
{ $ENDIF } { $ENDIF }
Classes
{ you can add units after this };

procedure SayHello;
begin
  Writeln('Hello there');
end;

procedure SayGoodbye;
begin
  Writeln('Good bye');
end;

begin // Here main application starts
  Writeln('This is the main application started');
  SayHello;
  Writeln('This is structured application');
  SayGoodbye;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

أظن أن الفكرة واضحة. ففي حالة مناداتنا لهذه الإجراءات يُنفذ البرنامج الكود المكتوب في الإجراء والذي كأنه برنامج مصغّر.

المدخلات Parameters

في البرنامج التالي نكتب إجراء يحتوي على مدخلات *parameters*. والمدخلات هي قيم تمرر عند نداء الإجراء:

```
procedure WriteSumm(x, y: Integer);
begin
  Writeln('The summation of ', x, ' + ', y, ' = ', x + y)
end;

begin
  WriteSumm(2, 7);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
```

end.

نلاحظ أن القيمتان 2 و 7 مُررتا عند نداء الإجراء *WriteSumm* والذي يستقبل هذه القيم في المتغيرات x, y ليكتب حاصل جمعهما.

في المثال التالي نُعيد كتابة برنامج المطعم بطريقة الإجراءات:

برنامج المطعم باستخدام الإجراءات

```
procedure Menu;
begin
  Writeln('Welcome to Pascal Restaurant. Please select your order');
  Writeln('1 - Chicken      (10 Geneh)');
  Writeln('2 - Fish          (7 Geneh)');
  Writeln('3 - Meat             (8 Geneh)');
  Writeln('4 - Salad           (2 Geneh)');
  Writeln('5 - Orange Juice (1 Geneh)');
  Writeln('6 - Milk            (1 Geneh)');
  Writeln;
end;

procedure GetOrder(AName: string; Minutes: Integer);
begin
  Writeln('You have ordered : ', AName, ', this will take ',
    Minutes, ' minutes');
end;

// Main application

var
  Meal: Byte;
begin
  Menu;
  Write('Please enter your selection: ');
  Readln(Meal);

  case Meal of
    1: GetOrder('Chicken', 15);
    2: GetOrder('Fish', 12);
    3: GetOrder('Meat', 18);
    4: GetOrder('Salad', 5);
    5: GetOrder('Orange juice', 2);
    6: GetOrder('Milk', 1);
  else
    Writeln('Wrong entry');
  end;
end.
```

نلاحظ أن البرنامج الرئيس أصبح أوضح وسهل الكتابة، والأجزاء الأخرى مثل القائمة الرئيسة *Menu* و الطلب منها أصبحتا إجراءان منفصلان. كذلك فإن الجزء الرئيس في البرنامج *Main* أصبح أصغر وحولت أجزاء كبيرة من الكود لتصبح في شكل إجراءات منفصلة. نلاحظ أن الإجراء *GetOrder* أستخدم عدد من المرات وفي كل مرة بمدخلات مختلفة، وهنا تحقق إعادة استخدام الكود.

الدوال functions

الدوال شبيهة بالإجراءات إلا أنها تزيد عليها بأنها تُرجع قيمة. فقد استخدمنا دوال من قبل مثل *Abs* التي ترجع القيمة المطلقة، و *UpperCase* التي تستقبل المقطع كمدخل، وتُرجع مقطع جديد مُحول إلى الحروف الإنجليزية الكبيرة.

في البرنامج التالي كتبنا دالة *GetSumm* والتي تستقبل مدخلين من الأعداد الصحيحة و تُرجع قيمة حاصل جمعها فقط دون كتابته:

```
function GetSumm(x, y: Integer): Integer;
begin
    Result:= x + y;
end;

var
    Sum: Integer;
begin
    Sum:= GetSumm(2, 7);
    Writeln('Summation of 2 + 7 = ', Sum);
    //For Windows, please uncomment below lines
    //writeln('Press enter key to continue..');
    //readln;
end.
```

نلاحظ هذه المرة أن للدالة نوع وهو *Integer* في هذه الحالة. كذلك استخدمنا الكلمة *Result* والتي تعبر عن القيمة التي سوف ترجع عند مناداة هذه الدالة. وعند ندائها استخدمنا المتغير *Sum* لاستقبال قيمة حاصل الجمع. لكن يمكن أن نستغني عن هذا المتغير بإدخال نداء الدالة في الإجراء *Writeln* وهو فرق آخر للدوال عن الإجراءات، حيث لا يمكن نداء إجراء بهذه الطريقة، فقط الدوال:

```
function GetSumm(x, y: Integer): Integer;
begin
    Result:= x + y;
end;

begin
    Writeln('Summation of 2 + 7 = ', GetSumm(2, 7));
    //For Windows, please uncomment below lines
    //writeln('Press enter key to continue..');
    //readln;
end.
```

في البرنامج التالي أعدنا كتابة برنامج المطعم باستخدام الدوال:

برنامج المطعم باستخدام الدوال وحلقة repeat

```
procedure Menu;
begin
  Writeln('Welcome to Pascal Resturant. Please select your order');
  Writeln('1 - Chicken      (10 Geneh)');
  Writeln('2 - Fish           (7 Geneh)');
  Writeln('3 - Meat             (8 Geneh)');
  Writeln('4 - Salad           (2 Geneh)');
  Writeln('5 - Orange Juice (1 Geneh)');
  Writeln('6 - Milk            (1 Geneh)');
  Writeln('X - nothing');
  Writeln;
end;

function GetOrder(AName: string; Minutes, Price: Integer): Integer;
begin
  Writeln('You have ordered: ', AName, ' this will take ',
    Minutes, ' minutes');
  Result:= Price;
end;

var
  Selection: Char;
  Price: Integer;
  Total: Integer;
begin
  Total:= 0;
  repeat
    Menu;
    Write('Please enter your selection: ');
    Readln(Selection);
    Price := 0;

    case Selection of
      '1': Price:= GetOrder('Chicken', 15, 10);
      '2': Price:= GetOrder('Fish', 12, 7);
      '3': Price:= GetOrder('Meat', 18, 8);
      '4': Price:= GetOrder('Salad', 5, 2);
      '5': Price:= GetOrder('Orange juice', 2, 1);
      '6': Price:= GetOrder('Milk', 1, 1);
      'x', 'X': Writeln('Thanks');
    else
      begin
        Writeln('Wrong entry');
```

```
        Price:= 0;
    end;
end;

Total:= Total + Price;

until (Selection = 'x') or (Selection = 'X');
writeln('Total price          = ', Total);
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

Local Variables المتغيرات المحلية

يمكن تعريف متغيرات محلية داخل الإجراء أو الدالة بحيث يكون استخدامها فقط محلياً في الإجراء أو الدالة ولا يمكن الوصول لهذه المتغيرات من البرنامج الرئيس. مثلاً:

```
procedure Loop(Counter: Integer);
var
    i: Integer;
    Sum: Integer;
begin
    Sum:= 0;
    for i:= 1 to Counter do
        Sum:= Sum + i;
    writeln('Summation of ', Counter, ' numbers is: ', Sum);
end;

begin // Main program section
    Loop;
    writeln('Returning to main program');
    //For Windows, please uncomment below lines
    //writeln('Press enter key to continue..');
    //readln;
end.
```

نلاحظ أن الإجراء *Loop* به تعريفات لمتغيرات محلية وهي *i*, *Sum* حيث يُحجز جزء من الذاكرة لهما مؤقتاً في ما يعرف بالمكدسة *Stack* وهي ذاكرة للمتغيرات المؤقتة مثل المتغيرات المحلية (*i*, *Sum*) في المثال السابق والمدخلات للإجراءات و الدوال مثل المدخل *Counter* في هذا المثال. وتتميز

المتغيرات التي تخزن في هذا النوع من الذاكرة بقصر عمرها، إذ أن المتغيرات تكون صالحة فقط أثناء تشغيل وتنفيذ هذه الدوال والإجراءات، فبعد نهاية تنفيذ الدالة والفراغ منها، كما في المثال السابق عند الوصول لهذه النقطة في البرنامج الرئيس:

```
WriteLn('Returning to main program');
```

تكون هذه المتغيرات قد انقضت ولا يمكن الوصول إليها أو لقيمها مرة أخرى بالطرق العادية. وهذه بخلاف المتغيرات العامة التي يُعرفها المبرمج في البرنامج الرئيس كما اعتدنا استخدامها سابقاً، فهي تكون أكثر عمراً حيث أنها تكون صالحة للاستخدام منذ بداية تشغيل البرنامج إلى نهايته، فإذا كان البرنامج يحتوي على حلقة فإنها تعمّر فترة طويلة، فإذا اشتغل البرنامج لمدة ساعة، كان عمر هذه المتغيرات ساعة، وهكذا.

هذه الطريقة لتعريف المتغيرات محلياً هي إحدى طرق تحقيق البرمجة الهيكلية، حيث أنها توفر حماية وخصوصية لهذا الجزء القائم بذاته من البرنامج (الإجراء أو الدالة) وتجعله غير مرتبط بمتغيرات أخرى في البرنامج مما يسهل نقله لبرامج أخرى أو الاستفادة منه عدة مرات. فمن أراد نداء هذه الدالة *Loop* ما عليه إلا مدها بقيمة *Counter* ثم يُلبي الإجراءات احتياجاته بتعريف متغيرات محلياً تعينه على تنفيذ هذا الكود مما يحقق إعادة استخدام الكود وسهولة الصيانة.

برنامج قاعدة بيانات الأخبار

البرنامج التالي فيه ثلاث إجراءات ودالة واحدة، وفيه عدة إمكانيات وهي : إضافي عنوان خبر جديد، عرض كافة الأخبار والبحث عن خبر باستخدام كلمة مفتاحية:

```
program news;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

type
  TNews = record
    ATime: TDateTime;
    Title: string[100];
  end;

procedure AddTitle;
var
  F: file of TNews;
  News: TNews;
begin
  AssignFile(F, 'news.dat');
  Write('Input current news title: ');
  Readln(News.Title);
  News.ATime:= Now;
  if FileExists('news.dat') then
  begin
    FileMode:= 2; // Read/Write
    Reset(F);
    Seek(F, System.FileSize(F)); // Go to last record to append
  end
  else
    Rewrite(F);
  Write(F, News);
  CloseFile(F);
end;

procedure ReadAllNews;
var
  F: file of TNews;
  News: TNews;
begin
  AssignFile(F, 'news.dat');
  if FileExists('news.dat') then
```

```
begin
  Reset(F);
  while not Eof(F) do
    begin
      Read(F, News);
      Writeln('-----');
      Writeln('Title: ', News.Title);
      Writeln('Time : ', DateTimeToStr(News.ATime));
    end;
  CloseFile(F);
end
else
  Writeln('Empty database');
end;

procedure Search;
var
  F: file of TNews;
  News: TNews;
  Keyword: string;
  Found: Boolean;
begin
  AssignFile(F, 'news.dat');
  Write('Input keyword to search for: ');
  Readln(Keyword);
  Found:= False;
  if FileExists('news.dat') then
    begin
      Reset(F);
      while not Eof(F) do
        begin
          Read(F, News);
          if Pos(LowerCase(Keyword), LowerCase(News.Title)) > 0 then
            begin
              Found:= True;
              Writeln('-----');
              Writeln('Title: ', News.Title);
              Writeln('Time : ', DateTimeToStr(News.ATime));
            end;
          end;
        CloseFile(F);
        if not Found then
          Writeln(Keyword, ' Not found');
        end
      else
        Writeln('Empty database');
    end;

function Menu: char;
begin
  Writeln;
  Writeln('.....News database.....');
  Writeln('1. Add news title');
```

```
Writeln('2. Display all news');
Writeln('3. Search');
Writeln('x. Exit');
Write('Please input a selection : ');
Readln(Result);
end;

// Main application

var
  Selection: Char;

begin

  repeat
    Selection:= Menu;
    case Selection of
      '1': AddTitle;
      '2': ReadAllNews;
      '3': Search;
    end;
  until LowerCase(Selection) = 'x';
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نجد أن برنامج الأخبار هذه المرة، أصبح أكثر وضوحاً ومقسماً إلى أقسام قائمة بذاتها، حيث يمكن للمبرمج حتى ولو لم يكن هو الذي كتب البرنامج أن يفهمه من أول وهلة وتعديله أو تطويره أو زيادة ميزات أخرى (زيادة إجراءات أو دوال) بكل سهولة. كذلك يمكن الاستفادة من بعض أجزائه في برامج أخرى.

الدالة كُدخل

الفرق الثاني والمهم بين الدوال والإجراءات هو أن الدالة يمكن نداؤها من داخل إجراء أو دالة أخرى كُدخل ، وهذه الميزة ناتجة عن الخاصية الأولى للدوال وهي أنها ترجع قيمة، أي يكن معاملتها كأنها قيمة ثابتة أو متغير نريد قراءة قيمته، مثلاً:

```
function DoubleNumber(x: Integer): Integer;
begin
    Result:= x * 2;
end;

// Main

begin
    Writeln('The double of 5 is : ', DoubleNumber(5));
    //For Windows, please uncomment below lines
    //writeln('Press enter key to continue..');
    //readln;
end.
```

نلاحظ أننا مررنا الدالة *DoubleNumber* داخل الإجراء *Writeln* ، وكان يمكن أن نستخدم الطريقة الطويلة كالآتي:

```
function DoubleNumber(x: Integer): Integer;
begin
    Result:= x * 2;
end;

// Main

var
    MyNum: Integer;
begin
    MyNum:= DoubleNumber(5);
    Writeln('The double of 5 is : ', MyNum);
    //For Windows, please uncomment below lines
    //writeln('Press enter key to continue..');
    //readln;
end.
```

كذلك يمكن نداء الدوال داخل الشروط وهذه الميزات لا تتحقق في الإجراءات:

```
function DoubleNumber(x: Integer): Integer;
begin
```

```
Result:= x * 2;
end;

// Main

begin
  if DoubleNumber(5) > 10 then
    Writeln('This number is larger than 10')
  else
    Writeln('This number is equal or less than 10);

  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

مخرجات الإجراءات و الدوال

لاحظنا من الأمثلة السابقة أننا نستطيع جعل الدالة ترجع قيمة واحدة، لكن ماذا لو أردنا إرجاع قيمتين أو أكثر، ففي هذه الحالة لا يصلح أن نستخدم فقط قيمة ما ترجعه الدالة. يكمن الحل لهذه المشكلة في استخدام المدخلات للإجراءات و الدوال. حيث نجد أن المدخلات نفسها يمكن أن تكون مخرجات بتغيير بسيط في طريقة التعريف. لكن لشرح هذا المفهوم دعنا نجرب هذه التجربة:

```
procedure DoSomething(x: Integer);
begin
  x:= x * 2;
  Writeln('From inside procedure: x = ', x);
end;

// main

var
  MyNumber: Integer;
begin
  MyNumber:= 5;

  DoSomething(MyNumber);
  Writeln('From main program, MyNumber = ', MyNumber);
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
```

```
//readln;  
end.
```

نجد في التجربة السابقة أن الإجراء *DoSomething* يستقبل مُدخل في المتغير *x* ثم يُضاعفه و يُظهره في الشاشة. وفي البرنامج الرئيس عرّفنا متغير *MyNumber* ووضعنا فيه القيمة 5 ثم نادينا بالإجراء *DoSomething* واستخدمنا المتغير *MyNumber* كمدخل لهذا الإجراء. في هذه الحالة تنتقل قيمة المتغير *MyNumber* إلى المتغير *x* في الإجراء السابق الذكر. في هذه الحالة يُضاعف الإجراء قيمة *x* والتي بدورها تحتوي على القيمة 5 لتصبح 10 ثم يُظهر هذه القيمة. بعد الفراغ من تنفيذ هذا الإجراء يعود مسار البرنامج إلى الجزء الرئيس من البرنامج وتُظهر قيمة المتغير *MyNumber* الذي نجده يحتفظ بالقيمة 5، أي أن مضاعفة قيمة *x* لم تؤثر عليه. هذه الطريقة من النداء تسمى النداء بالقيمة calling by value، حيث أننا نسخنا قيمة المدخل *MyNumber* إلى متغير *x* جديد مؤقت لا يؤثر على المتغير في البرنامج الرئيس.

النداء باستخدام المرجع calling by reference

إذا أضفنا الكلمة *var* إلى الإجراء السابق سوف تتغير الموازين فتصبح النتيجة مختلفة هذه المرة:

```
procedure DoSomething(var x: Integer);  
begin  
  x:= x * 2;  
  Writeln('From inside procedure: x = ', x);  
end;  
  
// main  
  
var  
  MyNumber: Integer;  
begin  
  MyNumber:= 5;  
  
  DoSomething(MyNumber);  
  Writeln('From main program, MyNumber = ', MyNumber);  
  //For Windows, please uncomment below lines  
  //writeln('Press enter key to continue..');  
  //readln;  
end.
```

في هذه الحالة نجد أن مضاعفة قيمة x أثرت مباشرة على قيمة *MyNumber* حيث أنهما يتشاركان الموقع أو العنوان في الذاكرة، أما في الحالة السابقة بدون *var* فكان لكل متغير منها حيز منفصل في الذاكرة.

لكن الشرط هذه المرة أن نمرر مدخل في شكل متغير من نفس النوع و لا يصلح أن يكون ثابت، حيث لا يمكن أن ننادي الدالة بهذه الطريقة:

```
DoSomething(5);
```

في الحالة الأولى كان يمكن أن يكون هذا الإجراء صحيح لأن الطريقة كانت نداء بوساطة قيمة وفي هذه الحالة القيمة فقط هي المهمة وهي 5.

في المثال التالي كتبنا إجراء لاستبدال قيمتين ببعضهما أو ما يسمى بالـ *Swap*

```
procedure SwapNumbers(var x, y: Integer);
var
    Temp: Integer;
begin
    Temp:= x;
    x:= y;
    y:= Temp;
end;

// main

var
    A, B: Integer;
begin

    Write('Please input value for A: ');
    Readln(A);

    Write('Please input value for B: ');
    Readln(B);

    SwapNumbers(A, B);
    Writeln('A = ', A, ', and B = ', B);
    //For Windows, please uncomment below lines
    //writeln('Press enter key to continue..');
    //readln;
end.
```


الوحدات units

الوحدات هي عبارة عن مكتبات للإجراءات و الدوال التي تستخدم بكثرة بالإضافة إلى الثوابت، و المتغيرات والأنواع المعرفة بواسطة المستخدم. ويمكن استخدام هذه الإجراءات و الدوال الموجودة في هذه الوحدات ونقلها لعدة برامج مما يحقق إحدى أهداف البرمجة الهيكلية وهي إعادة الاستفادة من الكود. كذلك فهي تقلل من ازدحام البرنامج الرئيس حيث تُوضع الإجراءات و الدوال التي تلي حل مشكلة معينة في وحدة خاصة حتى تسهل صيانة وقراءة الكود.

لإنشاء وحدة جديدة ما علينا إلى الذهاب إلى `File/New Unit` فنجد شكل وحدة جديدة فيها هذا الكود:

```
unit Unit1;
{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

implementation

end.
```

بعد ذلك نُعطيها اسم ونحفظها كملف بنفس الاسم، مثلاً إذا اخترنا اسم `Test` للوحدة بدلاً من `Unit1` لابد من حفظها كملف باسم `Test.pas`. بعد ذلك نكتب الإجراءات و الدوال التي نريد استخدامها لاحقاً:

```
unit Test;
{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

const
  GallonPrice = 6.5;

function GetKilometers(Payment, Consumption: Integer): Single;
```

implementation

```
function GetKilometers(Payment, Consumption: Integer): Single;
begin
  Result:= (Payment / GallonPrice) * Consumption;
end;

end.
```

أضفنا القيمة الثابتة: *GallonPrice* ودالة *GetKilometers* لتستخدم في أي برنامج. نلاحظ أننا أعدنا كتابة تعريف الدالة في قسم ال *Interface* أما كود الدالة فهو مكتوب في قسم *Implementation*. وضرورة إعادة كتابة تعريف الدالة أو الإجراءات في قسم ال *interface* يُمكن باقي البرامج من رؤية هذه الدالة ومن ثم مناداتها، فالدوال غير المكتوبة في هذا القسم لا يمكن نداؤها من خارج الوحدة، بل يمكن فقط نداؤها داخل هذه الوحدة من الإجراءات و الدوال التي تليها في الترتيب فقط.

حفظنا هذا الملف في نفس المجلد الذي يوجد فيه البرنامج التالي:

```
program PetrolConsumption;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this }, Test;

var
  Payment: Integer;
  Consumption: Integer;
  Kilos: Single;
begin
  Write('How much did you pay for your car''s petrol: ');
  Readln(Payment);
  Write('What is the consumption of your car (Kilos per one Gallon) ');
  Readln(Consumption);

  Kilos:= GetKilometers(Payment , Consumption);

  Writeln('This petrol will keep your car running for: ',
    Format('%0.1f', [Kilos]), ' Kilometers');
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
```

```
//readln;  
end.
```

نلاحظ أننا أضفنا اسم الوحدة *Test* في قسم الـ *Uses* و نادينا الدالة *GetKilometers* في هذا البرنامج.

لاستخدام وحدةٍ ما في برنامج يجب تحقق واحد من هذه الشروط:

1. يكون ملف الوحدة موجود في نفس المجلد الذي يوجد فيه البرنامج
2. فتح ملف الوحدة المعينة ثم إضافتها للمشروع بواسطة Project/Add Editor File to project
3. إضافة مسار أو مجلد الوحدة في Project/Compiler Options/Paths/Other Unit Files

الوحدات وبنية لازاراس وفري باسكال

نجد أن الوحدات في مترجم فري باسكال وأداة التطوير لازاراس تمثل البنية الأساسية للغة البرمجة، حيث نجدهما يحتويان على عدد كبير من الوحدات التي تمثل مكتبة لغة أوبجكت باسكال والتي بدورها تحتوي على عدد كبير جداً من الدوال والإجراءات والأنواع وغيرها الكثير من الأشياء التي جعلت لغة أوبجكت باسكال لغة عنية ثرية بكافة الأدوات التي يحتاج إليها المبرمج لبناء النظم والبرامج الكبيرة بسرعة وبدون جهد كبير. والسبب في ذلك أن هذه اللغة أخذت وقت كافي من التطوير والاستخدام وقد صاحب هذا الاستخدام بناء وحدات تمثل الخبرة التي جمعها المبرمجون من كافة أرجاء العالم لينشروها في شكل مكتبات مُجربة يمكن الاستفادة منها في معظم البرامج.

هذه الوحدات التي تحتويها لازاراس هي وحدات عامة مثل *Classes, SysUtils* وغيرها.

الوحدات التي يكتبها المبرمج

الوحدات التي يحتاج أن يكتبها المبرمج هي تعتبر وحدات خاصة تلبي حاجة برنامج الذي يكتبه ويصممه، ويُلبي كذلك حاجة البرامج المشابهة، فيكون بذلك قد بنى مكتبة من الإجراءات والدوال يُعيد استخدامها في عدد من البرامج.

مثلاً إذا افترضنا أن المبرمج يطور برنامج لإدارة صيانة السيارات، فإن الوحدات التي يكتبها يمكن أن تحتوي على إجراءات خاصة بالسيارات وبرنامج الصيانة، مثل إجراء تسجيل سيارة جديدة، أو البحث عن سيارة بدلالة رقم اللوحة، أو إضافة سجل صيانة، إلخ.

ففي هذه الحالة فإن المبرمج يكتب مكتبة أو وحدات تحتوي على دوال وإجراءات تمكنه من بناء قاعدة من الأدوات التي تسهل له بناء البرنامج الرئيس بسرعة وبوضوح ويحقق إعادة الاستخدام لتلك المكتبة. كذلك يمكن لعدد من المبرمجين أن ينفردوا بكتابة وحدات تمثل جزئية معينة من النظام، وعند تجميعها نكون قد حصلنا على برنامج كبير كتبه عدد من المبرمجين دون أن يكون هناك تعارض في أجزائه.

وحدة التاريخ الهجري

التاريخ الهجري له أهمية كبيرة للمسلمين حيث أن كل المعاملات الشرعية المرتبطة بتاريخ أو زمن معين فإن هذا الزمن أو الفترة تكون محسوبة بالتاريخ الهجري، مثلاً زكاة المال مرتبطة بتمام الحول وهو عام هجري.

التاريخ الهجري مبني على السنة القمرية وهي تحتوي على 12 شهر، كانت تستخدم من زمن الجاهلية، لكن عمر بن الخطاب رضى الله عنه اعتمد هجرة الرسول صلى الله عليه وسلم إلى المدينة هي بداية لعهد جديد يبدأ المسلمون بحساب السنوات استناداً له، وهو يوم 16 يوليو 622 ميلادية.

السنة القمرية فيها 354.367056 يوم، والشهر القمري فيه 29.530588 يوم.

وقد كتبت وحدة للتحويل بين السنة الهجرية والميلادية بناءً على هذه الثوابت. ويمكن أن يكون الخطأ في التحويل هو يوم واحد فقط زيادةً أو نقصاناً.

وكود الوحدة هو:

```
{
*****

HejriUtils:   Hejri Calnder converter, for FreePascal and Delphi
Author:       Motaz Abdel Azeem
email:        motaz@code.sd
Home page:    http://motaz.freevar.com/
License:      LGPL
Created on:   26.Sept.2009
Last modifie: 26.Sept.2009

*****
}

unit HejriUtils;

{$IFDEF FPC}
{$mode objfpc}{$H+}
{$ENDIF}

interface

uses
  Classes, SysUtils;

const
  HejriMonthsEn: array [1 .. 12] of string = ('Moharram', 'Safar', 'Rabie Awal',
    'Rabie Thani', 'Jumada Awal', 'Jumada Thani', 'Rajab', 'Shaban', 'Ramadan',
    'Shawal', 'Thi-Alqaida', 'Thi-Elhajah');
```

```
HejriMonthsAr: array [1 .. 12] of string = ('ربيع أول', 'صفر', 'محرم',  
'ربيع ثاني', 'جمادى الأول', 'جمادى الآخر', 'رجب', 'شعبان', 'رمضان',  
'شوال', 'ذي القعدة', 'ذي الحجة');
```

```
HejriYearDays = 354.367056;  
HejriMonthDays = 29.530588;
```

```
procedure DateToHejri(ADateTime: TDateTime; var Year, Month, Day: Word);  
function HejriToDate(Year, Month, Day: Word): TDateTime;
```

```
procedure HejriDifference(Year1, Month1, Day1, Year2, Month2, Day2: Word;  
var YearD, MonthD, DayD: Word);
```

implementation

```
var  
HejriStart : TDateTime;
```

```
procedure DateToHejri(ADateTime: TDateTime; var Year, Month, Day: Word);  
var  
HejriY: Double;  
Days: Double;  
HejriMonth: Double;  
RDay: Double;  
begin  
HejriY:= ((Trunc(ADateTime) - HejriStart - 1) / HejriYearDays);  
Days:= Frac(HejriY);  
Year:= Trunc(HejriY) + 1;  
HejriMonth:= ((Days * HejriYearDays) / HejriMonthDays);  
Month:= Trunc(HejriMonth) + 1;  
RDay:= (Frac(HejriMonth) * HejriMonthDays) + 1;  
Day:= Trunc(RDay);  
end;
```

```
function HejriToDate(Year, Month, Day: Word): TDateTime;  
begin  
Result:= (Year - 1) * HejriYearDays + (HejriStart - 0) +  
(Month - 1) * HejriMonthDays + Day + 1;  
end;
```

```
procedure HejriDifference(Year1, Month1, Day1, Year2, Month2, Day2: Word; var  
YearD, MonthD, DayD: Word);  
var  
Days1: Double;  
Days2: Double;  
DDays: Double;  
RYear, RMonth: Double;  
begin  
Days1:= Year1 * HejriYearDays + (Month1 - 1) * HejriMonthDays + Day1 - 1;  
Days2:= Year2 * HejriYearDays + (Month2 - 1) * HejriMonthDays + Day2 - 1;  
DDays:= Abs(Days2 - Days1);
```

```
RYear:= DDays / HejriYearDays;  
RMonth:= (Frac(RYear) * HejriYearDays) / HejriMonthDays;  
DayD:= Round(Frac(RMonth) * HejriMonthDays);  
  
YearD:= Trunc(RYear);  
MonthD:= Trunc(RMonth);  
  
end;  
  
initialization  
  
HejriStart:= EncodeDate(622, 7, 16);  
  
end.
```

وتحتوي هذه الوحدة على الدوال والإجراءات التالية:

1. `DateToHejri`: وتستخدم لتحويل تاريخ ميلادي لما يقابله من التاريخ الهجري، مثلاً:

```
program Project1;  
  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}  
  cthreads,  
  {$ENDIF}{$ENDIF}  
  Classes , HejriUtils, SysUtils  
  { you can add units after this };  
  
var  
  Year, Month, Day: Word;  
begin  
  DateToHejri(Now, Year, Month, Day);  
  Writeln('Today in Hejri: ', Day, '-', HejriMonthsEn[Month],  
    '-', Year);  
  //For Windows, please uncomment below lines  
  //writeln('Press enter key to continue..');  
  //readln;  
end.
```

2. `HejriToDate`: ويستخدم للتحويل من تاريخ هجري لما يقابله بالميلادي.

3. `HejriDifference`: ويستخدم لمعرفة الفارق الزمني بين تاريخين هجريين.

يوجد مثال آخر للتحويل بين التاريخ الهجري والميلادي في فصل الواجهة الرسومية، واسم المثال هو المحوّل الهجري.

تحميل الإجراءات و الدوال Procedure and function Overloading

المقصود بتحميل الإجراءات و الدوال هو أن تكون هناك أكثر من دالة أو إجراء مشترك في الاسم لكن مع تغيير المدخلات من حيث العدد أو النوع. مثلاً لو أردنا أن نكتب دالة لجمع عددين صحيحين أو حقيقيين، بحيث عندما يريد المبرمج نداء دالة لتجميع أعداد Sum مثلاً، مع أي من نوع المتغيرات السابقة فإن البرنامج يختار الدالة المناسبة:

```
program sum;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

function Sum(x, y: Integer): Integer; overload;
begin
  Result:= x + y;
end;

function Sum(x, y: Double): Double; overload;
begin
  Result:= x + y;
end;

var
  j, i: Integer;
  h, g: Double;
begin
  j:= 15;
  i:= 20;
  Writeln(J, ' + ', I, ' = ', Sum(j, i));

  h:= 2.5;
  g:= 7.12;
  Writeln(H, ' + ', G, ' = ', Sum(h, g));
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نلاحظ أننا أضفنا الكلمة المحجوزة *overload* في نهاية تعريف الدالة، وذلك لكتابة دالتين بنفس الاسم مع اختلاف نوع المتغير وناتج الدالة.

القيم الافتراضية للمدخلات default parameters

يمكن أن نضع قيم افتراضية في مدخلات الدوال أو الإجراءات. وفي هذه الحالة يمكن تجاهل هذه المدخلات عند النداء أو استخدامها كما في المثال التالي:

```
program defaultparam;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

function SumAll(a, b: Integer; c: Integer = 0;
  d: Integer = 0): Integer;
begin
  result:= a + b + c + d;
end;

begin
  Writeln(SumAll(1, 2));
  Writeln(SumAll(1, 2, 3));
  Writeln(SumAll(1, 2, 3, 4));
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

نجد أن كل النداءات السابقة صحيحة، وفي حالة عدم إرسال متغير معين تُسند القيمة الافتراضية له، وفي هذه الحالة القيمة صفر للمتغيرات **c** و **d**.

وفي حالة تجاهل المتغيرات وعدم إرسالها، يجب التقيّد بالترتيب، حيث يمكن تجاهل **d** فقط، أو تجاهل **c** و **d**، لكن لا يمكن تجاهل المتغير أو المُدخل **c** وإرسال **d**. فإذا تجاهلنا إرسال قيمة **c** لابد من تجاهل **d** تبعاً لذلك. لذلك على المبرمج أن يضع المدخلات الغير مهمة في نهاية تعريف الدالة والأكثر أهمية في بدايتها. أي أن الأقل أهمية في المثال السابق كان المتغير **d**.

ترتيب البيانات sorting

ترتيب البيانات هو موضوع يقع ضمن مفهوم هيكلية البيانات Data Structure، وقد ذكرنا في هذا الفصل كمثال للبرامج التي تعتمد البرمجة الهيكلية والإجراءات لتحقيق ترتيب البيانات. ترتيب البيانات نحتاجه مع المصفوفات، فمثلاً إذا افترضنا أن لدينا مصفوفة فيها أرقام، ونريد ترتيب هذه الأرقام تصاعدياً أو تنازلياً، في هذه الحالة يمكننا استخدام عدة طرق لتحقيق هذا الهدف.

خوارزمية ترتيب الفقاعة bubble sort

وهي من أسهل طرق الترتيب، حيث يقارن إجراء الترتيب العنصر الأول في المصفوفة مع العنصر الثاني، و في حالة الترتيب التصاعدي إذا وُجد أن العنصر الأول ذو قيمة أكبر من العنصر الثاني، فإن البرنامج يُبدّل تلك العناصر، ثم يقارن العنصر الثاني مع الثالث إلى نهاية المصفوفة. ثم يُكرر هذه العملية عدة مرات حتى تترتب كامل المصفوفة.

إذا افترضنا أن لدينا 6 عناصر في المصفوفة أدخل بالشكل أدناه:

```
7
10
2
5
6
3
```

نجد أن الإجراء يحتاج لأكثر من دورة لإتمام العملية. حيث أن في كل دورة يُقارن العنصر الأول مع الثاني وإبدالهما إذا كان الأول أكبر من الثاني، ثم الثاني مع الثالث، والثالث مع الرابع، ثم الرابع مع الخامس، ثم الخامس مع السادس. فإذا لم تُرتب بالكامل فسوف نحتاج لدورة أخرى، ثم ثالثة ورابعة إلى أن تُرتب بالكامل.

في الدورة الأولى تصبح المتغيرات في المصفوفة كالآتي:

```
7
2
5
6
3
```

10

وفي الدورة الثانية:

2
5
6
3
7
10

وفي الدورة الثالثة:

2
5
3
6
7
10

وفي الدورة الرابعة:

2
3
5
6
7
10

نلاحظ أنه بنهاية الدورة الرابعة ترتبت كامل الأرقام، لكن ليس من السهولة معرفة أن الترتيب قد اكتمل إلا بإضافة دورة خامسة، وفي هذه الدورة نجد أن الإجراء لم يُبدل أي متغيرات بالمصفوفة، وبهذه الطريقة نعرف أن البيانات قد رُتبت. وجاء اسم الترتيب الفقاعي بسبب أن القيم الأقل تطفو للأعلى في كل دورة.

كود برنامج ترتيب الفقاعة:

```
program BubbleSortProj;  
  
{$mode objfpc}{$H+}  
  
uses  
  {$IFDEF UNIX}{$IFDEF UseCThreads}
```

```
cthreads,  
{ $ENDIF } { $ENDIF }  
Classes;  
  
procedure BubbleSort(var X: array of Integer);  
var  
    Temp: Integer;  
    i: Integer;  
    Changed: Boolean;  
begin  
    repeat // Outer loop  
        Changed := False;  
        for i := 0 to High(X) - 1 do // Inner loop  
            if X[i] > X[i + 1] then  
                begin  
                    Temp := X[i];  
                    X[i] := X[i + 1];  
                    X[i + 1] := Temp;  
                    Changed := True;  
                end;  
        until not Changed;  
    end;  
  
var  
    Numbers: array [0 .. 9] of Integer;  
    i: Integer;  
begin  
    Writeln('Please input 10 random numbers');  
    for i := 0 to High(Numbers) do  
        begin  
            Write('#', i + 1, ': ');  
            Readln(Numbers[i]);  
        end;  
  
    BubbleSort(Numbers);  
    Writeln;  
    Writeln('Numbers after sort: ');  
  
    for i := 0 to High(Numbers) do  
        begin  
            Writeln(Numbers[i]);  
        end;  
    //For Windows, please uncomment below lines  
    //writeln('Press enter key to continue..');  
    //readln;  
end.
```

يمكن تحويل الترتيب التنازلي إلى ترتيب تصاعدي (الأكبر أولاً ثم الأصغر) وذلك بتحويل معامل المقارنة من أكبر من إلى أصغر من كالتالي:

```
if X[i] < X[i + 1] then
```

وفي المثال التالي تُرتب درجات الطلاب تصاعدياً لنعرف الأول ثم الثاني إلى الأخير.
وفي هذا المثال استخدمنا مصفوفة سجلات لتسجيل اسم الطالب ودرجته:

برنامج ترتيب درجات الطلاب :

```
program smSort; // Stuent's mark sort

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TStudent = record
    StudentName: string;
    Mark: Byte;
  end;

procedure BubbleSort(var X: array of TStudent);
var
  Temp: TStudent;
  i: Integer;
  Changed: Boolean;
begin
  repeat
    Changed:= False;
    for i:= 0 to High(X) - 1 do
      if X[i].Mark < X[i + 1].Mark then
        begin
          Temp:= X[i];
          X[i]:= X[i + 1];
          X[i + 1]:= Temp;
          Changed:= True;
        end;
    until not Changed;
end;

var
  Students: array [0 .. 9] of TStudent;
  i: Integer;

begin
  Writeln('Please input 10 student names and marks');
  for i:= 0 to High(Students) do
    begin
```

```
Write('Student #', i + 1, ' name : ');
Readln(Students[i].StudentName);

Write('Student #', i + 1, ' mark : ');
Readln(Students[i].Mark);
Writeln;
end;

BubbleSort(Students);
Writeln;
Writeln('Marks after Buble sort: ');
Writeln('-----');

for i:= 0 to High(Students) do
begin
  Writeln('# ', i + 1, ' ', Students[i].StudentName,
    ' with mark (' , Students[i].Mark, ')');
end;
//For Windows, please uncomment below lines
//writeln('Press enter key to continue..');
//readln;
end.
```

طريقة ترتيب الفقاعة تمتاز بالبساطة، حيث أن المبرمج يمكن أن يحفظها ويكتب الكود كل مرة بدون الرجوع لأي مرجع. وهي مناسبة للمصفوفات الصغيرة والشبه مرتبة، أما إذا كانت المصفوفة تحتوي على عناصر كثيرة (عشرات الآلاف مثلاً) وكانت عشوائية تماماً ففي هذه الحالة سوف يستغرق هذا النوع من الترتيب وقتاً ويجب اللجوء إلى نوع آخر من الترتيب كما في الأمثلة اللاحقة.

نلاحظ أن خوارزمية ترتيب الفقاعة تعتمد على دورتين، دورة خارجية تستخدم repeat until وعدد تكرارها غير محددة، معتمدة على شكل البيانات، فإذا كانت البيانات مرتبة فإن عدد التكرار يكون مرة واحدة فقط، أما إذا كانت عشوائية تماماً فإن عدد التكرار يكون بعدد عناصر المصفوفة. والدورة الداخلية باستخدام for loop يكون تكرارها دائماً بعدد عناصر المصفوفة ناقص واحد. فإذا افترضنا أن لدينا 1000 عنصر في المصفوفة وكانت المصفوفة مُرتبة فإن عدد الدورات الكلية هو دورة واحدة خارجية مضروبة في 999، أي الحصيلة 999 دورة. وإذا كانت نفس المصفوفة عشوائية تماماً فإن عدد الدورات الخارجية تكون 1000 مضروبة في 999 دورة داخلية، فتكون الحصيلة 999000 دورة. فكلما زاد عدد عناصر المصفوفة وزادت عشوائية فإن كفاءة هذا النوع من الترتيب يسوء.

خوارزمية الترتيب الاختياري Selection Sort

هذه الطريقة أشبه بطريقة الفقاعة، إلا أنها أسرع مع البيانات الكبيرة، حيث أنها تحتوي على دورتين، دورة خارجية مشابهة لدورة repeat until في المثال السابق، وهي تدور بعدد عناصر المصفوفة ناقص واحد، أما الدورة الداخلية فتقل في كل دورة بمقدار واحد، إلى أن تصل إلى دورتين.

```
program SelectionSort;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

procedure SelectSort(var X: array of Integer);
var
  i: Integer;
  j: Integer;
  SmallPos: Integer;
  Smallest: Integer;
begin
  for i:= 0 to High(X) -1 do // Outer loop
  begin
    SmallPos:= i;
    Smallest:= X[SmallPos];
    for j:= i + 1 to High(X) do // Inner loop
      if X[j] < Smallest then
      begin
        SmallPos:= j;
        Smallest:= X[SmallPos];
      end;
    X[SmallPos]:= X[i];
    X[i]:= Smallest;
  end;
end;

// Main application

var
  Numbers: array [0 .. 9] of Integer;
  i: Integer;
begin
  Writeln('Please input 10 random numbers');
  for i:= 0 to High(Numbers) do
  begin
    Write('#', i + 1, ': ');
```

```
    Readln(Numbers[i]);  
end;  
  
SelectSort(Numbers);  
Writeln;  
Writeln('Numbers after Selection sort: ');  
  
for i:= 0 to High(Numbers) do  
begin  
    Writeln(Numbers[i]);  
end;  
//For Windows, please uncomment below lines  
//writeln('Press enter key to continue..');  
//readln;  
end.
```

بالرغم من أن خوارزمية الترتيب الاختياري أسرع في حالة البيانات الكبيرة، إلا أنها لا ترتبط بمدى ترتيب أو عشوائية المصفوفة، ففي كل الحالات نجد أن سرعتها شبه ثابتة، حيث يحدث أحياناً تبديل لنفس العنصر مع نفسه بعد فراغ الحلقة الداخلية.

خوارزمية الترتيب Shell

تمتاز هذه الخوارزمية بالسرعة العالية مع البيانات الكبيرة، وسلوكها مشابهة لخوارزمية الفقاعة في حالة البيانات المرتبة أو الشبه مرتبة، إلا أنها أكثر تعقيداً من الخوارزميتين السابقتين. وسميت بهذا الاسم نسبة لمخترعها Donald Shell.

```
program ShellSort;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes;

procedure Shells(var X: array of Integer);
var
  Done: Boolean;
  Jump, j, i: Integer;
  Temp: Integer;
begin
  Jump:= High(X);
  while (Jump > 0) do // Outer loop
  begin
    Jump:= Jump div 2;
    repeat // Intermediate loop
      Done:= True;
      for j:= 0 to High(X) - Jump do // Inner loop
      begin
        i:= j + Jump;
        if X[j] > X[i] then // Swap
          begin
            Temp:= X[i];
            X[i]:= X[j];
            X[j]:= Temp;
            Done:= False;
          end;
        end; // end of inner loop
      until Done; // end of intermediate loop
    end; // end of outer loop
  end;

var
  Numbers: array [0 .. 9] of Integer;
```

```
i: Integer;

begin
  Writeln('Please input 10 random numbers');
  for i:= 0 to High(Numbers) do
    begin
      Write('#', i + 1, ': ');
      Readln(Numbers[i]);
    end;

    Shells(Numbers);
    Writeln;
    Writeln('Numbers after Shell sort: ');

    for i:= 0 to High(Numbers) do
      begin
        Writeln(Numbers[i]);
      end;
      //For Windows, please uncomment below lines
      //writeln('Press enter key to continue..');
      //readln;
end.
```

ترتيب المقاطع

يمكن ترتيب المقاطع بنفس طريقة ترتيب الأسماء، حيث أن المقاطع تُرتب حسب الأحرف، فمثلاً الحرف **A** قيمته أقل من الحرف **B**. حيث أن قيمة الحرف **A** في الذاكرة هو عبارة عن الرقم 65 والحرف **B** قيمته 66. في المثال التالي أعدنا كتابة برنامج ترتيب درجات الطلاب، لكن هذه المرة تُرتب هجائياً حسب الاسم وليس الدرجة:

برنامج ترتيب الطلاب بالأسماء

```
program smSort; // Stuent's mark sort by name

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes;

type
  TStudent = record
    StudentName: string;
    Mark: Byte;
  end;

procedure BubbleSort(var X: array of TStudent);
var
  Temp: TStudent;
  i: Integer;
  Changed: Boolean;
begin
  repeat
    Changed := False;
    for i := 0 to High(X) - 1 do
      if X[i].StudentName > X[i + 1].StudentName then
        begin
          Temp := X[i];
          X[i] := X[i + 1];
          X[i + 1] := Temp;
          Changed := True;
        end;
    until not Changed;
end;
```

```
var
  Students: array [0 .. 9] of TStudent;
  i: Integer;

begin
  Writeln('Please input 10 student names and marks');
  for i:= 0 to High(Students) do
  begin
    Write('Student #', i + 1, ' name : ');
    Readln(Students[i].StudentName);

    Write('Student #', i + 1, ' mark : ');
    Readln(Students[i].Mark);
    Writeln;
  end;

  BubbleSort(Students);
  Writeln;
  Writeln('Marks after Bubble sort: ');
  Writeln('-----');

  for i:= 0 to High(Students) do
  begin
    Writeln('# ', i + 1, ' ', Students[i].StudentName,
      ' with mark (' , Students[i].Mark, ')');
  end;
  //For Windows, please uncomment below lines
  //writeln('Press enter key to continue..');
  //readln;
end.
```

مقارنة خوارزميات الترتيب

في هذا البرنامج سوف تُدخل أرقام عشوائية في مصفوفة كبيرة ونجري عليها أنواع الترتيب الثلاث السابقة ونقارن الزمن الذي تأخذه كل خوارزمية:

```
program SortComparison;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils;
```

```
// Bubble sort

procedure BubbleSort(var X: array of Integer);
var
    Temp: Integer;
    i: Integer;
    Changed: Boolean;
begin
    repeat
        Changed:= False;
        for i:= 0 to High(X) - 1 do
            if X[i] > X[i + 1] then
                begin
                    Temp:= X[i];
                    X[i]:= X[i + 1];
                    X[i + 1]:= Temp;
                    Changed:= True;
                end;
        until not Changed;
end;

// Selection Sort

procedure SelectionSort(var X: array of Integer);
var
    i: Integer;
    j: Integer;
    SmallPos: Integer;
    Smallest: Integer;
begin
    for i:= 0 to High(X) -1 do // Outer loop
        begin
            SmallPos:= i;
            Smallest:= X[SmallPos];
            for j:= i + 1 to High(X) do // Inner loop
                if X[j] < Smallest then
                    begin
                        SmallPos:= j;
                        Smallest:= X[SmallPos];
                    end;
            X[SmallPos]:= X[i];
            X[i]:= Smallest;
        end;
end;

// Shell Sort

procedure ShellSort(var X: array of Integer);
var
    Done: Boolean;
    Jump, j, i: Integer;
    Temp: Integer;
begin
```

```
Jump:= High(X);
while (Jump > 0) do // Outer loop
begin
  Jump:= Jump div 2;
  repeat // Intermediate loop
    Done:= True;
    for j:= 0 to High(X) - Jump do // Inner loop
    begin
      i:= j + Jump;
      if X[j] > X[i] then // Swap
      begin
        Temp:= X[i];
        X[i]:= X[j];
        X[j]:= Temp;
        Done:= False;
      end;

    end; // inner loop
  until Done; // innermediate loop end
end; // outer loop end
end;

// Randomize Data

procedure RandomizeData(var X: array of Integer);
var
  i: Integer;
begin
  Randomize;
  for i:= 0 to High(X) do
    X[i]:= Random(10000000);
end;

var
  Numbers: array [0 .. 59999] of Integer;
  i: Integer;
  StartTime: TDateTime;

begin
  Writeln('----- Full random data');
  RandomizeData(Numbers);
  StartTime:= Now;
  Writeln('Sorting.. Bubble');
  BubbleSort(Numbers);
  Writeln('Bubble sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
  Writeln;

  RandomizeData(Numbers);
  Writeln('Sorting.. Selection');
  StartTime:= Now;
  SelectionSort(Numbers);
  Writeln('Selection sort tooks (mm:ss): ',
```

```
FormatDateTime('nn:ss', Now - StartTime));
WriteLn;

RandomizeData(Numbers);
WriteLn('Sorting.. Shell');
StartTime:= Now;
ShellSort(Numbers);
WriteLn('Shell sort tooks (mm:ss): ',
FormatDateTime('nn:ss', Now - StartTime));

WriteLn;
WriteLn('----- Nearly sorted data');
Numbers[0]:= Random(10000);
Numbers[High(Numbers)]:= Random(10000);
StartTime:= Now;
WriteLn('Sorting.. Bubble');
BubbleSort(Numbers);
WriteLn('Bubble sort tooks (mm:ss): ',
FormatDateTime('nn:ss', Now - StartTime));
WriteLn;

Numbers[0]:= Random(10000);
Numbers[High(Numbers)]:= Random(10000);
WriteLn('Sorting.. Selection');
StartTime:= Now;
SelectionSort(Numbers);
WriteLn('Selection sort tooks (mm:ss): ',
FormatDateTime('nn:ss', Now - StartTime));
WriteLn;

Numbers[0]:= Random(10000);
Numbers[High(Numbers)]:= Random(10000);
WriteLn('Sorting.. Shell');
StartTime:= Now;
ShellSort(Numbers);
WriteLn('Shell sort tooks (mm:ss): ',
FormatDateTime('nn:ss', Now - StartTime));
//For Windows, please uncomment below lines
//writeLn('Press enter key to continue..');
//readLn;
end.
```

الفصل الثالث

الواجهة الرسومية

Graphical User Interface

مقدمة

الواجهة الرسومية أو واجهة المستخدم الرسومية هي بديل حديث لواجهة سطر الأوامر command line، أو الواجهة النصية text interface، وهي أكثر سهولة للمستخدم، وتستخدم فيها إمكانيات جديدة مثل الماوس، الأزرار، مربعات النصوص، وإلى ما ذلك مما اعتاده المستخدم الآن. وكمثال للبرامج ذات الواجهة الرسومية في نظام لينكس: متصفح الإنترنت فيرفوكس، برامج المكتب Office، الألعاب وغيرها. فبرامج الواجهة الرسومية أصبحت تشكل البنية الأساسية في كل أنظمة الحاسوب، مثل الأنظمة المحاسبية، وبرامج التصميم والألعاب التي تحتوي على مكتبات متقدمة ومعقدة للواجهة الرسومية، كذلك فإن أجزاء أنظمة التشغيل مثل سطح المكتب Gnome و KDE هي عبارة عن مجموعة من برامج ذات واجهات رسومية.

لكتابة برامج ذات واجهة رسومية يجب استخدام إحدى المكتبات المشهورة المتوفرة في هذا المجال مثل:

المستخدمة في نظام لينكس وبعض واجهات الهواتف النقالة GTK+ library

المستخدمة في نظام لينكس ووندوز وماكنتوش QT library

المستخدمة في نظام وندوز Win32/64 GDI

أو يمكن استخدام أداة تطوير متكاملة تستخدم إحدى هذه المكتبات، مثل لازاراس، حيث نجد أنها تستخدم GTK في نظام لينكس، و Win32/64 في نظام وندوز. ولا يحتاج المبرمج أن يدخل في تفاصيل هذه المكتبات حيث أن لازاراس توفر طريقة ذات شفافية عالية في تصميم البرامج الرسومية، فما على المبرمج إلا تصميم هذه البرامج في إحدى بيئات التشغيل ثم إعادة ترجمتها وربطها في بيئات التشغيل الأخرى ليحصل على نفس النسخة من البرنامج الأصلي يعمل في كافة أنظمة التشغيل. لكن أحياناً يحتاج المبرمج أن يُثبت مكتبة الواجهة الرسومية مثل GTK في حال فشل البرنامج ذو الواجهة الرسومية من أن يعمل في حال عدم توفر هذه المكتبة مسبقاً في نظام التشغيل.

البرنامج ذو الواجهة الرسومية الأول

لعمل برنامج جديد ذو واجهة رسومية علينا اختيار :

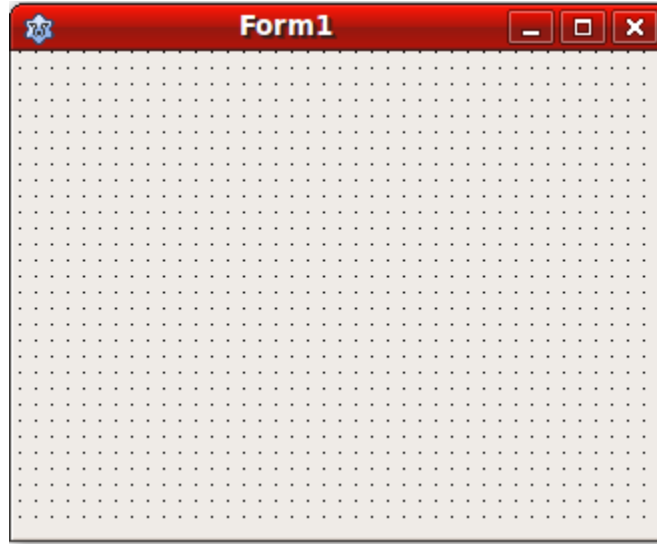
Project/New Project/Application

بعد ذلك نحفظ المشروع بواسطة الخيار:

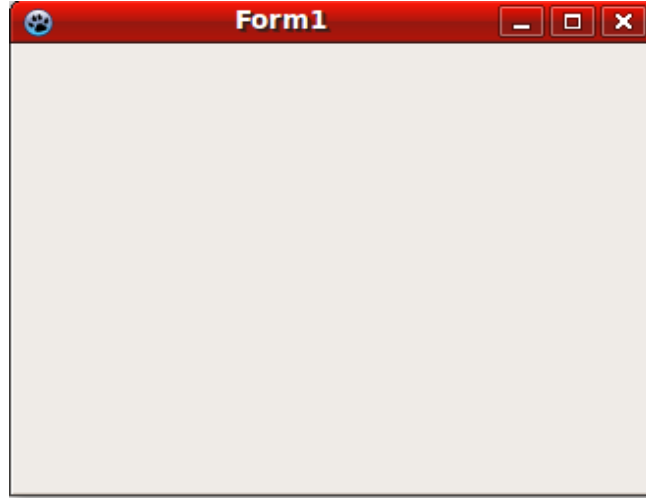
File/Save All

ثم نُنشئ مجلد جديد مثلاً `firstgui` لنحفظ فيه المشروع بكل متعلقاته.
بعد ذلك نحفظ الوحدة الرئيسة في المجلد السابق، ونسميها `main.pas`
ثم نحفظ ملف المشروع في نفس المجلد ونسميه `firstgui.lpi`

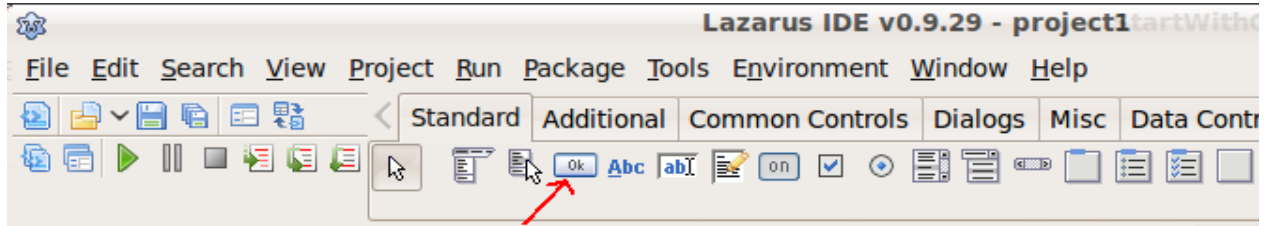
في الوحدة الرئيسة يمكننا الضغط على المفتاح F12 ليظهر لنا الفورم التالي:



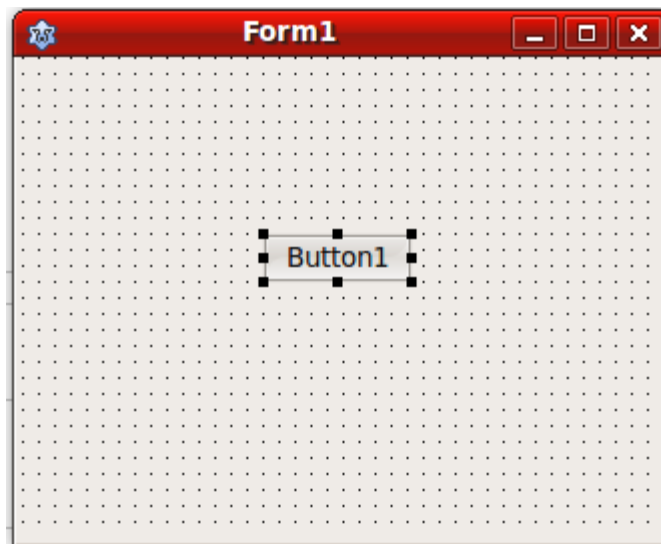
إذا شغلنا البرنامج سوف نحصل على الشكل التالي:



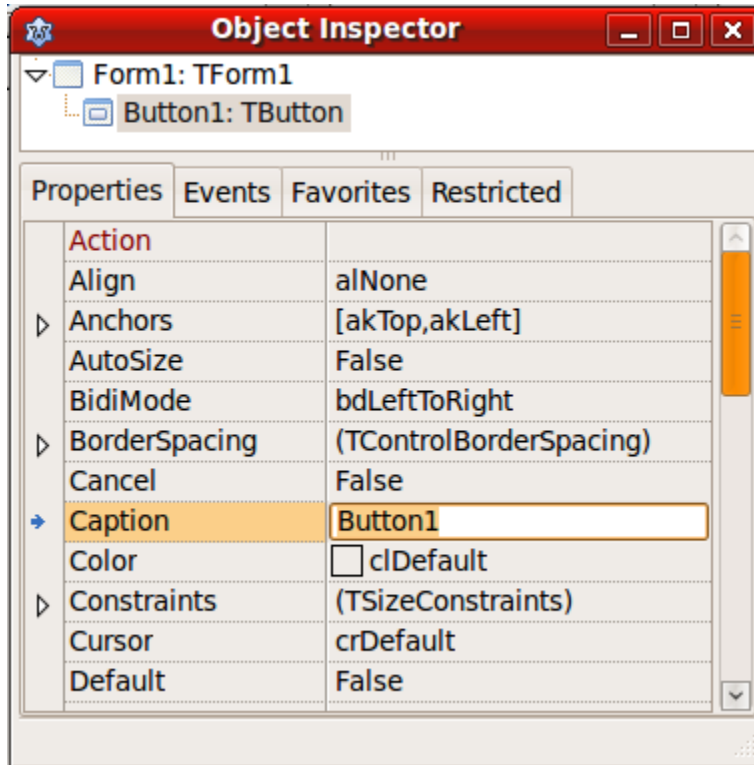
بعد ذلك نُغلق البرنامج ونرجع لحالة التصميم.
ثم نُضيف زر Button من صفحة المكونات التالية:



نضع هذا الزر في وسط الفورم السابق كالتالي:



بعد ذلك نذهب إلى صفحة الخصائص الموجودة في ال Object Inspector. فإذا لم تكن ال object Inspector موجودة يمكن إظهارها عن طريق Window/Object Inspector في القائمة الرئيسية، أو الوقوف على الفورم أو المكون ثم الضغط على المفتاح F11 لتظهر لنا الخصائص التالية:



لابد من التأكد من أننا نؤشر على الزر، ويمكن معرفة ذلك بقراءة القيمة المظلمة في ال Object Inspector والتي يجب أن تحتوي على:

Button1: TButton

في صفحة الخصائص Properties نُغير قيمة ال Caption من Button1 إلى كلمة Click. بعد ذلك نذهب إلى صفحة الأحداث Events في نفس ال Object Inspector ونقر نقرة مزدوجة على قيمة الحدث OnClick فنحصل على الكود التالي:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
end;
```

ثم نكتب السطر التالي بين begin end ثم نشغل البرنامج:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Hello world, this is my first GUI application');
end;
```

بعد تنفيذ البرنامج نضغط على الزر المكتوب عليه Click فنحصل على الرسالة أعلاه.
بهذا نكون قد كتبنا أول برنامج ذو واجهة رسومية.

في الدليل firstgui سوف نجد الملف التنفيذي firstgui إذا كنا في لينكس أو firstgui.exe إذا كنا في بيئة وندوز، وهي ملفات تنفيذية يمكن نقلها وتشغيلها في أي حاسوب آخر به نفس نظام التشغيل.

في البرنامج السابق نريد الوقوف على عدد من النقاط المهمة وهي:

1. البرنامج الرئيس: وهو الملف الذي حفظناه بالإسم: firstgui.lpi، وهو الملف الرئيس الذي يمثل البرنامج. ويمكن فتحه عن طريق:

Project/Source

أو بالضغط على CTRL-F12 ثم اختيار firstgui. وسوف نجد هذا الكود الذي أضفته بيئة لازاراس تلقائياً:

```
program firstgui;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms
  { you can add units after this }, main;

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

ومن النادر أن يُعدل المبرمج هذا الملف، فمعظم تصميم وكتابة الكود تكون في الوحدات الإضافية المصاحبة لهذا المشروع. كما ونلاحظ أن الوحدة الرئيسة `main` قد أُضيفت تلقائياً لهذا الملف.

2. الوحدة الرئيسة **main unit** وهي تحتوي على الفورم الرئيس الذي يظهر عند تشغيل البرنامج والكود المصاحب له. ولعمل برنامج ذو واجهة رسومية لابد من وجود وحدة بهذا الشكل تحتوي على فورم.

التالي هو الكود الذي تحتويه هذه الوحدة بعد إضافة الزر:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics, Dialogs,
  StdCtrls;

type

  { TForm1 }

  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Hello world, this is my first GUI application');
end;

initialization
  {$I main.lrs}
```

end.

نجد أنه في بداية هذه الوحدة عُرف **TForm1** على أنه من النوع **Class** وهو أشبه بالسجل الذي ذكرناه في دروس سابقة. وهو ما يُعرف بالبرمجة الكائنية والتي تعتبر درس متقدم سوف نذكره في الفصل القادم إن شاء الله. وداخل هذا الفورم يوجد زر من النوع **TButton** واسمه **Button1**. يمكن فتح هذه الوحدة بواسطة **CTRL-F12** ثم اختيار اسمها وهي في هذا المثال **main**.

3. **صفحة الخصائص Object Inspector/Properties:** وفي هذه الصفحة تظهر خصائص الكائنات أو المكونات مثل الأزرار و الفورم. فمثلاً نجد أن للزر بعض الخصائص التي يمكن للمبرمج تغييرها وتؤثر مباشرة على شكل الزر أو بعض سلوكه مثل: **Caption, Top, Left, Width, Height**. كذلك الفورم له بعض الخصائص مثل: **Color, Visible** إلخ. وهي كأنها حقول ومتغيرات داخل السجل **Form1** أو **Button1**. وهي في الحقيقة كائنات **Objects** وليس سجلات **Records** إنما شبهناها بالسجلات لتسهيل الفهم.

4. **صفحة الأحداث Object Inspector/Events:** وتحتوي هذه الصفحة على الأحداث التي يمكن استقبالها على الكائن المعين، مثلاً الزر يمكن أن يستقبل بعض الأحداث مثل: **OnClick, OnMouseMove**، أما الفورم فيستقبل أحداث مثل: **OnCreate, OnClose, OnActivate** ومن أسمائها يظهر جلياً متى استدعاء هذا الحدث، فمثلاً الحدث **OnClick** يُستدعى عند الضغط بالماوس على هذا الزر أثناء تشغيل البرنامج. أثناء تطوير البرنامج - في بيئة لآزاراس- وعند النقر المزدوج على هذا الزر تظهر لنا صفحة الكود التي نكتب داخلها ما سوف يُنفذه البرنامج عندما يحدث هذا الحدث، مثلاً الحدث الذي اخترناه عند النقر في الزر هو:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
  ShowMessage('Hello world, this is my first GUI application');  
end;
```

وهذا الكود أو الإجراء يعرف بإجراء الحدث **Event Handler** و يُنفذ هذا الإجراء تلقائياً عند النقر في هذا الزر في حالة التشغيل.

البرنامج الثاني: برنامج إدخال الاسم

نُنشئ برنامج جديد بنفس الطريقة السابقة، ونُنشئ مجلد جديد لهذا المشروع نسميه `inputform` حيث نحفظ فيه الوحدة الرئيسة بإسم `main` والبرنامج بإسم `inputform`. ثم نُضيف مكونات أو كائنات وهي :

2 Lables
Edit box
Button

ثم نُغيّر الخصائص التالية في هذه المكونات:

Form1:
Name: fmMain
Caption: Input form application

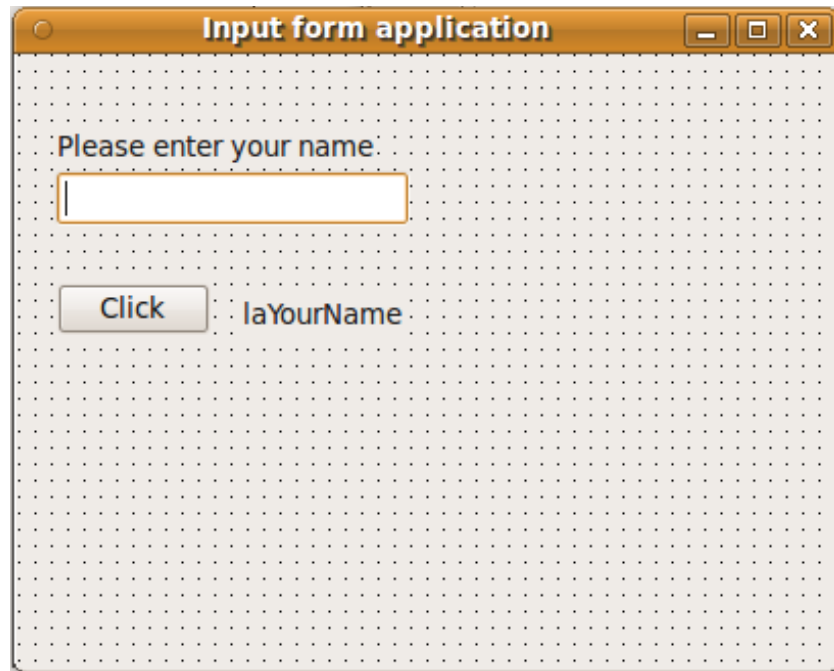
Label1:
Caption: Please enter your name

Label2:
Name: laYourName

Edit1:
Name: edName
Text:

Button1:
Caption: Click

فيصبح لدينا الشكل التالي



في الحدث `OnClick` للزر نضع هذا الكود:

```
procedure TfmMain.Button1Click(Sender: TObject);  
begin  
    laYourName.Caption:= 'Hello ' + edName.Text;  
end;
```

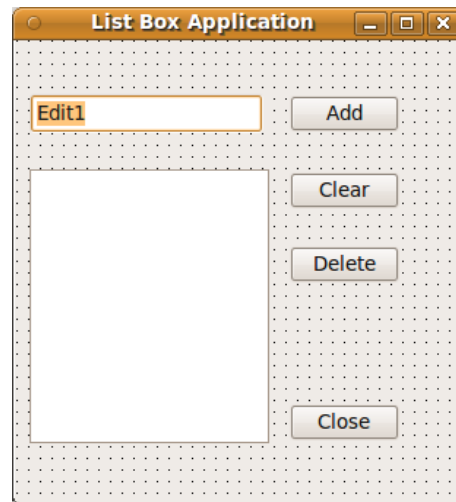
بعد ذلك نُشغل البرنامج ونكتب الاسم في مربع النص، ثم نضغط على الزر `Click` ونشاهد تنفيذ البرنامج.

في البرنامج السابق نلاحظ أننا استخدمنا الحقل `Text` الموجود في الكائن `edName` لقراءة ما أدخله المستخدم من بيانات. وكذلك استخدمنا الحقل `Caption` الموجود في الكائن `laYourName` لكتابة اسم المستخدم.

برنامج الـ ListBox

ننشئ برنامج جديد، ونضع فيه أربع أزرار ومربع نص و `ListBox` كما في الشكل أدناه. و نغير العنوان `Caption` بالنسبة للأزرار كما هو موضح، كذلك نغير أسماء هذه الأزرار إلى:

```
btAdd, btClear, btDelete, btClose
```



الكود المصاحب لهذه الأزرار هو:

```
procedure TForm1.btAddClick(Sender: TObject);
begin
  ListBox1.Items.Add(Edit1.Text);
end;

procedure TForm1.btClearClick(Sender: TObject);
begin
  ListBox1.Clear;
end;

procedure TForm1.btDeleteClick(Sender: TObject);
var
  Index: Integer;
begin
  Index:= ListBox1.ItemIndex;
  if Index <> -1 then
    ListBox1.Items.Delete(Index);
end;

procedure TForm1.btCloseClick(Sender: TObject);
begin
  Close;
end;
```

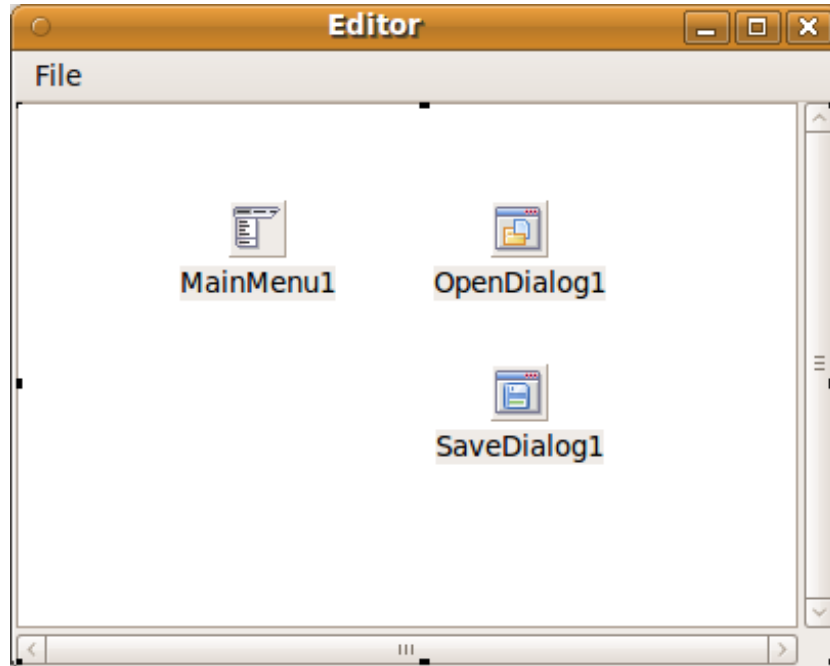
برنامج محرر النصوص Text Editor

نُشِئ برنامج جديد ونضع عليه المكونات التالية:

- قائمة رئيسية TMainMenu
- محرر TMemو ونغير قيمة الاصطفااف Align إلى alClient و التحريك ScrollBars إلى ssBoth
- Dialogs من صفحة TOpenDialog
- TSaveDialog

ثم ننقر نقر مزدوج على القائمة الرئيسية MainMenu1 ثم نُضيف قائمة تُسميها File وتحتها قائمة فرعية SubMenu فيها الخيارات التالية: Open File, Save File, Close

فيكون شكل البرنامج كالتالي:



نكتب الكود التالي:

في الخيار Open File

```
if OpenDialog1.Execute then  
    Memol.Lines.LoadFromFile(OpenDialog1.FileName);
```

وفي Save File

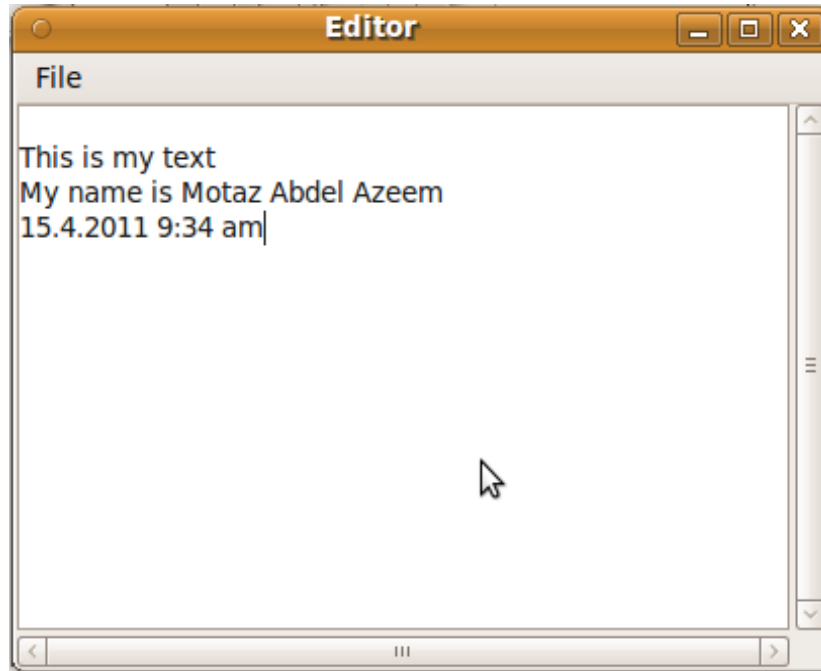
```
if SaveDialog1.Execute then  
    Memo1.Lines.SaveToFile(SaveDialog1.FileName);
```

وفي Close

```
Close;
```

بعد تشغيل البرنامج يمكن كتابة نص ثم حفظه، أو فتح ملف نصي موجود على القرص. مثلاً يمكن فتح ملف ينتهي بالامتداد pas. فهو يعتبر ملف نصي.

وهذا هو شكل البرنامج بعد التنفيذ:



برنامج الأخبار

هذه المرة نريد كتابة برنامج لتسجيل عناوين الأخبار به واجهة رسومية:

- نُنشئ برنامج جديد اسمه gnews
- نُضيف زرّين من نوع TButton
- نُضيف مربع نص TEdit
- نُضيف محرر من نوع TMemo

وخواصهم هي كالتالي:

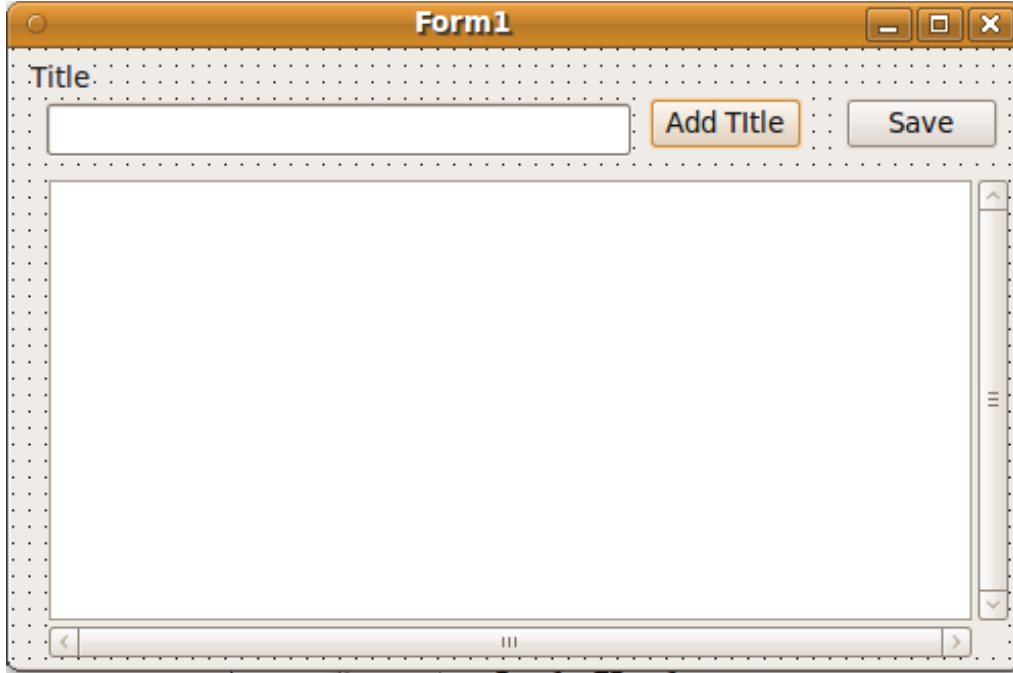
```
Button1
Caption: Add Title

Button2
Caption: Save
Anchors: Left=False, Right=True

Edit1:
Text=

Memo1
ScrollBars: ssBoth
ReadOnly: True
Anchors: Top=True, Left=True, Right=True, Bottom=True
```

ويكون بالشكل التالي:



أما كود Add Title فهو:

```
Memol.Lines.Insert(0,  
    FormatDateTime('yyyy-mm-dd hh:nn', Now) + ': ' + Edit1.Text);
```

وكود Save

```
Memol.Lines.SaveToFile('news.txt');
```

والكود بالنسبة للفورم في حالة إغلاقه: Tform OnClose event

```
Memol.Lines.SaveToFile('news.txt');
```

وكود الفورم في حالة إنشائه: Tform OnCreate event

```
if FileExists('news.txt') then  
    Memol.Lines.LoadFromFile('news.txt');
```

برنامج الفورم الثاني

كما يظهر لنا في معظم البرامج ذات الواجهة الرسومية فهي تحتوي على أكثر من فورم. ولعمل ذلك في بيئة لازاراس نتبع الخطوات التالية:

1. إنشاء برنامج جديد نحفظه في دليل نُسَميه secondform
2. نحفظ الوحدة الرئيسة بإسم main.pas ونُسَمي الفورم الرئيس بإسم fmMain و نسمي المشروع بإسم secondform.lpi
3. نضيف فورم آخر بواسطة File/ New Form ونحفظ الوحدة بإسم second.pas ونسمي الفورم fmSecond
4. فضيف Label نكتب فيه Second Form بخط كبير. يمكن تكبير الخط في خاصية حجم الخط Font.Size الموجودة في ال Label
5. نرجع للفورم الرئيس main ونضع زر جديد فيه.
6. نكتب هذا السطر في كود الوحدة الرئيسة main تحت قسم implementation:

```
uses second;
```

7. في الحدث OnClick بالنسبة للزر الموجود في الفورم الرئيس fmMain نكتب الكود التالي:

```
fmSecond.Show;
```

نُشغل البرنامج ونضغط الزر في الفورم الأول ليظهر لنا الفورم الثاني.

برنامج الهول الهجري

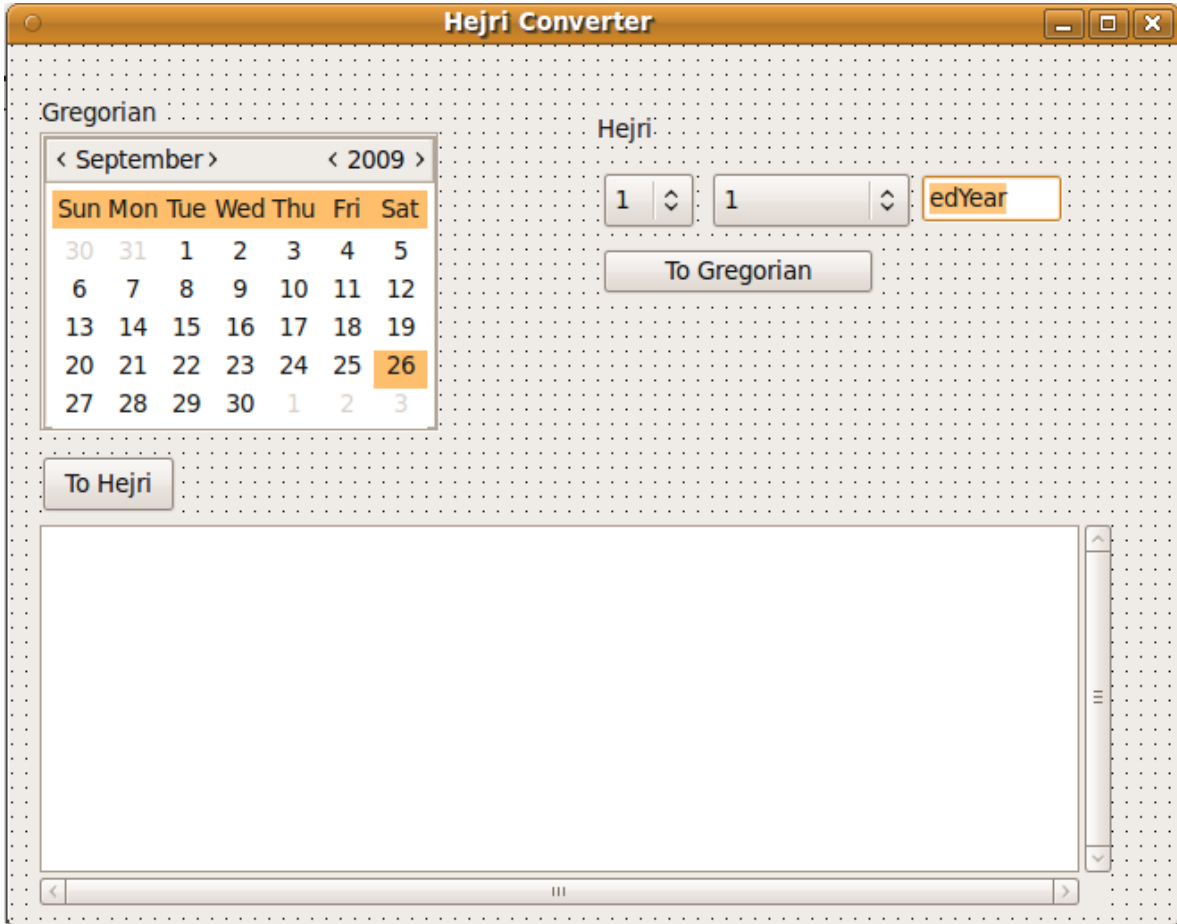
هذا البرنامج يستخدم الوحدة *HejriUtils* الموجودة في الفصل السابق، ويمكن الرجوع إليها في الأمثلة المصاحبة للكتاب.

نُنشئ برنامج جديد ونُدجج الكائنات التالية في الفورم الرئيس:

```
2 TLabel  
TCalnder
```

2 TCompoBox
2 Tbutton
Tmemo
TEdit

ثم نصمم الفورم كالشكل التالي:



وكود الوحدة المصاحبة للفورم الرئيس هو:

```
unit main;  
  
{$mode objfpc}{$H+}  
  
interface  
  
uses  
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics, Dialogs,
```



```
StdCtrls, Calendar, HejriUtils;
```

type

```
{ TfmMain }
```

```
TfmMain = class(TForm)
```

```
  Button1: TButton;
```

```
  Button2: TButton;
```

```
  Calendar1: TCalendar;
```

```
  cbDay: TComboBox;
```

```
  cbMonth: TComboBox;
```

```
  edYear: TEdit;
```

```
  Label1: TLabel;
```

```
  Label2: TLabel;
```

```
  Memo1: TMemo;
```

```
  procedure Button1Click(Sender: TObject);
```

```
  procedure Button2Click(Sender: TObject);
```

```
  procedure FormCreate(Sender: TObject);
```

private

```
  { private declarations }
```

public

```
  { public declarations }
```

```
end;
```

var

```
  fmMain: TfmMain;
```

implementation

```
{ TfmMain }
```

```
procedure TfmMain.Button1Click(Sender: TObject);
```

var

```
  Year, Month, Day: Word;
```

begin

```
  DateToHejri(Calendar1.DateTime, Year, Month, Day);
```

```
  Memo1.Lines.Add(DateToStr(Calendar1.DateTime) + ' = ' +
```

```
    Format('%d.%s.%d', [Day, HejriMonthsAr[Month], Year]));
```

```
  edYear.Text := IntToStr(Year);
```

```
  cbMonth.ItemIndex := Month - 1;
```

```
  cbDay.ItemIndex := Day - 1;
```

```
end;
```

```
procedure TfmMain.Button2Click(Sender: TObject);
```

var

```
  ADate: TDateTime;
```

```
  CYear, CMonth, CDay: Word;
```

```
  YearD, MonthD, DayD: Word;
```

begin

```
  ADate := HejriToDate(StrToInt(edYear.Text), cbMonth.ItemIndex + 1,
```

```
    cbDay.ItemIndex + 1);
    Memol.Lines.Add(FormatDateTime('yyyy-mm-dd ddd', ADate));

    DateToHejri(Now, CYear, CMonth, CDay);
    HejriDifference(StrToInt(edYear.Text), cbMonth.ItemIndex + 1,
    cbDay.ItemIndex + 1, CYear, CMonth,
    CDay, YearD, MonthD, DayD);
    Memol.Lines.Add('Difference = ' +
    Format('%d year, %d month, %d day ', [YearD, MonthD, DayD]));
    Calendar1.DateTime:= ADate;

end;

procedure TfmMain.FormCreate(Sender: TObject);
var
    i: Integer;
    Year, Month, Day: Word;
begin
    Calendar1.DateTime:= Now;
    cbMonth.Clear;
    for i:= 1 to 12 do
        cbMonth.Items.Add(HejriMonthsAr[i]);
    DateToHejri(Now, Year, Month, Day);
    cbDay.ItemIndex:= Day - 1;
    cbMonth.ItemIndex:= Month - 1;
    edYear.Text:= IntToStr(Year);

end;

initialization
    {$I main.lrs}

end.
```

الفصل الرابع

البرمجة الكائنية

Object Oriented Programming

مقدمة

ترتكز فكرة البرمجة الكائنية على أنها تُقسّم البرنامج ووحداته ومكوناته إلى كائنات. فالكائن **Object** هو عبارة عن:

1. مجموعة خصائص **Properties** التي يمكن أن نعتبرها متغيرات تمثل حالة الكائن.
2. إجراءات ودوال تسمى **Methods** عند تنفيذها يُمكن أن تؤثر على خصائص الكائن أو يتأثر التنفيذ بناءً على قيم هذه الخصائص.
3. أحداث **Events** تحدث له مثل نقرة بالماوس أو إظهار له في الشاشة وغيرها.
4. إجراءات الأحداث **Event Handlers** مثل الإجراء الذي يُنفذ عند نقرة الماوس.

وتتكامل هذه الخصائص والإجراءات لتكوّن كائن. وتؤثر هذه الإجراءات مباشرة على الخصائص.

بيانات + كود = كائن

وكمثال لكائن هو ما استخدمناه في الفصل السابق من برمجة الواجهة الرسومية، فهو يعكس الاستخدام اليومي للمبرمج للبرمجة الكائنية المنحى. فنجد أن الفورم هو عبارة عن كائن به بعض الخصائص مثل ال **Caption, Width, Height** وبه بعض الإجراءات التي تؤثر عليه تأثير مباشر مثل **Close, Show, Hide, ShowModal** وغيرها. كذلك فإن له أحداث مثل **OnClick, OnCreate, OnClose** وله إجراءات مصاحبة لهذه الأحداث، وهي في هذه الحالة الكود الذي يكتبه المبرمج يُنفذ استجابة لحدث معين.

المثال الأول، برنامج التاريخ والوقت

نفرض أننا نريد أن نكتب كود لكائن يحتوي على تاريخ ووقت، وبعض الإجراءات التي تخص التاريخ والوقت.

ننشئ برنامج جديد، ثم وحدة جديدة نسميها **DateTimeUnit** وفيها كائن يسمى **TmyDateTime** وهذا هو الكود الذي نكتبه في هذه الوحدة:

```
unit DateTimeUnit;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type
  { TMyDateTime }

  TMyDateTime = class
  private
    fDateTime: TDateTime;
  public
    function GetDateTime: TDateTime;
    procedure SetDateTime(ADateTime: TDateTime);
    procedure AddDays(Days: Integer);
    procedure AddHours(Hours: Single);
    function GetDateTimeAsString: string;
    function GetTimeAsString: string;
    function GetDateAsString: string;
    constructor Create(ADateTime: TDateTime);
    destructor Destroy; override;
  end;

implementation

{ TMyDateTime }

function TMyDateTime.GetDateTime: TDateTime;
begin
  Result:= fDateTime;
end;

procedure TMyDateTime.SetDateTime(ADateTime: TDateTime);
begin
  fDateTime:= ADateTime;
end;

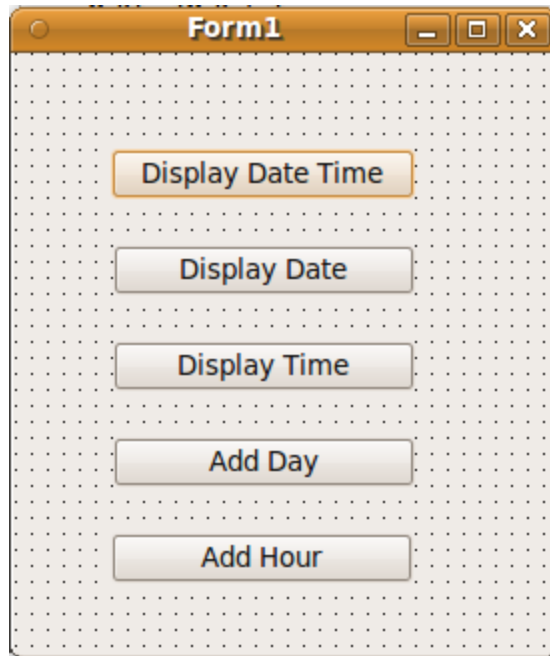
procedure TMyDateTime.AddDays(Days: Integer);
begin
  fDateTime:= fDateTime + Days;
end;

procedure TMyDateTime.AddHours(Hours: Single);
begin
  fDateTime:= fDateTime + Hours / 24;
end;

function TMyDateTime.GetDateTimeAsString: string;
begin
```

```
    Result:= DateTimeToStr(fDateTime);  
end;  
  
function TMyDateTime.GetTimeAsString: string;  
begin  
    Result:= TimeToStr(fDateTime);  
end;  
  
function TMyDateTime.GetDateAsString: string;  
begin  
    Result:= DateToStr(fDateTime);  
end;  
  
constructor TMyDateTime.Create(ADateTime: TDateTime);  
begin  
    fDateTime:= ADateTime;  
end;  
  
destructor TMyDateTime.Destroy;  
begin  
    inherited Destroy;  
end;  
  
end.
```

وفي الوحدة الرئيسة للبرنامج main أضفنا 5 أزرار كما في الشكل التالي:



ثم كتبنا الكود التالي في الأزرار:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics, Dialogs,
  StdCtrls, DateTimeUnit;

type

  { TForm1 }

TForm1 = class(TForm)
  Button1: TButton;
  Button2: TButton;
  Button3: TButton;
  Button4: TButton;
  Button5: TButton;
  procedure Button1Click(Sender: TObject);
  procedure Button2Click(Sender: TObject);
  procedure Button3Click(Sender: TObject);
  procedure Button4Click(Sender: TObject);
  procedure Button5Click(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
  { private declarations }
public
  MyDT: TMyDateTime;
  { public declarations }
end;

var
  Form1: TForm1;

implementation

{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);
begin
  MyDT := TMyDateTime.Create(Now);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(MyDT.GetDateTimeAsString);
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```
begin
  ShowMessage(MyDT.GetDateAsString);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  ShowMessage(MyDT.GetTimeAsString);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  MyDT.AddHours(1);
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
  MyDT.AddDays(1);
end;

initialization
  {$I main.lrs}

end.
```

نلاحظ في هذا البرنامج النقاط التالية:

في الوحدة DateTimeUnit

1. عرّفنا النوع **TmyDateTime** على أنه من النوع **Class** وهذه هي طريقة تعريف كائن والذي يمكن أن يحتوي على إجراءات، دوال ومتغيرات كما في هذا المثال
2. عرّفنا إجراء اسمه **Create** ونوعه هو: **Constructor** ، وهو إجراء من نوع خاص يستخدم لإنشاء وتفعيل وتهيئة متغيرات كائن ما في الذاكرة حتى يمكن استخدامه.
3. أضفنا إجراء اسمه **Destroy** و نوعه **Destructor** ، وهو إجراء من نوع خاص يُنفذ عند الانتهاء من العمل بهذا الكائن وإرادة حذفه من الذاكرة. وفي نهايته كلمة **override** سوف نتطرق لها في كتاب قادم إن شاء الله.
4. نلاحظ أن هناك جزأين، الجزء الأول هو **private** ، ومتغيراته و دواله وإجراءاته لا يمكن الوصول لها من وحدة أخرى عند استخدام هذا الكائن.

5. الجزء الثاني هو **public** وهو ذو متغيرات، دوال وإجراءات يمكن الوصول لها من خارج وحدة هذا الكائن، وهي تمثل الواجهة المرئية لهذا الكائن مثل Interfaces بالنسبة للوحدات.

أما في الوحدة الرئيسة للبرنامج main فنجد الآتي:

1. أضفنا اسم الوحدة DateTimeUnit مع باقي الوحدات في الجزء Uses

2. عرّفنا الكائن MyDT داخل كائن الفورم:

```
MyDT: TMyDateTime;
```

3. أنشأنا وهياً الكائن عند الحدث OnFormCreate:

```
MyDT := TMyDateTime.Create(Now);
```

وهذه هي طريقة إنشاء وتهيئة الكائن في لغة أوبجكت باسكال.

الكبسلة Encapsulation

من فوائد البرمجة الكائنية هي الكبسلة، وهي أن نجمع البيانات والإجراءات ونحميها من العالم الخارجي لتصبح في شكل كبسولة محمية وتُمثل قطعة واحدة، ولا يُسمح بالوصول إلى ما بداخلها إلا بواسطة بوابات معينة. وهذه البوابات تتمثل في الإجراءات المتاحة، حيث يمكن إضافة إجراء متاح للمستخدم **public** وإجراءات مخفية **private** أو محمية **protected** لا يستطيع المستخدم (المبرمج) الوصول إليها. كذلك فإن البيانات تكون محمية. مثلاً هذه البيانات يُمكن أن تكون مصفوفة رقمية أو مقطعية بها بيانات تُستخدم بواسطة إجراءات الكائن، فلا نُريد للمستخدم أن يتسبب في تلف البيانات أو حذفها بالوصول إلى هذه المصفوفة مباشرة. بدلاً عن ذلك نوفر إجراءات مثل إجراء لإضافة عناصر لتلك المصفوفة، أو حذف عنصر بالطريقة التي تُريدها ولا تؤثر على البيانات سلباً.

يمكن تشبيه عملية الكبسلة بجهاز موبايل يحتوي على بطارية ودوائر إلكترونية وشريحة، فيغلفها الصانع، فلا يُسمح للمستخدم مثلاً أن يصل إلى أسلاك البطارية لمحاولة شحنها أو تفريغها، بدلاً عن ذلك يوفر الصانع منفذ لشحن الموبايل، وإذا اكتملت عملية الشحن فإن هذا المنفذ أو الدوائر التي خلفه تُوقف عملية الشحن حتى لا تتلف البطارية. كذلك فإن الموبايل لا يسمح بتفريغ البطارية، إلا عن طريق استهلاكها في المكالمات أو الاستخدام العادي. كذلك فإن الدوائر الإلكترونية مخفية ولا تظهر، ولا يُمكن

أن يصل المستخدم إليها مباشرة ولا يستطيع توصيل أسلاك أو كهرباء مباشرة لها. بهذه الطريقة تضمن لنا الكبسلة حماية المحتويات الداخلة للموبايل أو البيانات بالنسبة للكائن وتضمن لنا الاستخدام الأمثل الذي يُريده الصانع لهذا الكائن أو الجهاز الإلكتروني.

برنامج الأخبار بطريقة كائنية

في المثال التالي أعدنا كتابة برنامج الأخبار بطريقة كائنية. وفي هذه المرة صنفنا الأخبار، فكل صف يُسجل في ملف بيانات منفصل من نوع ملفات الوصول المباشر. أنشأنا برنامج جديد ذو واجهة رسومية وأسميناه **oonews**، ثم أضفنا وحدة جديدة كتبنا فيها كود الكائن **TNews** كالآتي:

```
unit news;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type
  TNewsRec = record
    ATime: TDateTime;
    Title: string[100];
  end;

  { TNews }

  TNews = class
  private
    F: file of TNewsRec;
    fFileName: string;
  public
    constructor Create(FileName: string);
    destructor Destroy; override;
    procedure Add(ATitle: string);
    procedure ReadAll(var NewsList: TStringList);
    function Find(Keyword: string;
      var ResultList: TStringList): Boolean;
  end;

implementation

{ TNews }

constructor TNews.Create(FileName: string);
begin
  fFileName:= FileName;
end;

destructor TNews.Destroy;
begin
```

```
inherited Destroy;
end;

procedure TNews.Add(ATitle: string);
var
  Rec: TNewsRec;
begin
  AssignFile(F, fFileName);
  if FileExists(fFileName) then
    begin
      FileMode:= 2; // Read/write access
      Reset(F);
      Seek(F, FileSize(F));
    end

    else
      Rewrite(F);

  Rec.ATime:= Now;
  Rec.Title:= ATitle;
  Write(F, Rec);
  CloseFile(F);

end;

procedure TNews.ReadAll(var NewsList: TStringList);
var
  Rec: TNewsRec;
begin
  NewsList.Clear;
  AssignFile(F, fFileName);
  if FileExists(fFileName) then
    begin
      Reset(F);
      while not Eof(F) do
        begin
          Read(F, Rec);
          NewsList.Add(DateTimeToStr(Rec.ATime) + ' : ' + Rec.Title);
        end;
      CloseFile(F);
    end;

end;

function TNews.Find(Keyword: string; var ResultList: TStringList): Boolean;
var
  Rec: TNewsRec;
begin
  ResultList.Clear;
  Result:= False;
  AssignFile(F, fFileName);
  if FileExists(fFileName) then
    begin
      Reset(F);
```

```
while not Eof(F) do
begin
  Read(F, Rec);
  if Pos(LowerCase(Keyword), LowerCase(Rec.Title)) > 0 then
  begin
    ResultList.Add(DateTimeToStr(Rec.ATime) + ' : ' + Rec.Title);
    Result:= True;
  end;
end;
CloseFile(F);
end;

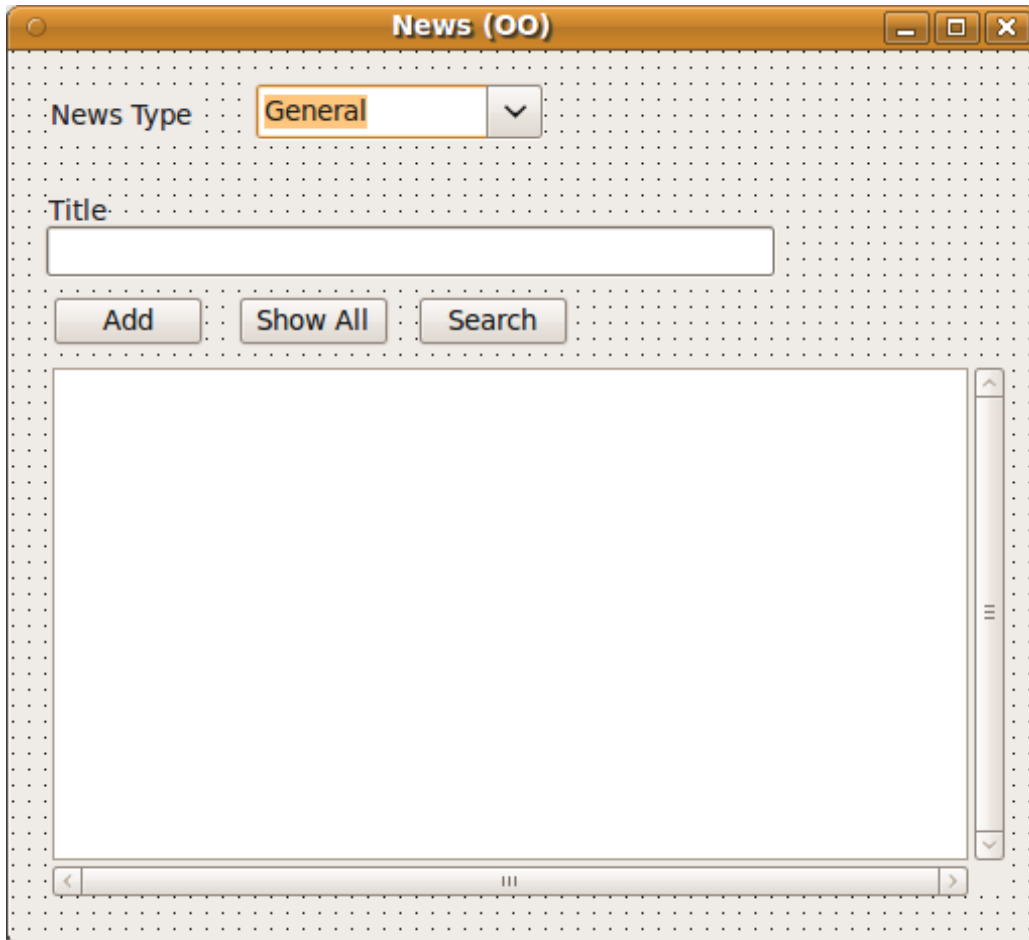
end;

end.
```

وفي البرنامج الرئيس أضفنا هذه المكونات:

Edit Box, ComboBox, 3 Buttons, Memo, 2 labels

فأصبح شكل الفورم كالاتي:



ثم كتبنا الكود التالي للوحدة المصاحبة للفورم الرئيس:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,
  Dialogs, News, StdCtrls;

type

  { TForm1 }

  TForm1 = class(TForm)
    btAdd: TButton;
    btShowAll: TButton;
    btSearch: TButton;
    cbType: TComboBox;
    edTitle: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Memo1: TMemo;
    procedure btAddClick(Sender: TObject);
    procedure btSearchClick(Sender: TObject);
    procedure btShowAllClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    NewsObj: array of TNews;
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  SetLength(NewsObj, cbType.Items.Count);
  for i:= 0 to High(NewsObj) do
    NewsObj[i]:= TNews.Create(cbType.Items[i] + '.news');
end;
```

```
procedure TForm1.btAddClick(Sender: TObject);
begin
  NewsObj[cbType.ItemIndex].Add(edTitle.Text);
end;

procedure TForm1.btSearchClick(Sender: TObject);
var
  SearchStr: string;
  ResultList: TStringList;
begin
  ResultList:= TStringList.Create;
  if InputQuery('Search News', 'Please input keyword', SearchStr) then
    if NewsObj[cbType.ItemIndex].Find(SearchStr, ResultList) then
      begin
        Memol.Lines.Clear;
        Memol.Lines.Add(cbType.Text + ' News');
        Memol.Lines.Add('-----');
        Memol.Lines.Add(ResultList.Text);
      end
    else
      Memol.Lines.Text:= SearchStr + ' not found in ' +
        cbType.Text + ' news';
      ResultList.Free;
end;

procedure TForm1.btShowAllClick(Sender: TObject);
var
  List: TStringList;
begin
  List:= TStringList.Create;
  NewsObj[cbType.ItemIndex].ReadAll(List);
  Memol.Lines.Clear;
  Memol.Lines.Add(cbType.Text + ' News');
  Memol.Lines.Add('-----');
  Memol.Lines.Add(List.Text);
  List.Free;
end;

procedure TForm1.FormClose(Sender: TObject; var CloseAction: TCloseAction);
var
  i: Integer;
begin
  for i:= 0 to High(NewsObj) do
    NewsObj[i].Free;

  NewsObj:= nil;
end;

initialization
  {$I main.lrs}

end.
```

نلاحظ في البرنامج السابق الآتي:

1. أننا استخدمنا نوع جديد من تعريف المصفوفات وهو ما يُعرف بالمصفوفة المرنة **Dynamic Array**، وهي مصفوفة غير محددة الطول في وقت كتابة الكود، لكن طولها يتغير زيادة أو نقصاناً أثناء تشغيل البرنامج حسب الحاجة، وطريقة التعريف هي كالآتي:

```
NewsObj: array of TNews;
```

وفي أثناء تنفيذ البرنامج وقبل استخدامها يجب أن نحجز مساحة لها في الذاكرة باستخدام الإجراء **SetLength**، مثلاً إذا كتبنا الكود التالي:

```
SetLength(NewsObj, 10);
```

فهذا يعني أننا حجزنا عشر خانات، فهي تماثل في هذه الحالة هذا التعريف:

```
NewsObj: array [0 .. 9] of Tnews;
```

وفي حالة برنامج الأخبار حجزنا خانات تمثل عدد أنواع الأخبار الموجودة في ال Combo Box:

```
SetLength(NewsObj, cbType.Items.Count);
```

وبهذه الطريقة تجعل عدد الكائنات معتمدة على عدد أنواع الأخبار المكتوبة في الكائن **ComboBox.Items** فكلما أضاف إليها المبرمج، كلما زادت تلقائياً دون الحاجة لإعادة تغيير الطول.

2. نوع الكائن **TNews** هو عبارة عن **Class** وهي تمثل النوع، ولا يمكن استخدامه مباشرة إلا بتعريف متغيرات منه تسمى كائنات **Objects**. بنفس الطريقة التي نعتبر فيها أن النوع الصحيح **Integer** لا يمكن استخدامه مباشرة بل يجب استخدام متغيرات من نوعه، مثل **x, j, i**.

3. في نهاية البرنامج حررنا الكائنات في المصفوفة من الذاكرة أولاً ثم حررنا وحذفنا المصفوفة المرنة ثانياً بواسطة الكود التالي:

```
for i:= 0 to High(NewsObj) do  
  NewsObj[i].Free;  
  
NewsObj:= nil;
```


برنامج الصفوف

الصفوف هي إحدى دروس هيكلية البيانات Data structure، وهي تستخدم كحلول لمعالجة الصف أو الطابور. ويتميز الصف بأن من يدخله أولاً يخرج أولاً إذا لم تكن هناك أولوية. في هذه الحال تكون الأولوية هي زمن الدخول في الصف.

في البرنامج التالي كتبنا وحدة أسميناها **Queue** تحتوي على نوع كائن جديد أسميناه **TQueue** يمكن استخدامه لإضافة بيانات (مثلاً أسماء) إلى صف ثم استخراج هذه البيانات من الصف. والصف يختلف عن المصفوفة في أن القراءة (الاستخراج) منه تؤثر على البيانات وطول الصف، فمثلاً إذا كان هناك صف يحتوي على 10 عناصر، ثم قرأنا ثلاث عناصر منها فإن عدد العناصر في الصف ينقص إلى سبعة. وإذا قرأنا العناصر جميعها، يُصبح الصف فارغاً.

كود وحدة الصف:

```
unit queue;
// This unit contains TQueue class,
// which is suitable for any string queue

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type

  { TQueue }

  TQueue = class
  private
    fArray: array of string;
    fTop: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    function Put(AValue: string): Integer;
    function Get(var AValue: string): Boolean;
    function Count: Integer;
    function ReOrganize: Boolean;
  end;

implementation

{ TQueue }

constructor TQueue.create;
```

```
begin
  fTop:= 0;
end;

destructor TQueue.destroy;
begin
  SetLength(fArray, 0); // Erase queue array from memory
  inherited destroy;
end;

function TQueue.Put(AValue: string): Integer;
begin
  if fTop >= 100 then
    ReOrganize;

  SetLength(fArray, Length(fArray) + 1);
  fArray[High(fArray)]:= AValue;
  Result:= High(fArray) - fTop;
end;

function TQueue.Get(var AValue: string): Boolean;
begin
  AValue:= '';
  if fTop <= High(fArray) then
    begin
      AValue:= fArray[fTop];
      Inc(fTop);
      Result:= True;
    end
  else // empty
    begin
      Result:= False;

      // Erase array
      SetLength(fArray, 0);
      fTop:= 0;
    end;
end;

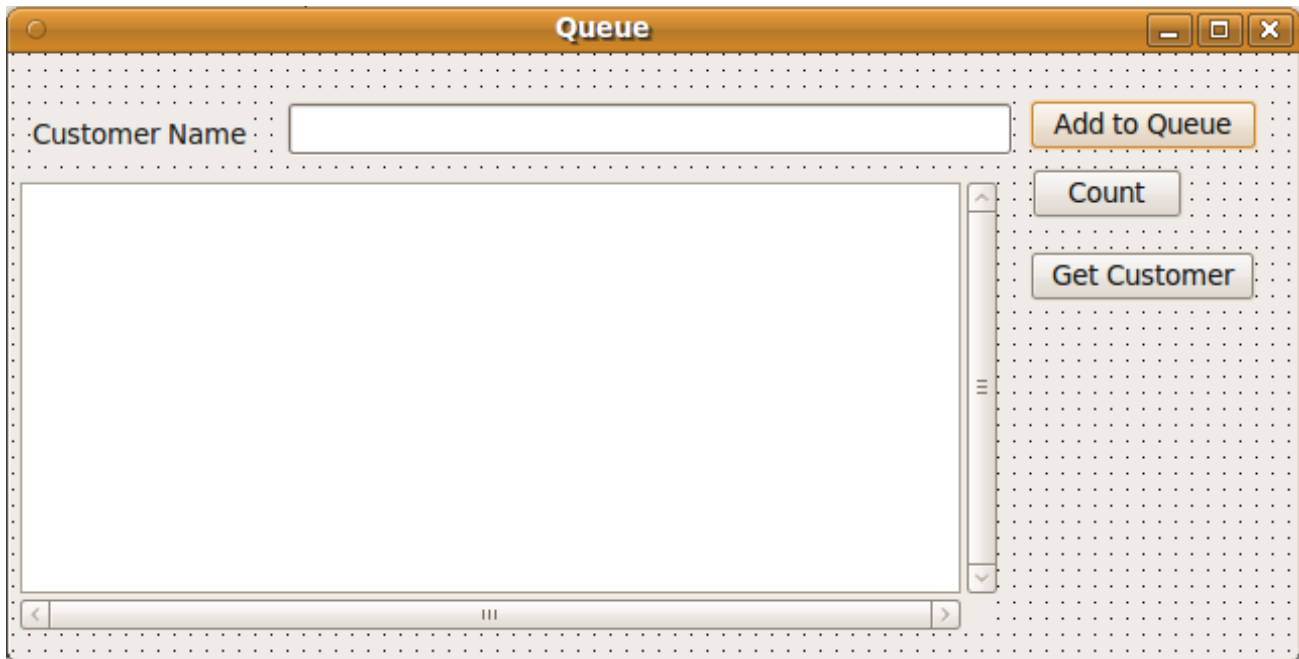
function TQueue.Count: Integer;
begin
  Result:= Length(fArray) - fTop;
end;

function TQueue.ReOrganize: Boolean;
var
  i: Integer;
  PCount: Integer;
begin
  if fTop > 0 then
    begin
      PCount:= Count;
      for i:= fTop to High(fArray) do
        fArray[i - fTop]:= fArray[i];
    end;
  end;
end;
```

```
// Truncate unused data
setLength(fArray, PCount);
fTop:= 0;
Result:= True; // Re Organize is done

end
else
  Result:= False; // nothing done
end;
end;
end.
```

الفورم الرئيس لبرنامج الصف:



كود الوحدة الرئيسة للفورم:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,
  Dialogs, Queue, StdCtrls;

type
```

```
{ TfmMain }

TfmMain = class(TForm)
  bbAdd: TButton;
  bbCount: TButton;
  bbGet: TButton;
  edCustomer: TEdit;
  Label1: TLabel;
  Memol: TMemo;
  procedure bbAddClick(Sender: TObject);
  procedure bbCountClick(Sender: TObject);
  procedure bbGetClick(Sender: TObject);
  procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);
  procedure FormCreate(Sender: TObject);
private
  { private declarations }
public
  MyQueue: TQueue;
  { public declarations }
end;

var
  fmMain: TfmMain;

implementation

{ TfmMain }

procedure TfmMain.FormCreate(Sender: TObject);
begin
  MyQueue := TQueue.Create;
end;

procedure TfmMain.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
  MyQueue.Free;
end;

procedure TfmMain.bbCountClick(Sender: TObject);
begin
  Memol.Lines.Add('Queue length is: ' + IntToStr(MyQueue.Count));
end;

procedure TfmMain.bbAddClick(Sender: TObject);
var
  APosition: Integer;
begin
  APosition := MyQueue.Put(edCustomer.Text);
  Memol.Lines.Add(edCustomer.Text + ' has been added as # ' +
    IntToStr(APosition + 1));
end;
```

```
procedure TfmMain.bbGetClick(Sender: TObject);
var
  ACustomer: string;
begin
  if MyQueue.Get(ACustomer) then
  begin
    Memol.Lines.Add('Got: ' + ACustomer + ' from the queue');
  end
  else
    Memol.Lines.Add('Queue is empty');
end;

initialization
  {$I main.lrs}

end.
```

في البرنامج السابق في الإجراء Put يزيد سعة المصفوفة المرنة لاستيعاب العنصر الجديد. وعند استخراج عنصر عن طريق الدالة Get تُحرك المؤشر fTop الذي يُؤشر على أول العناصر بالصف. وبعد هذا الإجراء لا يمكن حذف المصفوفة من البداية، حيث أن المصفوفة المرنة فقط يمكن تقليصها أو توسيعها من النهاية، لذلك اكتفينا فقط بتحريك المؤشر مع بقاء العناصر القديمة تحتل مكاناً في المصفوفة. ولمعالجة هذه المشكلة كتبنا الإجراء ReOrganize لتفريغ الصف من العناصر غير المستخدمة والتي أُستخرجت من قبل حتى لا تحتل ذاكرة بدون فائدة. الإجراء ReOrganize ببساطة يعمل كلما يصل عدد المستخرجين من الصف حوالي مائة، حيث يُنادى هذا الإجراء من الدالة Put. ويمكن زيادة هذا الرقم إلى ألف مثلاً حتى لا يتسبب هذا الإجراء في إبطاء الإضافة، فقط يعمل كلما يصل طول الصف الفعلي في الذاكرة إلى ألف. في هذا الإجراء ننقل العناصر الموجودة في الصف إلى مكان العناصر القديمة المستخرجة مسبقاً، بهذه الطريقة:

```
for i:= fTop to High(fArray) do
  fArray[i - fTop]:= fArray[i];
```

ثم نقض المصفوفة عند آخر عنصر داخل الصف ونضع مؤشر الصف في بداية المصفوفة كالتالي:

```
// Truncate unused data
setLength(fArray, PCount);
fTop:= 0;
```

من البرنامج السابق نجد أن البرمجة الكائنية وفرت لنا حماية البيانات **Information hiding** بواسطة الكبسلة كما تكلمنا عنها سابقاً. حيث أن البيانات الحساسة التي ربما تسبب في سلوك غير معلوم إذا أتحتنا للمستخدم الوصول المباشر لها. لذلك أخفيها هذه المتغيرات المهمة والتي لا يحتاج ولا يفترض أن يصل المستخدم لها مباشرة وهي:

```
private  
fArray: array of string;  
fTop: Integer;
```

حيث أن المبرمج الذي يستخدم هذا الكائن لا يستطيع الوصول لها من خلال برنامج الرئيس ولو وصل لها لتسبب في ضياع البيانات أو حدوث أخطاء. مثلاً نفرض أنه غير قيمة fTop إلى 1000 في حين وجود فقط 10 عناصر في الصف، فهذا يتسبب بخطأ أثناء التشغيل. وكبديل سمحنا له بالوصول لإجراءات ودوال آمنة طبيعية تتناسب وطبيعة تحقيق الهدف، مثل Put, Get. فمهما استخدمها المبرمج لا نتوقع منها حدوث خطأ أو شيء غير منطقي. وهذه الطريقة أشبه باستخدام بوابات معروفة للوصول للبيانات. وفي هذه الحالة البوابات هي الإجراءات و الدوال الموجودة في هذا الكائن، حيث لا يمكن استخدام إجراءات أخرى غير الموجودة في هذا الكائن للوصول للبيانات.

ومما سبق نجد أن الكائن بالنسبة للمستخدم (المبرمج الذي يستخدمه في برامجه) هو عبارة عن إجراءات ودوال تقبع خلفها بيانات تخص هذا الكائن.

الملف الكائني Object Oriented File

رأينا في الفصل الأول كيفية التعامل مع الملفات بأنواعها المختلفة. وقد كانت طريقة التعامل معتمدة على نوع البرمجة الهيكلية، أي أنها تتعامل مع دوال وإجراءات ومتغيرات فقط، أما هذه المرة نريد الوصول للملفات عن طريق الكائن الموجود في مكتبات فري باسكال و لازاراس وهو *TFileStream*. والكائن *TFileStream* يتعامل مع الملفات بطريقة أشبه بنوع الملف غير محدد النوع Untyped File، والذي بدوره يصلح لكافة أنواع الملفات. تتميز طريقة استخدام الملف الكائني بأنها تحتوي على إجراءات ودوال وخصائص غنيّة تمتاز بكل ما تمتاز به البرمجة الكائنية من سهولة الاستخدام والمنطقية في التعامل والقياسية وتوقع المبرمج لطريقة الاستخدام.

برنامج نسخ الملفات بواسطة TFileStream

في هذا المثال نريد نسخ ملف باستخدام هذا النوع للوصول للملفات، فنُنشئ برنامج جديد ذو واجهة رسومية ثم نُدرج عليه هذه المكونات:

TButton, TOpenDialog, TSaveDialog

ثم نكتب الكود التالي في الحدث OnClick بالنسبة للزر:

```
procedure TfmMain.Button1Click(Sender: TObject);
var
  SourceF, DestF: TFileStream;
  Buf: array [0 .. 1023] of Byte;
  NumRead: Integer;
begin
  if OpenDialog1.Execute and SaveDialog1.Execute then
  begin
    SourceF:= TFileStream.Create(OpenDialog1.FileName, fmOpenRead);
    DestF:= TFileStream.Create(SaveDialog1.FileName, fmCreate);
    while SourceF.Position < SourceF.Size do
    begin
      NumRead:= SourceF.Read(Buf, SizeOf(Buf));
      DestF.Write(Buf, NumRead);
    end;
    SourceF.Free;
    DestF.Free;
    ShowMessage('Copy finished');
  end;
end;
```

وهي طريقة مشابهة لطريقة نسخ الملفات باستخدام الملفات غير محددة النوع التي استخدمناها في الفصل الأول.

ويمكن أيضاً نسخ الملف بطريقة مبسطة وهي كالآتي:

```
procedure TfmMain.Button1Click(Sender: TObject);
var
  SourceF, DestF: TFileStream;
begin
  if OpenFileDialog1.Execute and SaveDialog1.Execute then
  begin
    SourceF:= TFileStream.Create(OpenDialog1.FileName, fmOpenRead);
    DestF:= TFileStream.Create(SaveDialog1.FileName, fmCreate);
    DestF.CopyFrom(SourceF, SourceF.Size);
    SourceF.Free;
    DestF.Free;
    ShowMessage('Copy finished');
  end;
end;
```

حيث أن الإجراء *CopyFrom* ينسخ محتويات الملف كاملاً لأننا حددنا حجم الجزء المراد نسخه وهو *SourceF.Size* الذي يمثل الحجم الكامل للملف بالبايت.

الوراثة Inheritance

الوراثة في البرمجة الكائنية تعني إنشاء كائن جديد من كائن موجود مسبقاً. وهو يعني أن الكائن الجديد يرث صفات الكائن القديم ويمكن أن يزيد عليه بعض الخصائص و الإجراءات و الدوال.

كمثال للوراثة نريد عمل كائن لصف من النوع الصحيح، فبدلاً من كتابته من الصفر يمكن الاعتماد على كائن الصف الذي كتبناه سابقاً والذي يستخدم المقاطع. ولوراثة كائن ما نُنشئ وحدة جديدة ثم نستخدم الوحدة القديمة المحتوية على الكائن القديم. و تُعرّف الكائن الجديد كالآتي:

```
TIntQueue = class(TQueue)
```

وقد أسمينا الوحدة الجديدة *IntQueue*. ثم أضفنا فقط دالتين في الكائن الجديد *PutInt, GetInt* وكود الوحدة الجديدة كاملاً هو:


```
unit IntQueue;

// This unit contains TIntQueue class, which is inherits TQueue
// class and adds PutInt, GetInt methods to be used with
// Integer queue

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, Queue;

type

  { TIntQueue }

  TIntQueue = class(TQueue)

  public
    function PutInt(AValue: Integer): Integer;
    function GetInt(var AValue: Integer): Boolean;

  end;

implementation

{ TIntQueue }

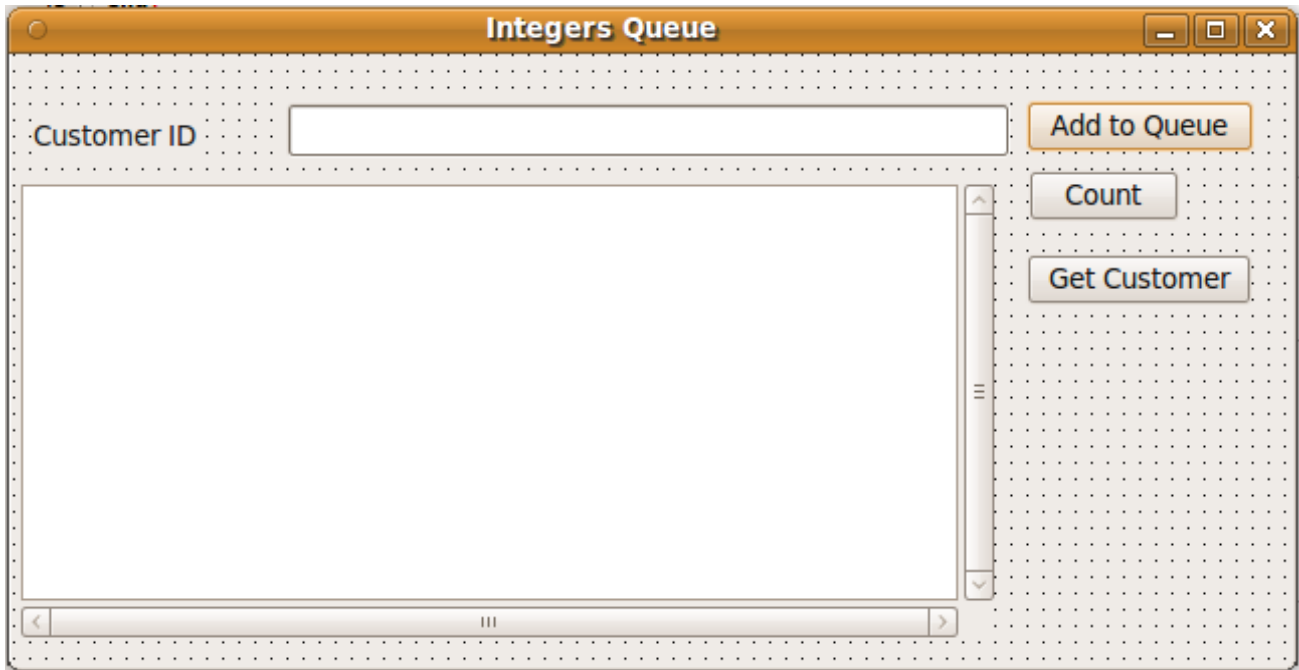
function TIntQueue.PutInt(AValue: Integer): Integer;
begin
  Result:= Put(IntToStr(AValue));
end;

function TIntQueue.GetInt(var AValue: Integer): Boolean;
var
  StrValue: string;
begin
  Result:= Get(StrValue);
  if Result then
    AValue:= StrToInt(StrValue);
end;

end.
```

نلاحظ أننا لم نُكرر بعض العمليات مثل *Count, Create, Destroy* لأنها موروثه من الكائن *TQueue*.

لاستخدام الوحدة الجديدة أنشأنا الفورم التالي:



وكود الفورم السابق هو:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,
  Dialogs, IntQueue, StdCtrls;

type

  { TfmMain }

  TfmMain = class(TForm)
    bbAdd: TButton;
    bbCount: TButton;
    bbGet: TButton;
    edCustomerID: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    procedure bbAddClick(Sender: TObject);
    procedure bbCountClick(Sender: TObject);
    procedure bbGetClick(Sender: TObject);
    procedure FormClose(Sender: TObject;
      var CloseAction: TCloseAction);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  end;

end;
```

```
public
  MyQueue: TIntQueue;
  { public declarations }
end;

var
  fmMain: TfmMain;

implementation

{ TfmMain }

procedure TfmMain.FormCreate(Sender: TObject);
begin
  MyQueue:= TIntQueue.Create;
end;

procedure TfmMain.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
  MyQueue.Free;
end;

procedure TfmMain.bbCountClick(Sender: TObject);
begin
  Memol.Lines.Add('Queue length is: ' + IntToStr(MyQueue.Count));
end;

procedure TfmMain.bbAddClick(Sender: TObject);
var
  APosition: Integer;
begin
  APosition:= MyQueue.PutInt(StrToInt(edCustomerID.Text));
  Memol.Lines.Add(edCustomerID.Text + ' has been added as # '
    + IntToStr(APosition + 1));
end;

procedure TfmMain.bbGetClick(Sender: TObject);
var
  ACustomerID: Integer;
begin
  if MyQueue.GetInt(ACustomerID) then
  begin
    Memol.Lines.Add('Got: Customer ID : ' + IntToStr(ACustomerID) +
      ' from the queue');
  end
  else
    Memol.Lines.Add('Queue is empty');
  end;

initialization
  {$I main.lrs}

end.
```

نلاحظ أننا نادينا بالإجراءات و الدوال الموجودة في الكائن *TQueue* و الدوال الجديدة الموجودة فقط في *TIntQueue*.

في هذه الحالة نُسَمي الكائن *TQueue* الكائن الأساس *base class* أو السلف *ancestor*. ونسَمي الكائن الجديد *TIntQueue* المُنحدر *descender* أو الوريث.

ملحوظة:

كان من الممكن عدم استخدام الوراثة في المثال السابق وإضافة الدوال الجديدة مباشرة في الكائن الأساسي *TQueue*، لكن لجأنا لذلك لشرح الوراثة، وهناك سبب آخر، هو أننا ربما لا نمتلك الكود المصدر للوحدة *Queue*، ففي هذه الحالة يتعذر تعديلها، ويكون السبيل الوحيد هي الوراثة منها ثم تعديلها. ويمكن للمبرمج أن يوزع النسخة الثنائية المترجمة للغة الآلة لتلك الوحدات في شكل ملفات ذات الامتداد **ppu** كما في فري باسكال أو **dcu** كما في دلفي، في هذه الحال لا يمكن الإطلاع على الكود المصدري، حيث يمكن الإطلاع على الكود فقط في حالة الحصول على الملف **pas**.

وفي الختام نتمنى أن يُنال بهذا الكتاب فائدة المسلمين.

سبحانك اللهم وبمجدك، نستغفرك ونتوب إليك

و بعد انتهاء الدارس من هذا الكتاب، يمكن بعد ذلك قراءة الكتاب الآخر "[الخطوة الثانية مع أوبجكت باسكال - صناعة البرمجيات](#)"