**الفصل الرابع**

**مسار البرنامج وأجزاؤه**

* [**جمل التحكّم في المسار Flow-Control Statements:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20جمل%20التحكّم%20في%20المسار.htm)

التعرّف على الجمل الشرطيّة والجمل التكراريّة.

* [**كتابة واستخدام الإجراءات Writing and Using Procedures:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات.htm)

التعرّف على الإجراءات: تعريفها، معاملاتها، قيمها المعادة، وطرق استدعائها.

**جمل التحكّم في المسار Flow-Control Statements:**

* [**ما هي جمل التحكّم في المسار؟**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20جمل%20التحكّم%20في%20المسار\0-%20ما%20هي%20جمل%20التحكّم%20في%20المسار.htm)
* [**تراكيب الاختبار Test Structures:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20جمل%20التحكّم%20في%20المسار\1-%20تراكيب%20الاختبار.htm)
* [**تراكيب التكرار Loop Structures:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20جمل%20التحكّم%20في%20المسار\2-%20تراكيب%20التكرار.htm)
* [**جمل التحكّم المتداخلة Nested Control Structures:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20جمل%20التحكّم%20في%20المسار\3-%20جمل%20التحكّم%20المتداخلة.htm)
* [**الانتقال الحرّ عن طريق "اذهب إلى" GoTo:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20جمل%20التحكّم%20في%20المسار\4-%20الانتقال%20الحرّ.htm)

**ما هي جمل التحكّم في المسار؟:**

إنّ البرمجة أعمق من أن تكون مجرّد تعريف متغيّرات.. إنّها تفكير منطقيّ يعتمد على حساب كل الاحتمالات، لاتخاذ الأفعال المناسبة لكل احتمال.. لهذا فلا بد أن توجد طرق نتحكّم بها فيما ينفّذ ومتى ينفّذ من البرنامج.

ولقد رأينا بالفعل كيف نكتب تطبيقنا مجزءا في مجيبات الأحداث Event Handlers، بحيث نستجيب لأداء المستخدم لفعل ما بالفأرة (كضغط زر مثلا) أو بلوحة المفاتيح (كالكتابة أو الحذف من مربع نص).

ولكن ليس هذا هو المقصود هنا.. إنّنا نريد المزيد من التحكّم.. حتّى داخل الإجراء نفسه.

وهذا قد رأيناه، فلقد أسرفنا في استخدام جمل الشرط If Statements في أمثلتنا السابقة، لنحدّد ما نريد تنفيذه من أوامر، تبعا لشروط معيّنة.

ورأينا أيضا كيف يمكن استخدام الجملة التكرارية "من إلى" For… Next لتكرار تنفيذ مقطع من الكود عددا من المرات.

كما رأينا كيف نكتب معالجات الاستثناءات Exception Handlers للحماية من حدوث خطإ معيّن، بكتابة الكود الذي يمنعه من تدمير البرنامج وإغلاقه.

إنّ الكمبيوتر بدون مثل هذه الطرق سيبدو كما لو كان مجرّد آلة حاسبة ضخمة!

**تراكيب الاختبار Test Structures**

* + [**جملة الشرط If…Then:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20تراكيب%20الاختبار\1-%20جملة%20الشرط.htm)
  + [**جملة الشرط If…Then…Else:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20تراكيب%20الاختبار\2-%20جملة%20الشرط.htm)
  + [**وكذلك AndAlso.. أو غير ذلك OrElse:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20تراكيب%20الاختبار\3-%20وكذلك%20وغير%20ذلك.htm)
  + [**جملة اختيار الحالة Select Case:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\1-%20تراكيب%20الاختبار\4-%20جملة%20اختيار%20الحالة.htm)

**جملة الشرط If…Then:**

تستطيع أن تختبر حدوث شرط معيّن، فإذا كان صحيحا يتمّ تنفيذ مقطع الشرط، وإن كان خاطئا يقفز التنفيذ إلى جملة نهاية الشرط End If.

ولكن ما هو الشرط؟

الشرط هو أيّ متغيّر منطقي أو أيّ تعبير يعطي نتيجة منطقيّة (True أو False)، مثل:

**X > 3**

**Y < 2 And X = 5**

**Not( (X+Y = 5) Or (X - Y < 3) )**

وكما قلنا آنفا، إذا كان مقطع الشرط جملة واحدة، فلا حاجة بنا إلى جملة نهاية الشرط End if، كالتالي:

**If شرط Then جملة**

ويمكن تنفيذ أكثر من جملة في سطر واحد، بفصلها بنقطتين قائمتين ":" كالتالي:

**If شرط Then جملة: جملة: جملة**

مثال:

**Dim X As Integer = 3, Y As Integer = 5**

**If X > Y Then X += 1: Y -= 2 : X /= Y : Y\*= X**

ويمكن كتابة جملة الشرط السابقة بالطريقة التقليديّة كالتالي:

**If X > Y Then**

**X += 1**

**Y -= 2**

**X /= Y**

**Y\*= X**

**End If**

وفي حالة التعامل مع متغيّر منطقيّ، يمكن استخدام جملة الشرط كالتالي:

**Dim Check As Boolean = True**

**If Check = True Then Check = False**

**If Check = False Then Exit Sub**

ولاختصار الكتابة، يعتبر VB أن قيمة المتغيّر المنطقيّ = True إذا لم يذكر ذلك صراحةً.. بمعنى أنّ الجملة التالية:

**If Check Then Check = False**

مكافئة تماما للجملة:

**If Check = True Then Check = False**

**احذر:**

كان مبرمجو VB6 معتادين على استخدام الاختصار في كتابة الشرط، حتّى مع المتغيّرات الرقميّة.. وكان هذا يعني تنفيذ جملة الشرط إذا كانت للمتغيّر أيّ قيمة غير الصفر.. إنّ هذا لم يعد مسموحا به في حالة اختيار التحويل الدقيق Option Strict On.. فمثلا جملة الشرط التالية لن يسمح بها في حالة اختيار التحويل الدقيق:

**Dim X AS Integer = 5**

**If X Then ' لو كانت قيمة المتغيّر لا تساوي صفرا فنفّذ الشرط**

**X = 0**

**End If**

ولكتابة ذلك بطريقة مقبولة لديك الحلان التاليان:

فإمّا أن تستخدم الشرط:

**If X <> 0 Then**

أو تحوّل الشرط إلى صيغة منطقيّة كالتالي:

**If CBool (X) Then**

**جملة الشرط If…Then…Else:**

هذه طريقة أخرى لكتابة جمل الشرط، نحتاج إليها إذا أردنا تنفيذ مقطع من الكود لو كان الشرط صحيحا، ومقطع آخر لو كان خاطئا:

**If شرط Then**

**مقطع1.. (وهو ينفذ في حالة تحقّق الشرط).**

**Else**

**مقطع2.. (وهو ينفّذ في حالة عدم تحقّق الشرط).**

**End If**

ويمكن التحقّق من أكثر من شرط، فإذا لم يتحقّق أيّ منها يقفز البرنامج لتنفيذ مقطع "ما عدا ذلك" Else، وذلك تبعا للصيغة التالية:

**If شرط1 Then**

**مقطع1 (وهو ينفّذ في حالة تحقّق شرط1)**

**ElseIf شرط2 Then**

**مقطع2 (وهو ينفّذ في حالة تحقّق شرط2)**

**ElseIf شرط3 Then**

**مقطع3 (وهو ينفّذ في حالة تحقّق شرط3)**

**Else**

**مقطع4 (وهو ينفّذ في حالة عدم تحقّق أيّ شرط)**

**End If**

ولا قيود عليك في إضافة العدد الذي تريده من جمل ElseIf.

مثال:

**Dim Score As Integer = 83**

**If score < 50 Then**

**Result = "راسب"**

**ElseIf score < 75 Then**

**Result = "ناجح"**

**ElseIf score < 90 Then**

**Result = "جيد جدا" ' هذا هو المقطع الذي سيتم تنفيذه**

**Else**

**Result = "ممتاز"**

**End If**

**MsgBox (Result) ' ستعرض الرسالة جيد جدا**

وتمتاز جملة ElseIf بأنّ VB يختبر الشروط من أعلى لأسفل، وعند نجاح واحد منها لا يختبر باقي الشروط، بل ينفّذ المقطع الخاص بالشرط المتحقّق ثمّ يقفز مباشرة لجملة نهاية الشرط.. لهذا فالمثال السابق أفضل من المثال التالي، مع أنّ كليهما يؤدّي نفس الوظيفة:

**If score < 50 Then**

**Result = "راسب"**

**End If**

**If score < 75 And score >= 50 Then**

**Result = "ناجح"**

**End If**

**If score < 90 And score > =75 Then**

**Result = "جيد جدا"**

**End If**

**If score >= 90 Then**

**Result = " ممتاز"**

**End If**

ففي هذه الحالة سيتمّ اختبار كلّ الشروط، رغم أنّه من المستحيل حدوث أكثر من واحد منها في نفس الوقت.. هذا بالإضافة للدرجة الأكبر من التعقيد التي اضطررنا لإضافتها للشروط حتّى تعمل بطريقة صحيحة، فمثلا: لكي تكون النتيجة "ناجح" فليس كافيا في هذه الحالة أن يكون الشرط هو:

score < 75

لأنّ المجموع لو كان 40، فسيتحقّق الشرط الأوّل (40 < 50) وستكون النتيجة "راسب"، ولكن عند اختبار الشرط التالي فسيجد VB كذلك أن 40 أصغر من 75، لذلك ستصبح النتيجة "ناجح"!!.. لهذا وجب تعديل الشرط إلى:

score < 75 And score >= 50

واضح تمامًا إذن، أنّ تركيب ElseIf أصغر في الحجم، وأسهل في تصميم الشروط، وأسرع في التنفيذ.

ولكنّ هذا لا يمنع أنك قد تحتاج إلى مجموعة جمل شرط مستقلّة متتالية، وذلك في الحالات التي لا ترتبط فيها الشروط ببعضها، بحيث يمكن حدوث أكثر من شرط منها معا في نفس الوقت.

كما يجب أن تلاحظ أنّ الترتيب هامٌّ جدًّا في تركيب جمل ElseIf.. فلو لخبطت ترتيبها لتصبح كالتالي:

**If score < 75 Then**

**Result = "ناجح"**

**ElseIf score < 50 Then**

**Result = "راسب"**

**ElseIf score < 90 Then**

**Result = "جيد جدا"**

**Else**

**Result = "ممتاز"**

**End If**

فلن تحصل على نتائج صحيحة.. فمثلا: لو كان المجموع 49، فسيجد VB أنّه أصغر من 75، وبالتالي ستكون النتيجة "ناجح" وهي بالطبع نتيجة خاطئة!

لهذا توخّ الدّقّة وأنت تكتب جمل الشرط، حتّى لا تحصل على نتائج غير متوقّعة.

**وكذلك AndAlso.. أو غير ذلك OrElse:**

تعرف طبعا أنّ الشرط يمكن أن يتكوّن من مجموعة شروط جزئيّة تربطها معا المعاملات And أو Or أو Not أو Xor.. في هذه الحالة يتمّ التحقّق من كلّ شرط على حدة أولا، ليتم استبدل كلّ شرط بناتجه (True أو False).. وبهذا تتحوّل الجملة إلى مجموعة قيم True و False تربطها المعاملات المنطقيّة.. وطبعا ناتج ذلك في النهاية هو قيمة واحدة فقط: True أو False.. ولو جرّبت الكود التالي:

**Dim B As Boolean = True**

**If B And MsgBox("رسالة اختبار1") = MsgBoxResult.OK Then**

**MsgBox("رسالة اختبار2")**

**End If**

فستجد أنّ الرسالة الأولى ستظهر، وعندما تضغط موافق، فإنّ الشرط:

**MsgBox("رسالة اختبار1") = MsgBoxResult.OK**

ستصبح نتيجته True، وبما أنّ قيمة B هي True، فإنّ الشرط يتحوّل بالنسبة لـ VB إلى:

**(True And True)**

وطبعا ناتج هذا الشرط هو True، وبالتالي تظهر الرسالة الثانية.

ولكن لو جرّبت الجملة التالية:

**If B = False And MsgBox("رسالة 1") = MsgBoxResult.OK Then**

**MsgBox("رسالة 2")**

**End If**

فستجد أنّ الرسالة الأولى ستظهر ولن تظهر الرسالة الثانية.

وستتساءل: لماذا ظهرت الرسالة الأولى، مع أنّ الشرط B = False هو شرط غير متحقّق؟.. إنّ كون الشرط الأوّل في عمليّة And خاطئا، يعنى بالضرورة أنّ ناتج العمليّة سيكون False مهما كان ناتج الشرط الثاني.. فلماذا اختبر VB الشرط الثاني مع أنّه بلا فائدة؟.. أليس هذا تضييعا للوقت؟

معك حقّ، ولكن أنت الذي بيده تحديد ذلك، فلربّما كنت تريد عرض الرسالة في كلّ الأحوال!

عامّةً لا تقلق: لقد ابتكر لك VB.Net معاملا جديدا اسمه "وكذلك" AndAlso، وذلك لاختصار التحقّق من الشروط عديمة الفائدة.. فلو كان الشرط الأوّل خاطئا، فلن يتمّ التحقّق من باقي الشروط.. فمثلا لو جرّبت:

**If B = False AndAlso MsgBox("رسالة 1") = MsgBoxResult.Cancel Then**

**MsgBox("رسالة 2")**

**End If**

فلن تظهر لك الرسالة الأولى.

أمّا لو جرّبت:

**If B AndAlso MsgBox("رسالة 1") = MsgBoxResult.Cancel Then**

**MsgBox("رسالة 2")**

**End If**

فإنّ كون الشرط الأوّل صحيحا لا يحسم عمليّة And، لهذا فلا بدّ أن يتمّ التحقّق من الشرط الثاني.. لهذا فستظهر لك الرسالة الأولى، ولكن لأنّ الشرط الثاني لن يكون صحيحا (لأنّك ستضغط زر Ok بينما الشرط يتحقّق من أنّك تضغط زر Cancel)، فإنّ الرسالة الثانية لن تظهر.

ولتدرك أهمّيّة هذا المعامل الجديد، انظر مثلا كيف يكتب مبرمج VB6 كودا يتأكّد من أنّ الجذر التربيعيّ للعدد X أكبر من 2.. في هذه الحالة يجب أن تتحقّق أولا من أنّ X ليس عددا سالبا:

**If X >= 0 Then**

**If Sqrt(X) > 2 Then MsgBox("أكبر من 2")**

**End If**

ولو حاول مبرمج VB6 أن يكتب ذلك كالتالي:

**If X >= 0 And Sqrt(X) > 2 Then MsgBox("أكبر من 2")**

فسيعرّض برنامجه للانهيار، لأنّ خطأ سيحدث لو كان العدد X سالبا، لأنّ جملة الجذر التربيعيّ ستنفّذ في كلّ الأحوال، وبهذا يبدو الشرط الأوّل عديم الفائدة!

ولكن الآن تستطيع كمبرمج VB.Net أن تكتب:

**If X >= 0 AndAlso Math.Sqrt(X) > 2 Then MsgBox("أكبر من 2")**

واثقا تمام الثقة أنّ أيّ خطإ لن يحدث، وذلك لأنّ كون X عددا سالبا سيعيق التحقّق من الشرط الثاني.. هذه هي عبقرّية المعامل AndAlso.

ولديك موقف مشابه في حالة Or، فكون الشرط الأوّل في عمليّة Or صحيحا، يعنى بالضرورة أنّ ناتج العمليّة سيكون True مهما كان ناتج الشرط الثاني، لهذا لو أردت اختصار الوقت وإهمال الشرط الثاني في هذه الحالة، فاستخدم المعامل "أو غير ذلك" OrElse.. بهذه الطريقة ستضمن عدم اختبار أيّ شرط إلا إذا كان الشرط السابق له خاطئا.. جرّب استبدال Or و OrElse بـ And و AndAlso في الأمثلة السابقة.

**الخلاصة:**

استخدم And و Or إذا أردت إجبار VB على اختبار كلّ الشروط، وذلك إذا كان بعض هذه الشروط يحتوي على دوال تريد استدعاءها دائما مهما كانت نتيجتها.

أمّا إذا أردت تسريع التنفيذ وتسهيل كتابة الكود، فاستخدم AndAlso و OrElse لتجنّب اختبار الشروط التي لا داعيَ لاختبارها.

**جملة اختيار الحالة Select Case:**

هذه حالة خاصة من جملة الشرط، لتسهيل كتابة الكود عندما تكون كل المقارنات بين متغير واحد، ومجموعة من القيم من نفس نوع المتغيّر.. ويوضع اسم المتغيّر بعد جملة Select Case، بينما توضع القيم المختلفة واحدة بواحدة بعد كلمة Case، متبوعة بالمقطع الذي سيتم تنفيذه في حالة تحقّق التساوي:

**Select Case تعبير**

**Case قيمة1**

**مقطع1 (يتم تنفيذه حينما يساوي التعبير القيمة1)**

**Case قيمة2**

**مقطع2 (يتم تنفيذه حينما يساوي التعبير القيمة2).**

**………………….**

**………………….**

**Case Else**

**مقطع\_أخير (يتم تنفيذه حينما لا يساوي التعبير أي قيمة مما سبق)**

**End Select**

لماذا استخدمنا كلمة تعبير وليس كلمة متغيّر؟.. ببساطة لأنّك تستطيع اختبار قيمة المتغيّر X، أو اختبار قيمة التعبير 2 \* X كذلك.. وهكذا.

وإليك هذا المثال:

**Dim Message As String**

**Select Case Now.DayOfWeek**

**Case DayOfWeek.Monday**

**Message = "أسبوعا ممتعا"**

**Case DayOfWeek.Friday**

**Message = "عطلة ممتعة"**

**Case Else**

**Message = "مرحبا بعودتك"**

**End Select**

**MsgBox(Message)**

وفي هذا المثال نختبر في أيّ يوم من الأسبوع نحن، باستخدام الخاصّيّة "يوم الأسبوع" DayOfWeek الخاصّة بالمتغيّرات من النوع تاريخ.. ولكن أين هو ذلك المتغيّر من النوع تاريخ؟.. ليس موجودا مباشرة، ولكن يمكن افتراض وجوده، لأنّ الدالة Now ترجع قيمة من نوع هذا المتغيّر، ولقد ذكرنا سابقا أنّ القيم المعادة يمكن استخدام أعضاء نوعها.

بعد ذلك، نختبر القيم التي تعنينا في كل "حالة" Case.. وتلاحظ أنّنا نستخدم تعبيرات رقميّة لأيّام الأسبوع بدلا من الأرقام، مع أنّ الأرقام يمكن استخدامها، ولكنّ هذه الطريقة ستكون أسهل في الفهم عند قراءتها.

وكلّ ما سيحدث، هو أن VB سيختبر الحالات حالةً حالةً من أعلى إلى أسفل، وعند تحقّق إحداها سينفّذ المقطع الخاصّ بها، ثم سيقفز لجملة نهاية الاختيار End Select، دون أن يكمل اختبار باقي الحالات.

أمّا لو لم تتحقّق أيّ حالة، فسينفّذ VB مقطع "حالة أخرى" Case Else (إن كان موجودا، فبإمكانك ألا تكتبه).

ولكن هل أنا مضطرّ لاختبار قيمة واحدة في كلّ حالة؟

لا بالطبع، فبإمكاننا وضع أكثر من قيمة في الحالة الواحدة، بحيث يكفي حدوث إحداها ليتمّ تنفيذ مقطع حالته.. انظر للمثال التالي:

**Select Case Now.DayOfWeek**

**Case 0**

**Message = "أسبوعا ممتعا"**

**Case 1, 2, 3 ' تنفيذ المقطع في حالة 1 أو 2 أو 3**

**Message = "مرحبا بعودتك"**

**Case 4, 7, 6**

**Message = "عطلة ممتعة"**

**End Select**

**MsgBox(Message)**

والآن قارن جملة اختيار الحالة مع جملة الشرط التالية:

**If Now.DayOfWeek = 0 Then**

**Message = "أسبوعا ممتعا"**

**Else**

**If Now.DayOfWeek >= 1 And Now.DayOfWeek <= 3 Then**

**Message = "مرحبا بعودتك"**

**Else**

**Message = "عطلة ممتعة"**

**End If**

**End If**

**MsgBox(message)**

أعتقد أنّك تشاركني الرأي، في أنّ جملة اختيار الحالة أسهل بكثير.

**ملاحظة:**

العلامة ">=" تعني "أكبر من أو يساوي" (لا تنس أنّها بالإنجليزية، أي أنها ستبدو لك عكس علامة أكبر من بالعربيّة)، وعلامة "<=" تعني "أصغر من أو يساوي".

وهناك طرق كثيرة لدمج القيم في تعبيرات الحالات.. انظر هذا المثال، ولاحظ هذه الطرق.. لن نكتب أي مقاطع من الكود، وعلى فكرة، لن يعترض VB على هذا، معّ أنّه يعني أنّ شيئا لن ينفّذ:

**Dim X As Integer = 367, Y As Integer = 200**

**Select Case X**

**Case X / 2 ' المقارنة بعملية يدخل فيها المتغيّر نفسه**

**' لاحظ أنّ هذا الشرط يمكن أن يتحقّق في حالة واحدة،**

**' إذا كانت قيمة المتغيّر صفرا!**

**Case Y ' المقارنة بقيمة متغير آخر**

**Case 1 'المقارنة بقيمة واحدة**

**Case 2, 3, 7 ' المقارنة بمجموعة من القيم**

**Case 10 To 20 ' المقارنة بنطاق القيم من 10 إلى 20**

**' لاحظ أن 10 و 20 داخل المقارنة.**

**Case Is > 100 ' مقارنة منطقيّة: لو كان المتغير أكبر من 100 يتم تنفيذ الحالة**

**' لست مجبرا على كتابة كلمة Is فاللغة ستكتبها لك لو لم تفعل.**

**Case Is <= 1000 ' مقارنة منطقيّة أخرى، ولكن باستخدام أصغر من أو يساوي**

**Case Is < 1000, 10 ' أصغر من 1000 أو يساوي 10**

**Case 20 , Is >= 10000, 20000 To 100000**

**' مزيج من المقارنات معا**

**Case Else ' أي حالة خلاف ما سبق**

**End Select**

**تراكيب التكرار Loop Structures**

إنّ قدرة الكمبيوتر على تكرار أي جزء من الكود ـ خاصّةً مع سرعته الفائقة ـ هي ما تجعله مريحا جدا للبشر، ليحمل عنهم عناء الرتابة والبطء والملل.

ويقدّم لك VB هذه التركيبات التكراريّة:

* + [**تركيب التكرار "من إلى" For…Next:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20تراكيب%20التكرار\1-%20من%20إلى.htm)
  + [**تركيب التكرار "نفّذ مرارا" Do…Loop:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20تراكيب%20التكرار\2-%20نفّذ%20مرارا.htm)
  + [**تركيب التكرار "بينما" While:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20تراكيب%20التكرار\3-%20بينما.htm)

**جملة التكرار "من إلى" For…Next:**

هذا هو التركيب التكراريّ الوحيد الذي يتطلّب أن تعرف عدد المرّات التي سيتمّ تنفيذه فيها، حيث يعتمد على وجود عداد للتكرار Counter، تحدّد له قيمة البداية، وقيمة النهاية، والخطوة Step التي سيقفزها (القيمة التي سيزيد بها) كلّما وصل التنفيذ لكلمة "التالي" Next:

**For اسم\_متغير = بداية To نهاية [Step قيمة\_الزيادة]**

**{المقطع الذي سيتم تكراره}**

**Next [اسم\_المتغير]**

**ملاحظة:**

في كتب البرمجة، وجود أي جزء من الصيغة بين قوسين مضلعين [ ]، معناه أنّ هذا الجزء اختياريّ Optional، لستَ مجبرا على كتابته في الكود.

وهذه هي الطريقة التي يُنفّذ بها VB هذا التركيب:

1- توضع قيمة البداية في المتغيّر الذي يعمل كعدّاد للتكرار.

2- يتم اختبار إذا ما كان العدّاد أكبر من قيمة النهاية أم لا، فإذا كان أكبر، يخرج التنفيذ من جملة التكرار إلى السطر التالي لجملة Next.. ومن الممكن أن تكون قيمة الزيادة سالبة، في هذا الحالة يجب أن تكون قيمة النهاية أصغر من قيمة البداية.. وسيتأكّد VB ممّا إذا كان العداد أصغر من قيمة النهاية أم لا.. تذكّر أنّ -3 أصغر من -2.

3- يتم تنفيذ الجمل الموجودة في مقطع التكرار.

4- عند وصول التنفيذ إلى جملة Next، يتم جمع قيمة الزيادة على العدّاد.. وطبعا لو كانت قيمة الزيادة سالبة، فستقلّ قيمة العداد.. ولو لم تكن قيمة الزيادة مذكورة، فسيتم اعتبارها 1.

5- يتم تكرار الخطوات 2، 3، 4، حيث يتم تنفيذ مقطع التكرار في كلّ دورة، حتّى يتجاوز العدّاد قيمة النهاية، فيخرج التنفيذ من مقطع التكرار.

مثال: هذا البرنامج سيعرض لك 11 رسالة متتالية، كل منها تحمل رقمها:

**Dim I As Integer**

**For I = 0 To 10**

**MsgBox(I)**

**Next I**

**ملاحظة:**

رغم أنّ ذكر اسم المتغيّر بعد كلمة Next ليس إجباريّا، إلا إنّني أنصحك به، لأنّه سيسهّل عليك قراءة الكود، خاصّةً في الحالات التي تتداخل فيها أكثر من جملة تكرار.

ويمكن الخروج في الحال من جملة التكرار "من إلى" باستخدام جملة Exit For، ولو جرّبت المثال التالي، فستجد أنّ الرسالة ستظهر لك ست مرات فقط بدلا من 11:

**Dim I As Integer**

**For I = 0 To 10**

**MsgBox(I)**

**If I = 5 Then Exit For**

**Next I**

أهمّ شيء يجب أن تعرفه، هو أنّ معاملات التكرار (البداية والنهاية والخطوة) يتم قراءتها مرة واحدة في بداية تنفيذ تركيب التكرار، لهذا فإنّ تغيير قيم هذه المعاملات في مقطع التكرار نفسه، لن يكون له أي تأثير.

ولو أعدت كتابة المثال الأوّل بالطريقة التالية، فلن يتغيّر شيء في تنفيذ البرنامج:

**Dim I As Integer, X As Integer = 10**

**For I = 0 To X**

**MsgBox(I)**

**X = 5 ' لن تؤثّر هذه الجملة في قيمة نهاية جملة التكرار**

**Next I**

إنّ هذا يسمح لك بجعل قيمة النهاية تعبيرا مبنيا على العدّاد نفسه، دون أن تخشى من أن يؤثّر تغيّر العدّاد عليها.. فمثلا، الجملة التكراريّة التالية ستعرض 11 رسالة:

**Dim I As Integer = 20**

**For I = 0 To I / 2**

**MsgBox(I)**

**Next I**

ولكن على العكس، يؤثّر تغيّر قيمة العدّاد على تنفيذ المقطع، وهو ما يجب أن تعيه جيّدا وتحترس له.. إنّ المقطع التالي سينفّذ 6 مرات فقط بدلا من 11، نتيجة التلاعب بقيمة العداد في ثناياه:

**Dim I As Integer**

**For I = 0 To 10**

**MsgBox(I)**

**I += 1**

**Next I**

بينما المقطع التالي لن ينتهي أبدا، لأنّ العدّاد ينقص 1 قبل أن تتم زيادة قيمته في جملة Next، ممّا يعمل على ثبات قيمته إلى الأبد:

**Dim I As Integer**

**For I = 0 To 10**

**MsgBox(I)**

**I -= 1**

**Next I**

ولإيقاف مثل هذا البرنامج العالق، انتقل إلى نافذة VS.Net، واضغط Ctrl+Pause من لوحة المفاتيح (أو Ctrl+Break، على حسب ما هو مكتوب على هذا الزر في لوحة المفاتيح).

أعتقد أنّ هذا المثال سيردعك عن استخدام مثل هذه الطريقة الخطيرة للتلاعب في عدد مرات تكرار المقطع.

ولكنّك أحيانا لا تعلم بالضبط عدد المرات التي تريد تكرار المقطع بها.. فماذا تفعل؟

لهذا السبب يا صديقي توجد أنواع أخرى من جمل التكرار!.. اصبر وسترى.

**تركيب التكرار "نفّذ مرارا" Do…Loop:**

هذا التركيب أيضا ينفّذ مقطعا من الكود، ولكنّه أعمّ من متكرّرة For… Next، لأنّه يعتمد على صحّة شرط ما لاستمرار التكرار، بدلا من الاعتماد على قيمة العدّاد.. إنّ هذا يُمكّنك من تعقيد هذا الشرط لأيّ درجة.

وهناك صيغ عديدة لهذا التركيب:

**1- الصيغة اللانهائيّة:**

**Do**

**{ مقطع من الكود**

**If شرط Then Exit Do**

**بقية مقطع الكود }**

**Loop**

وهي لا نهائيّة، لأنّ تركيب التكرار لا يحتوي على أي شرط، وما لم يتم الخروج من هذا المقطع بجملة Exit Do، فهو لن ينتهي أبدا.

**2- صيغة "نفّذ بينما":**

**Do While شرط**

**مقطع**

**Loop**

وفي هذه الصيغة، يتم تنفيذ المقطع **بينما** يكون الشرط صحيحا، فإذا صار الشرط خطأً توقّف التنفيذ.. ويتم اختبار الشرط في بداية تنفيذ المقطع، ثمّ في كل مرّة يتم فيها تنفيذ كلمة التكرار Loop.

وهناك صيغة فرعيّة من هذه الصيغة:

**Do**

**مقطع**

**Loop While شرط**

وتمتاز هذه الصيغة بأنّها تُنفّذ المقطع مرّة أولا، قبل أن تتحقّق من صحّة الشرط، وبهذا تضمن تنفيذ المقطع مرّة واحدةً على الأقلّ.

**3- صيغة "نفّذ حتّى":**

**Do Until شرط**

**Loop**

وهي عكس الصيغة السابقة، فالتنفيذ سيستمرّ طالما أنّ الشرط خطأ، فإذا تحقّق توقّف التنفيذ.. وأيضا توجد صيغة فرعيّة من هذه الصيغة:

**Do**

**Loop Until شرط**

مثال:

نريد أن نكتب برنامجا يحسب عدد الكلمات التي يدخلها المستخدم في مربّع إدخال.. ونظرا لأنّ الكلمات تفصلها مسافات، فسنقوم بحساب عدد المسافات داخل النصّ، وسنفترض أنّه لا توجد مسافات متتالية متكرّرة.

في هذا البرنامج، سنحتاج للدالة "في النص" InStr للبحث عن حرف المسافة " " في النص.. وهذه الدالة تأخذ ثلاثة معاملات، هي بالترتيب: رقم الحرف الذي يبدأ عنده البحث داخل النص، والنص الذي سيتم بحثه، والنص الهدف الذي سيتمّ البحث عنه.

وفي حالة عثور هذه الدالة على الهدف، فإنّها تُرجع رقم أوّل حرف منه في النص.. أمّا إذا لم تعثر على شيء، فإنّ الناتج يكون صفرا.

**ملاحظة:**

تبدأ دالة "في النص" InStr البحث من موضع البداية المحدّد في معاملاتها، فلو كان الهدف في موضع البداية، فسيكون الناتج هو هذا الموضع نفسه.. ولو كرّرت هذه العمليّة ألف مرّة فلن يتغيّر الناتج أبدا.. لهذا عند استخدام هذه الدالة في جملة تكراريّة، تأكّد أنّك تبحث بدايةً من الموضع التالي لآخر موضع عثرت فيه على الهدف، حتّى لا يستمر التكرار إلى ما لا نهاية.

**Dim MyText As String = InputBox("أدخل النصّ الذي تريد حساب عدد كلماته")**

**Dim Position = 1, Words As Integer**

**Do While Position > 0 ' سيصبح هذا المتغير صفرا عندما لا تعثر الدالة على مسافات**

**' في كل مرة نبحث من الموضع التالي مباشرة لموضع المسافة السابقة.. لهذا نجمع 1**

**Position = InStr(Position + 1, MyText, " ")**

**' حتّى لو لم نعثر على مسافة فسنجمع 1 على عدد الكلمات،**

**' لأنّ عدد الكلمات أكبر من عدد المسافات بواحد.**

**Words += 1**

**Loop**

**MsgBox ("عدد الكلمات هو " & Words)**

**ملحوظة:**

هناك وسائل جاهزة وسهلة لتقطيع النصوص إلى أجزاء تبعا لوجود فواصل معيّنة فيها، كالمسافات وغيرها، مثل الدالة Split الخاصّة بخليّة النصوص String Class، ولكنّنا نستخدم هذا المثال فقط لتوضيح عمل الجمل التكراريّة.

هناك مشكلة في المثال السابق، فهو يفترض أنّ النصّ دائما به كلمات، لهذا فهو يبدأ البحث من الموضع 1، ويجمع 1 على عدد الكلمات حتّى لو لم يجد أيّ مسافات.. ولكن لو كان النص فارغا، فستجد أن البرنامج سيخبرك بأنّ عدد الكلمات 1!!

ويمكن إصلاح هذه المشكلة كالتالي:

**Do While InStr(Position + 1, MyText, " ")**

**Position = InStr(Position + 1, MyText, " ")**

**Words += 1**

**Loop**

**MsgBox ("عدد الكلمات هو " & Words)**

والحكمة في هذا التعديل، أنّ عدم العثور على مسافات سيجعل شرط جملة التكرار خاطئا منذ البداية، ممّا سيعمل على عدم تنفيذ مقطع التكرار في هذه الحالة.

ولكنّك مستاء.. من الغباء تكرار جملة InStr مرّتين، خاصّة في مقطع تكراريّ لا ندري كم عدد المرات التي سيتكرّرها.. فكيف نصلح هذا إذن؟

خذ هذا الحل المثاليّ، الذي أعتقد أنّك ستفهمه بمجرّد النظر:

**Do**

**Position = InStr(Position + 1, MyText, " ")**

**If Position Then**

**Words += 1**

**Else**

**Exit Do**

**End If**

**Loop**

**تركيب التكرار "بينما" While:**

وهو مماثل تماما لتركيب" نفّذ بينما" Do While، مع اختلاف الكلمات فقط، لدرجة أنّك تتساءل: ما هي الحكمة في إضافة هذا التركيب لـ VB؟

**While شرط**

**مقطع**

**End While**

ويمكن اصطناع متكرّرة لا نهائيّة عن طريق صيغة كالتالية:

**While True**

**{ مقطع من الكود**

**If شرط Then Exit While**

**بقية مقطع الكود }**

**End While**

**الانتقال الحرّ عن طريق "اذهب إلى" GoTo:**

من أدوات التحكّم في مسار تنفيذ البرنامج أيضا، جملة "اذهب إلى" GoTo، حيث تمنحها اسم لافتة وضعتها في مكان ما من الإجراء الحاليّ، لتنتقل إليها:

**MsgBox("1")**

**GoTo Line1**

**MsgBox("2")**

**Line1:**

**MsgBox("3")**

وفي هذا المثال، سيتم عرض الرسالة "1" ثم الرسالة "3"، لأنّ جملة GoTo تجعل التنفيذ يقفز للسطر التالي للافتة المسمّاة Line1، وبهذا لن يتم عرض الرسالة "2" أبدا.

واللافتة هي أي اسم مقبول (يسير على قواعد تعريف المتغيرات)، تكتبه في بداية سطرٍ وتتبعه بنقطتين متعامدتين ":".

ويمكن استخدام GoTo شرطيّا، كالتالي:

**Dim X As Integer = 3**

**MsgBox("1")**

**If X > 2 Then GoTo Line1**

**MsgBox("2")**

**Line1:**

**MsgBox("3")**

ولكن لا يمكن استخدام GoTo للانتقال من إجراء إلى إجراء آخر.

ملحوظة لمبرمجي VB6:

تم حذف جملة GoSub من اللغة.

**جمل التحكّم المتداخلة Nested Control Structures:**

في كل جملة تحكّم شرطيّة أو تكراريّة، مقاطع يتم تنفيذها، ولا يوجد أي مانع لأن تحتوي هذه المقاطع بدورها على جمل تحكّم تنفّذ مقاطعا، بها جمل تحكم بها مقاطع..... إلخ!!

إن الطريق مفتوحة أمامك لتعقيد تراكيب برنامجك كما يحلو لك، وذلك تبعا للهدف الذي تريد تحقيقه.. أهمّ شيء، هو أن تراعي ترتيب بداية ونهاية كل مقطع من مقاطع التحكّم، وستساعدك بيئة التطوير في هذا، فبمجرد كتابة بدايته وضغط زر الإدخال Enter، ستقوم بيئة التطوير بوضع نهاية التركيب آليّا، بالإضافة إلى أنّ محرّر الكود ينظّم المسافات البادئة لكل مقطع تلقائيّا.

انظر لهذا المثال، لتتبين طريقة تداخل الجمل:

**Dim X(6, 4) As Integer**

**Dim I, J As Integer**

**For I = 0 To X.GetUpperBound(0)**

**For J = 0 To X.GetUpperBound(1)**

**X(I , J) = I \* 100 +J**

**Console.Write(I & ", " & J & " = " & X(I, J) & " ")**

**Next J**

**Console.WriteLine()**

**Next I**

وهذا المثال يوضّح لك كيف تتعامل مع مصفوفة ثنائيّة البعد، وذلك عن طريق متكرّرة تمرّ على صفوف المصفوفة صفا صفا، وفي كل صف تستخدم متكرّرة أخرى تمرّ على أعمدة هذا الصفّ عمودا عمودا.

معنى هذا أنّ المتكرّرة الخارجيّة ستلفّ بعدد صفوف المصفوفة، وفي كل لفّة ستلفّ المتكرّرة الداخليّة بعدد أعمدتها.. أي أنّ عدد مرّات تنفيذ جملة Console.Write هو: عدد الصفوف × عدد الأعمدة، أي 7 × 5 = 35 مرّة.

وهكذا سيبدو ما تراه في نافذة المخرجات (بافتراض أنّنا قد ملأنا المصفوفة X ببعض الأرقام أولا):

0, 0 = 0 0, 1 = 1 0, 2 = 2 0, 3 = 3 0, 4 = 4

1, 0 = 100 1, 1 = 101 1, 2 = 102 1, 3 = 103 1, 4 = 104

2, 0 = 200 2, 1 = 201 2, 2 = 202 2, 3 = 203 2, 4 = 204

3, 0 = 300 3, 1 = 301 3, 2 = 302 3, 3 = 303 3, 4 = 304

4, 0 = 400 4, 1 = 401 4, 2 = 402 4, 3 = 403 4, 4 = 404

5, 0 = 500 5, 1 = 501 5, 2 = 502 5, 3 = 503 5, 4 = 504

6, 0 = 600 6, 1 = 601 6, 2 = 602 6, 3 = 603 6, 4 = 604

بقيت هناك طريقة هامّة جدا للتحكم في مسار البرنامج، ألا وهي استدعاء الإجراءات والدوال، ولكنّ شرحها يتطلّب أولا أن تكتب الإجراءات والدوالّ الخاصّة بك، وهذا هو موضوعنا التالي.

**كتابة واستخدام الإجراءات Writing And Using Procedures:**

* [**لماذا نحتاج للإجراءات؟:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\1-%20لماذا%20نحتاج%20للإجراءات.htm)
* [**تعريف الإجراءات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\2-%20تعريف%20الإجراءات.htm)
* [**المعاملات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\3-%20المعاملات.htm)
* [**استدعاء الإجراء باستخدام متغيّر:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\4-%20استدعاء%20الإجراء%20باستخدام%20متغيّر.htm)
* [**أنواع أخرى للقيم المعادة:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\5-%20أنواع%20أخرى%20للقيم%20المعادة.htm)
* [**الدوال والسجلات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\6-%20الدوال%20والسجلات.htm)
* [**تعدد التعريفات Overloading:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\7-%20تعدد%20التعريفات.htm)
* [**الإجراءات الارتدادية Recursive Procedures:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\8-%20الإجراءات%20الارتدادية.htm)
* [**القوالب Modules:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\2-%20كتابة%20واستخدام%20الإجراءات\9-%20القوالب.htm)

**لماذا نحتاج للإجراءات؟:**

حتّى في البرمجة يمكنُك أن تستخدم مبدأ : "فرِّق تَسُدْ"، بمعني أنّ أذكى ما تفعلُه، هو تقسيم مشكلتك إلى أجزاءٍ صغيرة يسهل التعامل مع كل منها على حِدَة.

وليس هذا غريبا علينا، فلقد رأينا بالفعل كيف نتعامل مع برنامجنا كمجموعة منفصلة من الأحداث Events، لكل حدث منها مستجيب Handler، تكتب فيه الخطوات التي تريد تنفيذها عند حدوثه.

ولكن حتّى هذا ليس كافيا، فكثيرا ما تجد أنّ هناك أجزاء معيّنة من برنامجك تتكرّر بطريقة ثابتة في مواضع مختلفة، حتّى إنّك تتساءل في ملل: ألا توجد وسيلة تريحني من تكرار كتابة هذه الأجزاء كل مرّة؟

ثمّ ستكتشفُ أنّك تستطيع أن تنسخها من موضع سابق وتلصقها في الموضع الجديد، حيث ستهنّئ نفسك باعتبارك عبقريّا (وهو ما لا أشكّك فيه هنا)!

ولكنّ مشكلةً رهيبةً أخرى ستواجهك: ماذا سيحدث إذا اكتشفتَ خطأً في هذه الأجزاء؟

كارثة: ستفتّش عنها يا ولدي في كلّ مكان، وستسأل موج البحر وفيروز الشطآن، لكي تعدّلها كلّها جزءا بعد جزء!

ولكن هل البرمجة التي تتربّع على عرش التفكير المنطقيّ بهذا الغباء؟

كلا بالطبع، فحل هذه المشكلة موجود منذ اختراع البرمجة، حتّى في اللغات منخفضة المستوى: Low Level.

هذا الحل هو الإجراءات: كل ما ستفعله هو أنّك ببساطة ستجمع الجزء الذي يتكرّر معك كثيرا في برامجك، وتضعه في إجراء بالاسم الذي يعجبك، ثم تناديه بعد ذلك، من أيّ موضع من البرنامج يروق لك.. ولو اكتشفت خطأ أو أردت تعديلا في الأداء، فما عليك سوى تعديل هذا الإجراء مرّة واحدة!

وتوجد فائدة أخرى للإجراءات، هي أنّها تجعل الكود أسهل في فهمه، ففي البرامج الكبيرة، يشترك فريق من المبرمجين في كتابة البرنامج.. في هذه الحالة يكتب كل منهم الجزء الخاص به على صورة إجراءات واضحة الأسماء، بحيث يمكن لأي فرد من الفريق أن يستخدمها دون أن يشغل ذهنه بمحتوياتها أو بطريقة كتابتها.

هل يبدو لك هذا الكلام غريبا؟ .. لا يجب، فلقد تعاملنا معه مرارا: مثلا عندما استخدمنا الدالة ToString الخاصّة بالأرقام لتحويلها إلى نصوص، وعندما استخدمنا الدالة UBound() لتخبرنا برقم آخر خانة في مصفوفة.. نعم: هذا هو ما نتحدّث عنه هنا: استدعاء وظائف معيّنة لأيّ عدد من المرات بمجرد كتابة اسم إجراءاتها ودوالها، بدلا من كتابة كود هذه الوظائف من جديد كلّ مرّة.

ولكنّ الدوال التي تعاملنا معها دوال جاهزة، كتبها لنا VB، ولا يمكن الوصول إلى الكود الذي ينفّذها.

لا يهمّ، فما زال بإمكاننا أن نكتب إجراءاتنا ودوالنا الخاصّة.. هذا هو موضوع هذا المقطع، وفيه سنتعرف على نوعين من الإجراءات: الإجراءات الفرعيّة Subroutines والدوال Functions، وهما متماثلان تماما، إلا إنّ الدالة لها قيمة معادة Return Value (سمّها "ناتجا" لو أحببت)، وهو ما ليس موجودا في الإجراء الفرعيّ.

**تعريف الإجراءات:**

الإجراء هو مقطع من الكود ينفّذ عمليّةً ما.. ويتم وضع تعريف الإجراء داخل مقطع أيّ خليّة Class.. End Class أو قالب Module.. End Module، ولكن ليس داخل أي إجراء فرعيّ أو دالّة.

وللإجراء صيغتان:

**1- صيغة الإجراء الفرعيّ:**

**[مجال الإجراء] Sub اسم\_الإجراء ([معاملات الإجراء])**

**مقطع من الكود**

**End Sub**

**2- صيغة الدالة:**

**[مجال الدالة] Function اسم\_الدالة ([معاملات الدالة]) [As القيمة\_المعادة]**

**مقطع من الكود**

**End Function**

حيث مجال الإجراء أو الدالة هو واحدة من هذه الكلمات:

1- Public: ليصبح الإجراء متاحا للاستخدام من أي موضع في مشروعك، أو من أي مشروع آخر يضيف مرجعا لمشروعك.

2- Friend: ليصبح الإجراء متاحا للاستخدام من أي موضع في مشروعك، لكن ليس من أي مشروع آخر.

3- Private: ليصبح الإجراء متاحا للاستخدام، فقط من داخل القالب Module أو الخليّة Class أو النموذج Form أو السجّل Structure الذي ينتمي إليه.

4- Protected.

5- Protected Friend.

أمّا معاملات الإجراء أو الدالة، فهي مجموعة من المتغيّرات تعرّفها بين قوسيه (بدون كلمة Dim)، لاستقبال البيانات التي تحتاجها لتنفيذ وظيفة الإجراء أو الدالة.. ويمكن أن تكون هذه المعاملات من أيّ نوع: رقميّة أو نصّيّة أو زمنيّة أو كائنات أو حتّى مصفوفات.

مثال:

الإجراء الحاليّ يعرض تاريخ اليوم في رسالة.. ضع هذا التعريف قبل جملة End Class في النموذج:

**Sub ShowDate()**

**MsgBox(Now())**

**End Sub**

ولاستدعاء هذا الإجراء من أي موضع، يكفي كتابة اسمه:

**Private Sub Button1\_Click(ByVal sender As System.Object, \_**

**ByVal e As System.EventArgs) Handles Button1.Click**

**ShowDate()**

**End Sub**

حيث عند ضغط الزر سيتم استدعاء الإجراء "عرض التاريخ" ShowDate().

ما يجب أن تلاحظه هنا، هو أنّ الإجراء المستجيب لضغط الزر هو أيضا إجراء فرعيّ، ولا يختلف عن الإجراء العاديّ، إلا في وجود تعبير "يستجيب" Handles.

كما يمكنك استخدام كلمة Call قبل اسم الإجراء، عند استدعائه:

**Call ShowDate()**

ولكن كيف يتمّ استدعاء الإجراء؟

إنّ التنفيذ ينتقل مباشرةً من سطر الاستدعاء إلى الإجراء الفرعيّ، حيث يتمّ تنفيذه، فإذا ما وصل التنفيذ إلى جملة "نهاية الإجراء" End Sub، أو جملة "غادر الإجراء" Exit Sub، أو جملة "عُدْ" Return، فإنّ التنفيذ ينتقل للسطر التالي مباشرةً للسطر الذي تمّ فيه استدعاء الإجراء.

وكما ذكرنا من قبل، فإنّ كل المتغيّرات التي يتمّ تعريفها في إجراء ما، هي موضعيّة وخاصّة بهذا الإجراء فحسب، ولا يمكن استخدامها من خارجه.

نفس الأمر ستجده عند التعامل مع الدوال، باختلاف وجود القيمة المعادة فقط.. انظر لهذا المثال لتعريف دالة ترجع لك تاريخ اليوم:

**Function ShowDate() As Date**

**Return Now()**

**End Function**

لاحظ أنّنا استخدمنا التعبير Return لإعادة ناتج الدالة، وهو شيء جديد على مبرمجي VB6.. عامّةً فليطمئنّ هؤلاء، فما زال بإمكانهم استخدام الصيغة القديمة كالتالي:

**ShowDate = Now()**

وذلك بكتابة اسم الدالة بدون معاملات، متبوعا بعلامة "=" متبوعة بالقيمة المعادة.. كأنّ اسم الدالة هو متغيّر عادي، تضع فيه قيمةً من نوعه.. وإن كان من الأفضل استخدام الطريقة الأولى، لأنّها تسمح لك بتعديل اسم الدالة في أيّ لحظة، بدون أن تقلق بشأن تعديل جملة القيمة المعادة.

**ملحوظة:**

يمكن ظهور جملة إعادة قيمة الدالة (بأيّ من صيغتيها) في أي موضع من الدالة، ويمكن تكرار هذه الجملة في أكثر من موضع.. ولكنّ هناك فرقا هامّا بين صيغة Return، وبين صيغة وضع القيمة في اسم الدالة، فجملة Return تؤدّي للخروج من الدالة مباشرة، ولا يمكن تنفيذ أي كود يليها في نفس مقطعها، أمّا الصيغة الثانية، فهي لا تُنهي الدالة، ولو كان من المحتّم أن تفعل ذلك، فاستخدم جملة Exit Function بعدها مباشرة.

والآن استخدم هذه الدالة كما يلي:

**Dim X As Date**

**X = ShowDate()**

**MsgBox(X)**

أو يمكنك إرسال ناتج الدالة كمعامل للرسالة مباشرة كالتالي:

**MsgBox(ShowDate())**

**ملاحظة:**

لو لم تحدّد نوع القيمة المعادة، بعدم كتابة التعبير As بعد قوسي الدالّة، فسيتم اعتبار القيمة المعادة عامّة من النوع كائن Object.. هذا بالطبع إذا كان خيار التحويل الدقيق غير فعّال Option Strict Off.

**المعاملات**

* + [**ولكن ما هي فائدة المعاملات؟:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\01-%20ولكن%20ما%20هي%20فائدة%20المعاملات.htm)
  + [**بالقيمة ByVal وبالمرجع ByRef:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\02-%20الاستدعاء%20بالقيمة%20وبالمرجع.htm)
  + [**مثال: دالة الترقيم الدولي ISBN:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\03-%20دالة%20الترقيم%20الدولي.htm)
  + [**ملاحظات على استدعاء الدوال والإجراءات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\04-%20ملاحظات%20على%20استدعاء%20الدوال%20والإجراءات.htm)
  + [**الكائناتObjects كمعاملات للإجراءات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\05-%20الكائنات%20كمعاملات%20للإجراءات.htm)
  + [**معاملات الإجراءات المستجيبة للأحداث Event-Handler Arguments:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\06-%20معاملات%20الإجراءات%20المستجيبة%20للأحداث.htm)
  + [**المعاملات الاختياريّة Optional Arguments:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\07-%20المعاملات%20الاختياريّة.htm)
  + [**مصفوفة المعاملات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\08-%20مصفوفة%20المعاملات.htm)
  + [**المعامل المصفوفة:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\09-%20المعامل%20المصفوفة.htm)
  + [**استخدام إجراء كمعامل لإجراء آخر:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\3-%20المعاملات\10-%20استخدام%20إجراء%20كمعامل%20لإجراء%20آخر.htm)

**ولكن ما هي فائدة المعاملات؟:**

إنّ لها فائدة جمّة، فهي تمكّنك من التحكّم في أداء الإجراء.. فمثلا، الإجراء الفرعيّ "عرض التاريخ" يعرض تاريخ اليوم فقط.. لكن افترض أنّك تريد أن تعرض تاريخ يوم آخر، وليكن يوم ميلاد شخص ما، فماذا تفعل؟

في هذه الحالة أنت تحتاج لإرسال معامل لهذا الإجراء لتحديد التاريخ الذي تريده.. انظر لهذا التعديل:

**Sub ShowDate(ByVal BirthDate As Date)**

**MsgBox(BirthDate)**

**End Sub**

والآن لاستخدام هذا الإجراء يمكنك كتابة التالي:

**ShowDate(#2/9/1960#)**

كما يمكنك إرسال القيمة من خلال متغيّر كالتالي:

**Dim MyBirthDate As Date = #2/9/1960#**

**ShowDate(MyBirthDate)**

**دالة الترقيم الدولي ISBN:**

لكلّ كتاب رقم دوليّ International Standard Book Number (ISBN).. افترض الآن أنّك تريد أن تتحقّق من صحّة هذا الرقم.

إن الرقم الدوليّ يتكون من 10 خانات، 9 خانات مخصّصة للرقم، والخانة الأخيرة مخصّصة للتحقّق من الصّحّة.. هذه الخانة يتمّ حسابها كالتالي:

1- يتم ضرب قيمة كل خانة من الخانات التسعة في معكوس ترتيبها (10 – رقم الخانة)، مع ملاحظة أنّ الخانة الموجودة في أقصى اليسار هي الخانة رقم 0.. ثم تُجمع كل النواتج معا.

**ChkSum += (10 - I) \* Val(ISBN.Chars(I))**

لاحظ أنّنا نستخدم نصّا وليس مصفوفة، وإن كنّا قد اتفقّنا من فبل على أنّ النصّ هو مصفوفة حروف.. لهذا استخدمنا الوسيلة Chars للحصول على الحرف الموجود في الخانة الحاليّة (رقم I) من النصّ.

2- يقسم حاصل الجمع على 11.

3- الباقي من عملية القسمة يتم طرحه من 11.. هذه هي قيمة خانة التحقّق من الصّحّة.. ونظرا لأنّ هذه القيمة قد تكون أي عدد من 0 إلى 10، فإنّ العدد 10 يتم تمثيله بالحرف X ، حتّى يتمّ استيعابه في خانة واحدة.

مثال: حساب رقم التحقّق للرقم الدولي ISBN 078212283:

1- 0×10 + 7×9 + 8×8 + 2×7 + 1×6 + 2×5 + 2×4 + 8×3 + 3×2 = 195.

2- 195 ÷ 11 = 17 والباقي 8.

3- 11 – 8 = 3.

إذن فخانة التحقّق هي 3، وبهذا يصير الرقم الدوليّ 0782122833 .

تعال نكتب الدالة التي تحسب هذه الخانة:

**Function ISBNCheckDigit(ByVal ISBN As String) As String**

**Dim I, ChkSum, ChkDigit As Integer**

**For I = 0 To 8**

**ChkSum += (10 - I) \* Val(ISBN.Chars(I))**

**Next**

**ChkDigit = 11 - (ChkSum Mod 11)**

**If ChkDigit = 10 Then**

**Return ("X")**

**Else**

**Return (ChkDigit.ToString)**

**End If**

**End Function**

طبعا يجب أن تُرجع هذه الدالة نصّا، لأنّ خانة التحقّق قد تكون رقما أو القيمة X.. أيضا تستقبل هذه الدالة الرقم الدولي كنصّ، لأنّ الخانات التسع كلها هامّة حتّى لو كان بعضها أصفارا، لهذا لا نستطيع وضع الرقم الدولي في متغيّر رقميّ، لأنّ وجود أصفار على يسار الرقم ليس له معنى ويتم التخلص منها في المتغيّر الرقميّ.

ولاختبار هذه الدالّة، جرّب ما يلي:

**Console.WriteLine(ISBNCheckDigit("078212283"))**

**ملاحظات على استدعاء الدوال والإجراءات:**

- يجب دائما أن توضع المعاملات بين قوسين.

- يجب إرسال العدد الذي تنتظره الدالة من المعاملات، لا أكثر ولا أقلّ.

- يجب إرسال المعاملات بالترتيب الذي تريده الدالة أو الإجراء (كما سيظهر لك في تلميح الشاشة بعد كتابة اسم الدالة أو الإجراء).

- يجب أن تتوافق القيم التي ترسلها للمعاملات، مع القيم التي تقبلها هذه المعاملات، فمثلا: لن يقبل منك VB أن ترسل تاريخا أو نصًّا لمعامل تم تعريفه على أنّه عدد صحيح.

- يمكنك استدعاء الدالة بدون الاستفادة من قيمتها المعادة، تماما كما تستدعي الإجراء الفرعيّ.. فمثلا: يمكنك استدعاء دالة حساب خانة التحقّق كالتالي:

**ISBNCheckDigit("078212283")**

أو كالتالي:

**Call ISBNCheckDigit("078212283")**

ولكنّك في هذه الحالة لن تستفيد شيئا، لأنّك تجاهلت القيمة المعادة من الدالة.. وإن كانت هناك دوال أخرى تقوم بتنفيذ وظيفة أساسيّة، بالإضافة لإرجاع قيمة.. في هذه الحالة يمكنك تجاهل القيمة المعادة إذا لم تكن تعنيك، مكتفيا بتنفيذ الدالة للوظيفة.

- يمكن استخدام الدوالّ في أي عمليّة حسابيّة أو نصّيّة، أو في أي جملة شرطيّة:

**Dim ISBN As String**

**ISBN = "078212283" + ISBNCheckDigit("078212283")**

**If ISBNCheckDigit("078212283") > 3 Then Exit Sub**

كما يمكن استخدام الدوال (قيمها المعادة على وجه الدّقّة) كمعاملات لدوال وإجراءات أخرى:

**MsgBox (ISBNCheckDigit("078212283"))**

**Dim ISBN As String = ISBNCheckDigit("078212283")**

**بالقيمة ByVal وبالمرجع ByRef**

تتساءل طبعا عن سبب وجود الكلمة "بالقيمة" ByVal في تعريف معامل الإجراء.. إنّها تعني ببساطة: أرسل قيمة المتغيّر MyBirthDate لهذا المعامل، لكن لا تسمح للإجراء بتغيير قيمة المتغيّر الأصليّ.

ستنتابك الدهشة: وهل يمكن تغيير متغيّر من داخل الإجراء، رغم أنّه معرّف خارج الإجراء؟.. ألا يتعارض هذا مع موضوع مجال المتغيرات الذي شرحناه من قبل؟

معك حقّ، ولكنّ هذه حالة خاصّة، وستجدها مفيدة في كثير من الأحيان.. تعال نتعرّف على الموضوع أكثر:

* [**بالقيمة ByVal:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\02-%20الاستدعاء%20بالقيمة%20وبالمرجع\1-%20بالقيمة.htm)
* [**بالمرجع ByRef:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\02-%20الاستدعاء%20بالقيمة%20وبالمرجع\2-%20بالمرجع.htm)
* [**فيمَ يفيدنا تعريف المعاملات بالمرجع؟**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\02-%20الاستدعاء%20بالقيمة%20وبالمرجع\3-%20فيمَ%20يفيدنا.htm)
* [**التعبئة Boxing والتفريغ Unboxing:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\02-%20الاستدعاء%20بالقيمة%20وبالمرجع\4-%20التعبئة%20والتفريغ.htm)
* [**معاملات الإخراج Output Parameters:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\02-%20الاستدعاء%20بالقيمة%20وبالمرجع\5-%20معاملات%20الإخراج.htm)

**بالقيمة ByVal:**

في المثال التالي سنعرّف الإجراء الفرعيّ كالتالي:

**Sub ShowDate(ByVal BirthDate As Date)**

**MsgBox(BirthDate)**

**BirthDate = #1/7/1977#**

**End Sub**

والآن جرّب ما يلي، في حدث ضغط أيّ زرّ:

**Dim MyBirthDate As Date = #2/9/1960#**

**ShowDate(MyBirthDate)**

**MsgBox(MyBirthDate)**

شيء طبيعيّ أن تظهر لك الرسالتان وعليهما نفس التاريخ: 2/9/1960، فتغيير قيمة المعامل BirthDate داخل الإجراء الفرعيّ لم يؤثّر على قيمة المتغيّر MyBirthDate.

إنّ المتغيّر ما هو إلا مخزنٌ في الذاكرة.. هذا المخزن له عنوان.

والآن عندما تخبر VB أن يرسل متغيّرا بالقيمة لإجراء، فإنّه يحجز مكانا جديدا في الذاكرة، وينسخ فيه قيمة المتغيّر، ليتمّ استخدامه كمعامل للإجراء.. لهذا فإنّ أيّ تغيير يحدث لهذا المعامل، لا يؤثّر أبدا على المتغيّر الأصليّ (هذا مخزن وهذا مخزن آخر).

هذا الجدول يلخص العمليّة كلّها:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| عنوان الذاكرة  (هذه الأعداد لضرب المثل فقط) | 1001 | 1002 | 1003 | 1004 |
| محتوياته | ؟ | المتغير MyBirthDate | ؟ | المعامل BirthDate |
| قيمته | ؟ | 2/9/1960 | ؟ | 2/9/1960 |
| بعد التغيير | ؟ | 2/9/1960 | ؟ | 1/7/1977 |

**ملاحظة لمبرمجي VB6:**

القيمة الافتراضيّة للمعامل الآن هي ByVal، على عكس ما كان في الماضي، وذلك حتّى يتوافق VB مع باقي لغات البرمجة.. أي أنّه في حالة عدم كتابة ByVal أو ByRef أمام تعريف المعامل، فإنّ VB سيفترض أنّه ByVal، بل وسيكتبها لك تلقائيّا.

**بالمرجع ByRef:**

انظر الآن لهذا المثال:

**Sub ShowDate(ByRef BirthDate As Date)**

**MsgBox(BirthDate)**

**BirthDate = #1/7/1977#**

**End Sub**

والآن جرّب ما يلي في حدث ضغط أيّ زرّ:

**Dim MyBirthDate As Date = #2/9/1960#**

**ShowDate(MyBirthDate)**

**MsgBox(MyBirthDate)**

ستجد أنّ الرسالة الأولى (الخاصة بالإجراء) ستعرض التاريخ 2/9/1960، بينما تعرض الرسالة الثانية التاريخ 1/7/1977!

لقد أدّى تغيير قيمة المعامل BirthDate داخل الإجراء الفرعيّ، إلى تغيير قيمة المتغيّر MyBirthDate!

لماذا حدث هذا؟

حدث لأنّنا استخدمنا كلمة "بالمرجع" ByRef في تعريف معامل الإجراء، ممّا يعني أنّ أيّ تغيير يحدث لهذا المعامل، سيؤثّر على المتغيّر الذي أر سل له قيمته.

ففي حالة تعريف المعامل بالمرجع، لا يحجز VB مكانا جديدا في الذاكرة.. بل يسمح للمعامل بالتعامل مباشرة مع مخزن المتغيّر في الذاكرة.. وبهذا فإنّ أي تعديل في قيمة المعامل من داخل الإجراء، يظهر أثره على المتغيّر (مخزن واحد فقط يتعامل معه الاثنان)!

هذا الجدول يوضّح الأمر أكثر:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| عنوان الذاكرة  (هذه الأعداد لضرب المثل فقط) | 1001 | 1002 | 1003 | 1004 |
| محتوياته | ؟ | المتغير MyBirthDate  والمعامل BirthDate معا! | ؟ | ؟ |
| قيمته | ؟ | 2/9/1960 | ؟ | **؟** |
| بعد التغيير | ؟ | 1/7/1977  (المعامل والمتغيّر تأثّرا معا) | ؟ | **؟** |

طبعا يمكنك أن تتخيّل أن إرسال المعاملات بالمرجع أسرع بكثير من إرسالها بالقيمة.. فمثلا يفضّلُ إرسال المصفوفات كمعاملات بالمرجع، وليس بالقيمة، وذلك لأنّ نسخ المصفوفة ـ خاصّة إذا كانت كبيرة ـ في كل استدعاء يستهلك الوقت ومساحة الذاكرة.

**فيمَ يفيدنا تعريف المعاملات بالمرجع؟**

قلنا إنّ الدالة لها قيمة معادة Return Value، يمكنك استقبالها كناتجٍ للعمليّة التي قامت بها الدالة.. ولكن ماذا لو كنت تتوقّع أكثر من ناتج: اثنين أو ثلاثة أو أكثر؟

في هذه الحالة يمكنك أن تحوّل وظيفة معاملات الإجراء من مدخلات للإجراء، إلى مخرجات، وذلك بتعريفها بالمرجع ByRef، مع وضع النتائج بها من داخل الإجراء.. انظر المثال التالي ليتضح لك الأمر أكثر:

افترض أنّك تريد أن تكتب إجراء واحدا، يحسب لك ناتج جمع عددين وناتج ضربهما، بدلا من كتابة ذلك في دالتين إحداهما للجمع والأخرى للضرب.. استخدم الكود التالي:

**Private Sub AddAndMultiply(ByVal X As Integer, ByVal Y As Integer, \_**

**ByRef Sum As Integer, ByRef Mult As Integer)**

**Sum = X + Y**

**Mult = X \* Y**

**End Sub**

والآن لاستخدام هذا الإجراء، جرّب ما يلي:

**Dim SumResult As Integer, MultResult As Integer**

**AddAndMultiply(3, 2, SumResult, MultResult)**

**MsgBox (SumResult) 'ستعرض الرسالة 5**

**MsgBox(MultResult) ' ستعرض الرسالة 6**

طبعا اتضح لك الأمر.. حمدا لله.

وطبعا لاحظت أنّ X و Y هما معاملا إدخال Input Parameters، بينما Sum و Mult هما معاملا إخراج Output Parameters.. وإن كان من الممكن استخدامهما كمعاملي إخراج.. انظر كيف سنعدّل الدالة السابقة، لتقوم بنفس الوظيفة، لكن بمعاملين اثنين فقط:

**Private Sub AddAndMultiply(ByRef X\_Sum As Integer, \_**

**ByRef Y\_Mult As Integer)**

**X\_Sum = X\_Sum + Y\_Mult**

**Y\_Mult = X\_Sum \* Y\_Mult**

**End Sub**

ولاستدعاء هذا الإجراء، استخدم الكود التالي:

**Dim SumResult As Integer = 3, MultResult As Integer =2**

**AddAndMultiply(SumResult, MultResult)**

**MsgBox (SumResult) 'ستعرض الرسالة 5**

**MsgBox(MultResult) ' ستعرض الرسالة 6**

ولكن من الواضح أنّ الكود بهذا الشكل صار غامضا ومربكا وصعب الفهم!.. لهذا فالطريقة الأولى ذات الأربع (أقصد المعاملات!!) أفضل.

ما أحبّ أن ألفت نظرك إليه هنا، هو أنّ قواعد مجال المتغيّرات التي أوضحناها من قبل ما زالت سارية كما هي، فلا ترتبك بسبب قدرتك على تغيير قيمة متغيّر خارجيّ من داخل إجراء، عن طريق تعريف المعامل بالمرجع ByRef، فإنّ ذلك لا يعني أبدا أنّ المتغيّر الخارجيّ قد صار ضمن نطاق الإجراء الفرعيّ.. فمثلا لا يمكنك استخدام المتغيّر SumResult من داخل الإجراء AddAndMultiply، رغم أنّ لديك طريقة غير مباشرة لتغيير قيمته، باستخدام المتغيّر Sum.. كما لا يوجد ما يمنعك من تعريف متغيّرين خارجيين يحملان نفس اسمي متغيري الإجراء، فلكلّ نطاقه المعزول.. انظر لما يلي:

**Dim Sum As Integer, Mult As Integer**

**AddAndMultiply(3,2, Sum, Mult)**

**MsgBox (Sum) 'ستعرض الرسالة 5**

**MsgBox(Mult) ' ستعرض الرسالة 6**

**التعبئة Boxing والتفريغ Unboxing:**

طيب.. نريد الآن أن نتساءل:

هل سيقبل VB إرسال قيمة لمعامل مرجعيّ، مثل:

**AddAndMultiply(3,2, 4, 2)**

حيث أرسلنا 4 و 2 كقيمتين، بدلا من متغيّرين لاستقبال القيمة العائدة؟

في الواقع لن يعترض VB، وسيعمل كلّ شيء على ما يرام.. وإليك ما يحدث:

1- يقوم VB بإجراء عمليّة تسمّى "التعبئة" Boxing:

حيث يحجز مساحة في نوع من الذاكرة يسمّى "الكومة" Heap (وهي خاضعة لتحكّم جامع القمامة Garbage coolector، وبها يتمّ تخزين الكائنات المرجعيّة)، حيث يضع VB فيها القيمة المرسلة، ثمّ يرسل عنوان هذه الذاكرة للمعامل المرجعيّ، ليعمل كلّ شيء بطريقة طبيعيّة.

2- ثمّ يقوم VB بإجراء عمليّة تسمّى "التفريغ" Unboxing:

فبد انتهاء الإجراء، يقوم VB بتحرير المساحة المؤقّتة التي حجزها في الذاكرة.. وطبعا ستضيع القيمة الموجودة بها، فلا يوجد متغيّر يستقبلها في هذه الحالة.

جميل.. VB متفاهم، ويؤدّي لك معظم العمل آليّا.. وهذا مفيد لك، في الحالات التي لا تحتاج فيها للقيم العائدة.. ولا يشترط أن تكون الدالة خاصة بك.. قد تكون دالة جاهزة من دوال اللغة.

لكن.....

حذار.. فهذه العمليّة مرهقة، ويمكن أن تؤثّر على كفاءة وسرعة البرنامج، خاصة إذا كان استدعاء الإجراء يتمّ من داخل جملة تكراريّة Loop تنفّذ مئات المرّات، أو من داخل إجراء ارتداديّ Recursive Procedure يستدعي نفسه مرارا ([وسنتعرّف عليه في هذا الفصل](file:///C:\Users\Administrator\Desktop\vb.net\8-%20الإجراءات%20الارتدادية.htm)).

فإذا كنت تتعامل مع إحدى هاتين الحالتين، فمن الأفضل أن تعرّف متغيّرا، وترسله كمعامل، حتّى لو لم تكن تحتاج لقيمته العائدة من الدالة.

جدير بالذكر أنّ عمليات التعبئة والتفريغ، تحدث أيضا عند وضع الأرقام والنصوص والقيم المختلفة في الكائنات Objects أو العكس.. مثل:

**Dim age As Integer = 5**

**Dim oAge As Object = age ' عملية تعبئة**

**age = oAge ' عملية تفريغ**

**معاملات الإخراج Output Parameters:**

من كلّ ما سبق يتّضح أنّ المعاملات يمكن أنت تستخدم:

1- للإدخال Input:

وذلك لإرسال البيانات للإجراء.. ويتمّ تعريفها باستخدام كلمة ByVal.

2- للإدخال والإخراج معا Input/Output:

وذلك لإرسال البيانات للدالة واستقبالها منها.. ويتمّ تعريفها باستخدام كلمة ByRef.

جميل.. ألا تشعر أنّ هناك شيئا ناقصا؟

ألا تعتقد أنّه ربّما توجد معاملات للإخراج فقط Output؟

في الواقع نعم.. ولكنّ تعريفها يحتاج لاستخدام إحدى السمات Attributes التي يمنحها لنا إطار العمل.

ابدأ باستيراد فضاء الاسم التالي:

**Imports System.Runtime.InteropServices**

الآن يمكنك أن تعرّف معاملات الإخراج باستخدام السمة <Out(  )>، كالتالي:

**Private Sub AddAndMultiply(ByVal X As Integer, ByVal Y As Integer, \_**

**<Out(  )>** **ByRef Sum As Integer, \_**

**<Out(  )> ByRef Mult As Integer)**

**Sum = X + Y**

**Mult = X \* Y**

**End Sub**

فما هو التأثير الذي تتوقّع الحصول عليه؟

بصراحة لا شيء!

إنّ استخدام كلمة Out في C# مثلا، يجبرك على أن يكون المعامل مرجعيّا، ويجبرك على أن تضع له قيمة داخل الدالة.

أمّا في VB.Net، فلن تشعر بأيّ اختلاف، بل إنّ بإمكانك أن تعرّف المعامل بالقيمة في نفس الوقت الذي تعرّفه فيه على أنّه معامل إخراج!!!!!!!!!!

عموما.. كان هذا للعلم بالشيء.. فلا تشغل به بالك الآن كثيرا!

**الكائناتObjects كمعاملات للإجراءات:**

تشتمل الكائنات على الخلايا Classes، والنماذج Forms، والمصفوفات Arrays من جميع الأنواع، والسجلات Structures، وما شابه.

ويمكنك أن تمرّر هذه الكائنات كمعاملات للإجراءات، ولكنّ ذلك سيتم دائما بالمرجع ByRef، حتّى لو ذكرت أنت في التعريف أنّه بالقيمة فقط ByVal، وذلك توفيرا للوقت ومساحة الذاكرة (يستثنى من ذلك المصفوفات، فمن الممكن تمريرها بالقيمة).. معنى هذا أنّ أي تغيير في قيمة هذه الكائنات خلال الإجراء، سيظهر تأثيره في الكائن الأصليّ خارج الإجراء.

مثال:

افتح مشروعا جديدا، وأضف الإجراء التالي قبل جملة End Class في الخليّة الخاصة بالنموذج:

**Private Sub H(ByVal Frm As Windows.Forms.Form)**

**Frm.Height /= 2 ' تقليل ارتفاع النموذج إلى النصف**

**Frm = Nothing**

**End Sub**

أضف زرا إلى النموذج، واكتب به ما يلي:

**Call H(Me)**

شغل البرنامج، واضغط الزر.. ستجد أن ارتفاع النموذج قد تقلّص إلى النصف.. اضغط الزر مرّة أخرى (لو كان ما يزال ظاهرا على النموذج).. ستجد أنّ ارتفاع النموذج قد تقلص إلى النصف مرة أخرى.

إذن فالإجراء قد أثّر على الكائن (النموذج) رغم أنّنا مررناه إليه بالقيمة!

ولكن.....

هل يعني هذا أنّه لا يوجد أيّ فارق بين طريقتي الاستدعاء في حالة الكائنات؟

بل يوجد..

فالسطر الأخير في الإجراء يجعل قيمة المعامل "لا شيء" Nothing.. ولكنّ هذا لا يؤثر على المتغيّر الأصليّ، ولا يدمّر نسخة النموذج المعروضة.

انظر للجدول التالي، ليوضّح لك الأمر أكثر:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| عنوان الذاكرة | 1001 | 1002 | 1003 | 1004 |
| محتوياته | نسخة من النموذج | المتغيّر الذي يشير إلى النموذج (وهو هنا Me) | المعامل Frm | ؟ |
| قيمته | قيم خصائص النموذج: ارتفاعه، عرضه.... إلخ. | 1001  (عنوان النموذج في الذاكرة) | 1001  (عنوان النموذج بالذاكرة) | ؟ |
| بعد تنفيذ الإجراء | تتغيّر قيم بعض الخصائص | 1001 | 1001 | ؟ |
| بعد تغيير قيمة المعامل | لا تغيير | 1001 | 0  (Nothing) | ؟ |

هل اتضح لك الأمر أم لا؟

إنّ كلّ شيء يسير كما اعتدنا عليه تماما عند تمرير المعامل بالقيمة.. ولكنّ الخدعة هنا هي أنّ المتغيّر والمعامل في حالتنا هذه يحويان مرجعا لكائن (نموذج).. ولهذا فإنّهما يؤثران في نفس الكائن، رغم أنّ كلا منهما مستقلّ عن الآخر في الذاكرة.

ويتّضح هذا حينما يتمّ تغيير قيمة المعامل، بجملة ممّا يأتي:

**Frm = Nothing ' تحرير المتغيّر، لكي لا يشير إلى شيء**

**Frm = New Form1() ' وضع نسخة جديدة من النموذج في المتغيّر**

**Frm = Frm2 ' ربط المتغيّر بنسخة موجودة من نموذج آخر**

ففي أيّ حالة من هذه الحالات، يصبح المتغيّر في وادٍ، والمعامل في واد آخر، حيث يتصرّف كلّ منهما بدون أيّ ارتباط بالآخر على الإطلاق.

جميل.. ولكن ماذا عن تمرير الكائن بالمرجع؟

في هذه الحالة يمكن تغيير المرجع نفسه، ووضع نسخة أخرى (أو كائن آخر أو لا شيء) في المتغيّر كالتالي:

**Private Sub H(ByRef Frm As Windows.Forms.Form)**

**Frm.BackColor = Color.Red**

**Frm = New Form1()**

**Frm.Show()**

**Frm.Text = "نموذج 2"**

**End Sub**

حيث وضعنا نسخة جديدة من النموذج في المعامل المرجعيّ Frm، وقمنا بعرضها في هذا الإجراء، مع تغيير عنوانها، للتمييز بينها وبين النموذج الأصليّ.

تعال نجرّب ذلك.. ضع زرّا على النموذج، واكتب به ما يلي:

**Private Sub Button1\_Click(ByVal sender As System.Object, ByVal e \_**

**As System.EventArgs) Handles Button1.Click**

**Dim F As Form1 = Me**

**F.Text = "نموذج 1"**

**H(F)**

**F.BackColor = Color.Yello**

**End Sub**

**تحذير:**

حاذر أن تنفّذ الكود السابق في حدث تحميل النموذج، فعند تنفيذه سيستدعي الإجراء، الذي سيقوم بتعريف نسخة جديدة من النموذج، ممّا سيستدعي حدث تحميل هذه النسخة، فيتمّ استدعاء الإجراء مرّة أخرى، والذي سيعرّف نسخة جديدة من النموذج..... وهكذا داليك، إلى أن يتمّ تدمير الذاكرة وحدوث خطأ!

فحاذر من مثل هذه العمليّات الدائريّة غير المنتهية.

والآن جرّب الكود في حدث ضغط الزرّ..

سيتمّ عرض النموذج الأوّل.. اضغط الزرّ.. سيتمّ عرض النموذج الثاني وتلوينه بالأحمر!

إذن فالتغيير الذي حدث في الإجراء قد أثّر على المتغيّر F، وحوّله ليشير إلى النموذج الثاني بدلا من النموذج الأوّل.

وهذا الجدول يوضّح لك الأمر أكثر، في حالة تمرير الكائن بالمرجع:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| عنوان الذاكرة | 1001 | 1002 | 1003 | 1004 |
| محتوياته | نسخة من النموذج | المتغيّر F  والمعامل Frm.. معا! | ؟ | نسخة جديدة من النموذج |
| قيمته | قيم خصائص النموذج: ارتفاعه، عرضه.... إلخ. | 1001  (عنوان النموذج في الذاكرة) | ؟ | Nothing |
| بعد تنفيذ السطر الأوّل من الإجراء | يتغيّر لون النموذج إلى الأحمر | 1001 | ؟ | Nothing |
| بعد تنفيذ السطر الثاني من الإجراء | لا تغيير | 1004  (عنوان النسخة الثانية من النموذج في الذاكرة.. يشير إليها المتغيّر والمعامل معا!) | ؟ | قيم خصائص النموذج: ارتفاعه، عرضه.... إلخ. |
| بعد تنفيذ الأمر:  F.BackColor = Color.Yello | لا تغيير | 1004  (لاحظ أنّه لم يعد هناك الآن تأثير للمعامل على هذه الخانة في الذاكرة، بعد انتهاء الإجراء الفرعيّ.. الآن لدينا المتغيّر F فقط، للتعامل مع هذه الخانة) | ؟ | يتغيّر لون النموذج إلى الأصفر |

أمّا لو غيّرت كلمة ByRef في تعريف المعامل Frm إلى ByVal، فستجد أنّ النموذج الأوّل هو الذي تمّ تلوينه باللون الأحمر، وأنّ التغيير الذي حدث لمعامل الدالة لم يصبه بأيّ أذى.

ولكي يتضح لك الأمر أكثر، جرّب الكود التالي:

**Private Sub H(ByVal Frm As Windows.Forms.Form)**

**Frm.BackColor = Color.Red**

**Frm = New Form1()**

**Frm.Show()**

**Frm.Text = "نموذج 2"**

**Frm.BackColor = Color.Yello**

**End Sub**

**Private Sub Button1\_Click(ByVal sender As System.Object, ByVal e \_**

**As System.EventArgs) Handles Button1.Click**

**Dim F As Form1 = Me**

**F.Text = "نموذج 1"**

**H(F)**

**F.Hight /=2**

**End Sub**

ستجد أنّ النموذج الأوّل قد تمّ تلوينه باللون الأحمر (مرتبط بالمتغيّر)، بينما الثاني بالأصفر (مرتبط بالمعامل).. ففي البداية يشير المتغيّر Frm إلى مرجع النموذج الأوّل في الذاكرة.. لهذا تمّ تغيير النموذج الأوّل إلى اللون الأحمر.

بعد ذلك تغيّر مرجع المتغيّر Frm ليشير إلى النموذج الجديد الذي تمّ تلوينه باللون الأصفر (دون أن يتأثّر المتغيّر F في إجراء ضغط الزرّ بهذا التغيير، فسيظلّ يشير إلى النموذج الأوّل).

والجدول التالي يوضح لك ما حدث:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| عنوان الذاكرة | 1001 | 1002 | 1003 | 1004 |
| محتوياته | نسخة من النموذج | المتغيّر F | المعامل Frm | نسخة جديدة من النموذج |
| قيمته | قيم خصائص النموذج: ارتفاعه، عرضه.... إلخ. | 1001  (عنوان النموذج بالذاكرة) | 1001 | Nothing |
| بعد تنفيذ السطر الأوّل من الإجراء | يتغيّر لون النموذج إلى الأحمر | 1001 | 1001 | Nothing |
| بعد تنفيذ السطر الثاني من الإجراء | لا تغيير | 1001 | 1004  (عنوان النسخة الثانية من النموذج في الذاكرة) | قيم خصائص النموذج: ارتفاعه، عرضه.... إلخ. |
| بعد تنفيذ السطر الأخير من الإجراء | لا تغيير | 1001 | 1004 | يتغيّر لون النموذج إلى الأصفر |
| بعد تنفيذ الأمر:  F.Hight /=2 | يتقلّص ارتفاع هذا النموذج إلى النصف | 1001 | Nothing  (انتهى عمر المتغيّر، لأنّه خاص بالإجراء الفرعيّ فقط، ويموت بعد انتهائه) | لا تغيير |

**معاملات الإجراءات المستجيبة للأحداث Event-Handler Arguments:**

لن يكون غريبا هنا أن نشير إلى معاملات الإجراءات المستجيبة للأحداث، فهي إجراءات فرعيّة عاديّة تماما إلا في أمرين: نوعية المعاملات، وتعبير يستجيب Handles الذي يوضّح الأحداث التي يستجيب لها الإجراء.

إنّ هناك معاملين مشتركين في كل مستجيبات الأحداث: "المرسِل" sender، و e (وهما نوعان من الكائنات).. انظر مثلا لتعريف المستجيب لحدث الضغط:

**Private Sub Button1\_Click(ByVal sender As System.Object, \_**

**ByVal e As System.EventArgs) Handles Button1.Click**

**End Sub**

وفيه يقوم المعامل sender، بحمل معلومات عن الكائن الذي تسبّب في إطلاق الحدث، من الكائنات التي تم ذكرها في تعبير Handles.. وفي التعريف السابق ذكرنا حدث الضغط الخاص بالزر Button1 فقط، ممّا يعني بالضرورة أنّ المعامل sender لن يكون إلا Button1.

ولو أضفت الجملتين التاليتين للإجراء:

**Console.WriteLine(sender.ToString)**

**Console.WriteLine(sender.GetType)**

فسيظهر لك في شاشة المخرجات السطران التاليان:

System.Windows.Forms.Button, Text: Button1

System.Windows.Forms.Button

ما يهمّنا أكثر هو المعامل الثاني e، فهو يحوي كل المعلومات التي تريدها للاستجابة للحدث والتفاعل معه.

وهذا المعامل يمتلك بعض الخصائص، ولكنّها تختلف من مستجيب للحدث إلى آخر، تبعا لنوعية الحدث والأداة التي أطلقته.

وعليك أن تختبر هذه الخصائص لكل حدث تكتبه، بكتابة e.، حيث ستظهر لك قائمة الأعضاء، والتي من خلالها ستتضح لك كل إمكانيات هذا المعامل، وذلك إلى أن نشرح ذلك بالتفصيل في فصل لاحق إن شاء الله.

**المعاملات الاختياريّة Optional Arguments:**

كثيرا ما تكون قيم بعض المعاملات متوقّعة، لدرجة أنّك تسأم من تكرارها عند كل استدعاء الإجراء، متسائلا في سآمة: ألا توجد طريقة تخلّصني من هذا التكرار؟

نعم توجد وسيلة، فإنّ بإمكانك أن تعرّف أيّ عدد من معاملات الإجراء على أنّها اختيارية Optional يحقّ لك عند استدعاء الإجراء ألا ترسلها.

ولكن كيف سيعرف الإجراء قيمة المعامل في هذه الحالة؟

لا تخشَ شيئا، فستقوم بتحديد القيمة الافتراضيّة للمعامل، ليتم استخدامها في حالة عدم إرساله.

انظر لهذا المثال، حيث سنقوم بتعريف دالة لحساب الأسس، بمنحها العدد الذي يمثّل الأساس، والعدد الذي يمثّل الأسّ، وسنجعل هذا الأخير اختياريا، بحيث لو لم نرسله عند الاستدعاء، تكون قيمته الافتراضية 2، مما يعني حساب مربع العدد:

**Function Exps(No As Double, Optional ByVal Ex As Double = 2) \_**

**As Double**

**Return (No ^ Ex)**

**End Function**

والآن يمكنك تجربة كل التعبيرات التالية، فكلّها ممكنة:

**Dim X As Double = Exps(2, 4) ' الناتج 16**

**X = Exps(X,X)**

**X = Exps(2, X)**

**X = Exps(X) ' لم نرسل المعامل الثاني، لهذا ستعتبره الدالة 2**

**X = Exps(X,2) ' هذه الجملة مماثلة للجملة المختصرة السابقة**

ويمكن تعريف أكثر من معامل اختياري، ولكنها يجب أن تكون الأخيرة في ترتيب المعاملات، حيث لا يمكن أن يكون هناك معامل اختياري، ثم يتبعه معامل مطلوب Required.. تعالَ نعدّل تعريف الدالّة السابقة لنجعل المعامل الأوّل اختياريا أيضا، بحيث لو لم يتم إرساله، تكون قيمته الافتراضيّة هي 10.

**Function Exps(Optional No As Double = 10, \_**

**Optional ByVal Ex As Double = 2) As Double**

**Return (No ^ Ex)**

**End Function**

والآن يمكنك استخدام تعبيرات كالتالية:

**Dim X As Integer = Exps(3, 3) ' الناتج 27**

**X = Exps(4) ' الناتج 16**

**X = Exps() ' الناتج 100**

ولكن ماذا لو أردت إرسال المعامل الثاني وعدم إرسال الأوّل؟

في هذه الحالة اترك مكانه فارغا، واستخدم الفاصلة لتحفظ موضعه:

**X = ( , 3) ' الناتج 1000**

أو يمكنك استخدام "المعاملات المسمّاة" Named Arguments، وذلك بكتابة اسم المتغيّر مع قيمته عن طريق استخدام الرمز ":=" كالتالي:

**X = (Ex := 3) ' الناتج 100**

**X = (No := 3) ' الناتج 9**

**عامّةً:**

يمكنك استخدام المعاملات المسمّاة حتّى مع المعاملات غير الاختياريّة، وفائدة ذلك هي قدرتك على لخبطة ترتيب المعاملات التي ترسلها للدالة، ما دمت قد ذكرت اسم كل منها بجوار قيمته، حيث يقوم VB في هذه الحالة بمطابقة القيم المرسلة عن طريق الأسماء وليس عن طريق الموضع.

**مصفوفة المعاملات:**

لو أردت أن تكتب دالة لحساب المتوسّط الحسابيّ فستواجهك مشكلة: لكم عدد ستحسب هذه الدالة المتوسّط الحسابيّ؟.. اثنين؟.. ثلاثة؟.. إنّني أريدها عامّة، بحيث يمكنني أن أرسل لها أيّ عدد من المعاملات، لحساب متوسطها الحسابيّ.

لا تشغل بالك: لدينا حل رائع:

إن VB يمنحك الكلمة الأساسيّة "مصفوفة معاملات" ParamArray، التي تمكّنك من إرسال أيّ عدد من المعاملات للإجراء.. تعالَ نستخدمها لتعريف دالة المتوسّط الحسابيّ:

**Function Avg(ByVal ParamArray Numbers() As Integer) As Double**

**Dim I As Integer, Sum As Integer**

**For I = 0 To Numbers.GetUpperBound(0)**

**Sum += Numbers(I)**

**Next**

**Return Sum / Numbers.Length**

**End Function**

والآن جرّب ما يأتي في حدث ضغط أي زر:

**MsgBox(Avg(2, 3, 4, 5, 7)) ' ستعرض الرسالة 4.2**

**MsgBox(Avg(2, 3, 4, 5, 7, 3)) ' ستعرض الرسالة 4**

وهكذا تلاحظ أنّ بإمكانك إرسال أيّ عدد من المعاملات لهذه الدالّة.

وهناك طريقة أخرى لاستدعاء هذه الدالة، بإرسال مصفوفة لها كمعامل:

**Dim No() As Integer = {2, 3, 4, 5, 7}**

**MsgBox(Avg(No))**

ويمكنك أن ترسل القيمة Nothing بدلا من مصفوفة المعاملات.. أو يمكنك ألا ترسل شيئا على الإطلاق:

**MsgBox(Avg())**

ولكن عليك أن تحاذر من الأخطاء التي قد تنجم عن ذلك، حيث يجب أن تتأكّد أولا إذا كانت هناك خانات في مصفوفة المعاملات أم لا، قبل أن تتعامل معها في كود الإجراء.. ففي حالتنا هذه، لن يمكن حساب الدالة GetUpperBound بينما المصفوفة فارغة، وعليك أن تكتب الكود المناسب لمعالجة هذا الاستثناء.

طبعا لك أن تتخيّل أنّ مصفوفة المعاملات يجب أن تكون آخر معامل في تعريف الإجراء، نظرا لعدم معرفة عدد خاناتها، وأيّ معاملات أخرى يجب أن تظهر قبل هذه المصفوفة.. مثلا:

**Function Avg(ByVal Positive As Boolean, ByVal ParamArray \_**

**Numbers() As Integer) As Double**

**End Function**

**المعامل المصفوفة:**

هناك حلّ آخر لكتابة دالة المتوسّط الحسابيّ، وهو إرسال الأعداد التي تريدها في مصفوفة عاديّة هذه المرّة (بدلا من مصفوفة المعاملات).. جرّب ما يلي:

**Function Avg(ByVal Numbers() As Integer) As Double**

**Dim I As Integer, Sum As Integer**

**For I = 0 To Numbers.GetUpperBound(0)**

**Sum += Numbers(I)**

**Next**

**Return Sum / Numbers.Length**

**End Function**

كل ما حدث هو أن أزلنا كلمة ParamArray من تعريف المعامل Numbers()، وإن كنا تبعنا اسم المعامل بقوسين فارغين، لنعبّر عن كونه مصفوفة مرنة، تستطيع استقبال أيّ مصفوفة بأيّ حجم.

ولكنّ هذا التعديل يستدعي حدوث اختلاف في طريقة استدعاء الدالّة، فستصير كالتالي:

**Dim X() As Integer = {2, 3, 4, 5, 7}**

**MsgBox(Avg(X))**

فلن يعود باستطاعتنا إرسال الأرقام للدالة مباشرة، ولكن علينا وضعها أولا في مصفوفة، ثم تمرير اسم هذه المصفوفة للدالة.

ولكن هناك أهمّيّة كبيرة لهذه الطريقة، ليس فقط أنّك تستطيع وضع المعامل المصفوفة في أيّ موضع في ترتيب تعريف المعاملات، بل لأنّك تستطيع كذلك تعريف أيّ عدد من المصفوفات كمعاملات للإجراء، على حسب احتياجك:

**Function Avg(ByVal X() As String, ByVal Positive As Boolean, \_**

**ByVal Numbers() As Integer) As Double**

**End Function**

ويمكن أن تكون المصفوفة من أيّ نوع، حتّى لو كان كائنا أو سجلا Structure.

**استخدام إجراء كمعامل لإجراء آخر:**

لا غرو إن أدهشك هذا العنوان، سواء أكنت جديدا على البرمجةِ أو كنت من مبرمجي VB6.

نعم: يمكن إرسال إجراء (دالة أو إجراء فرعيّ) كمعامل لإجراء آخر!

إنّ هذا يمنحك مرونةً عالية، بحيث يمكنك كتابة إجراءات عامّة بها أجزاء قابلة للاختلاف.. هذه الأجزاء ستكتب في إجراءات مستقلّة، يتمّ إرسال أسمائها للإجراء الأوّل ليستدعيها.

ولكن كيف يمكن فعل ذلك؟

إنّ من لهم دراية بلغات برمجة أخرى كلغة C++ سيجيبون على الفور: نستخدم المؤشّرات Pointers.

ولكنّ الأمر لا يتمّ بهذه الطريقة في VB.. إنّ VB.Net تقدّم طريقةً أسهل وآمنَ من المؤشّرات.. هذه الطريقة هي استخدام المندوب Delegate!

بدهيٌّ إذن أنّ هذا المندوب يقوم بنفس وظيفة المؤشّر: يمثّل عنوان الإجراء في الذاكرة.

ولتعريف مندوب عن إجراء فرعيّ، استخدم الصيغة التالية:

**Delegate Sub اسم المندوب (معاملات الإجراء)**

ولتعريف مندوب عن دالّة استخدم الصيغة التالية:

**Delegate Function اسم المندوب (معاملات الدالة) As القيمة المعادة**

بعد هذا يمكنك استخدام اسم هذا المندوب كنوع لمعاملات الإجراءات.. تعال نرى ذلك بمثال بسيط:

ابدأ بتعريف المندوب التالي:

**Delegate Sub MsgSub (Msg As String)**

والآن سنستخدمه في تعريف الإجراء التالي:

**Sub ShowMsg(M As MsgSub)**

**Call M("رسالة إيضاحيّة")**

**End Sub**

لاحظ كيف استخدمنا المتغيّر M باعتبار أنّه إجراء له معامل نصّيّ.

ويمكن استدعاء الأجراء الذي يمثّله المندوب بطريقة أخرى، وذلك باستخدام الوسيلة "استدعاء" Invoke، كالتالي:

**Sub ShowMsg(M As MsgSub)**

**M.Invoke("رسالة إيضاحيّة")**

**End Sub**

الآن بقيَ أن نُعرّف بعض الإجراءات التي يمكن أن تصلح كمعاملات للإجراء ShowMsg:

**Sub Show1( S As String)**

**MsgBox (s)**

**End Sub**

**Sub Show2( MessageText As String)**

**MsgBox (MessageText & " أخرى")**

**End Sub**

**Sub Show3( T As String)**

**MsgBox (T & " ثالثة")**

**End Sub**

والآن يمكنك استدعاء الإجراء ShowMsg بأيّ من الطرق التالية:

**ShowMsg(AddressOf Show1)**

**' ستظهر الرسالة "رسالة إيضاحيّة"**

**ShowMsg(AddressOf Show2)**

**' ستظهر الرسالة "رسالة إيضاحيّة أخرى"**

**ShowMsg(AddressOf Show3)**

**' ستظهر الرسالة "رسالة إيضاحيّة ثالثة"**

لاحظ استخدام كلمة AddressOf قبل اسم الإجراء، وذلك حتّى يستطيع VB التفريق بين الدالة المرسلة كمعامل، والدالة التي تُرسل قيمتها المعادة Return Value كمعامل.

أعتقد أنّ الأمر قد استبان لك، وإن كنت لن تشعر بجدوى المثال السابق.. إنّه فقط للإيضاح.. ولكنّك سرعان ما ستكتشف ما لهذه الإمكانيّة من ميزات.. وسأضرب لك مثالا:

افترض أنّك كتبت إجراءً ترسل له مسار أحد المجلّدات Folders، حيث يقوم بالمرور على كلّ ما به من ملفّات، حتّى تلك الموجودة داخل المجّلدات الفرعيّة فيه، ولأيّ مستوى.. إنّ كيفيّة عمل ذلك سنتعرّف عليها في الفصول القادمة بإذن الله.. لهذا فلن نكتب هذا الكود، وسنقتصر على شرح الفكرة.

الآن يجب أن يفعل هذا الإجراء شيئا، فهو لن يمرّ على الملفّات بدون هدف.. يحسب عددها، يحسب مجموع مساحاتها، يحذفها، ينقلها لمجلّد آخر... إلخ.

هنا سيكون من الذكيّ أن نكتب هذا الإجراء كالتالي:

**Delegate Function FolderAction (FilePath As String) As Integer**

**Sub ScanFolder(FolderPath As String, Action As FolderAction) \_**

**As Integer**

**' الكود المناسب للمرور عبر ملفّات المجلّد**

**' Fافترض أنّ اسم الملفّ الحالي موجود في متغيّر اسمه**

**Action (F)**

**End Sub**

الآن يمكنك استخدام هذا الإجراء في تطبيقك وفي أيّ تطبيقات أخرى لأداء أيّ وظيفة على الملفّات، وذلك بتعريف الدوال التي تقوم بهذه الوظائف، وتمرير أسماءها إلى هذا الإجراء.. فمثلا يمكنك أن تعرّف دالة لحذف الملفّ كالتالي:

**Function Delete (Path As String) As Integer**

**' الكود المناسب لحذف الملفّ**

**End Function**

ولاستدعاء هذه الدالة استخدم جملة كالتالية:

**ScanFolder("C:\Folder1", AddressOf Delete)**

وعلى نفس هذا المنوال يمكنك تعريف عشرات الدوال للتعامل مع ملفّات المجلّد: لحساب مجموع أحجامها، لحساب أكبر أو أصغر أو أحدث أو أقدم ملفّ فيها، لتغيير امتدادات الملفّات أو أسمائها.. إلخ.

كلّ ما هو مطلوب منك فقط، هو أن يكون لكلّ هذه الدوال معامل واحد فقط من النوع نصّ String، وقيمة معادة من النوع عدد صحيح Integer.

وبهذا ستستريح من نسخ الكود وتعديله في كلّ مرّة ليناسب الوظيفة الجديدة.

وهناك أمثلة أخرى لا حصر لها يمكن استخدام المندوب Delegate فيها، مثل كتابة دالة تمرّ عبر عناصر شجرة TreeView لأداء أيّ وظيفة، أو تمرّ عبر العناصر الفرعيّة لقائمة Menu لأداء أيّ وظيفة... إلخ.

وسنرى هذا بإذن الله عند التعرّف على هذه المكوّنات في الفصول التالية.

**استدعاء الإجراء باستخدام متغيّر:**

هناك وسيلةٌ أخرى للقيام بعمل المندوب Delegate، وإن كانت تختلف عنه بعض الشيء، وذلك باستخدام الدالة CallByName، التي ظهرت لأوّل مرّة مع صدور نسخة VB6.. ولهذه الدالّة الصيغة التالية:

القيمة\_المعادة = CallByName(اسم الخليّة التي تحتوي على الإجراء, \_

نصّ أو متغيّر نصّيّ يمثّل اسم الإجراء,نوع الإجراء المستدعى, مصفوفة معاملات)

ويأخذ المعامل الثاني واحدة من قيم المرقّم CallType التالية:

|  |  |
| --- | --- |
| Method | استخدم هذا النوع مع الإجراءات والدوال. |
| Get | استخدم هذا النوع عندما تريد قراءة قيمة خاصّيّة. |
| Set | استخدم هذا النوع عندما تريد تغيير قيمة خاصّيّة. |
| Let | لم يعد هذا النوع متاحا في VB.Net. |

ولا بدّ أنّك استنتجت السبب في أنّ المعامل الأخير لهذه الوسيلة مصفوفة معاملات.. طبعا حتّى ترسل معاملات الإجراء الذي تستدعيه.

وطبعا القيمة المعادة من هذه الدالة، هي القيمة المعادة من المعادة من الدالة التي تستدعيها.

ما رأيك أن نعيد كتابة الدالة ShowMsg التي كتبناها في الفقرة السابقة، ولكن باستخدام الاستدعاء بالاسم هذه المرّة؟.. هذه هي:

**Sub ShowMsg(MsgSub As String)**

**CallByName (Me, MsgSub, CallType.Method, "رسالة إيضاحيّة")**

**End Sub**

الآن يمكنك استدعاء هذه الدالّة، مرسلا لها أسماء الإجراءات الثلاثة التي عرّفناها سابقا، كالتالي:

**ShowMsg("Show1") ' ستظهر الرسالة "رسالة إيضاحيّة"**

**ShowMsg("Show2") ' ستظهر الرسالة "رسالة إيضاحيّة أخرى"**

**ShowMsg("Show3") ' ستظهر الرسالة "رسالة إيضاحيّة ثالثة"**

أعتقد أنّك لاحظت الفارق في طريقة الاستدعاء بين هذه الوسيلة وبين استخدام المندوب Delegate.. وأنت حرّ في اختيار الطريقة التي تناسبك أكثر.

**أنواع أخرى للقيم المعادة:**

نفس هذه المرونة تستطيع أن تحصل عليها عند تعريف القيم المعادة من الدوال، فهي ليست مقيّدة بأن تكون أعدادا صحيحة أو نصوصا، بل يمكن كذلك أن تكون مصفوفات أو سجلات Structures أو كائنات من أي نوع.. انظر للمثال التالي:

افترض أنّك تريد كتابة دالة تُنتج لك متتابعة حسابيّة، بمجرّد أن تعطيها رقم البداية ومعدّل الزيادة وعدد الحدود التي تريدها:

**Function Series(Start As Integer, B As Integer, \_**

**N As Integer) As Integer()**

**Dim X(N) As Integer, I As Integer**

**X(0) = Start ' وضع الحدّ الأوّل في خانة المصفوفة الأولى**

**For I = 1 To N - 1**

**' قيمة الخانة الحالية في المصفوفة = قيمة الخانة السابقة + قيمة الزيادة**

**X(I) = X(I - 1) + B**

**Next**

**Return X ' القيمة المعادة مصفوفة**

**End Function**

أعتقد أنّ ما فعلناه واضح: في البداية أضفنا قوسين فارغين بعد نوع الدالة في سطر التعريف As Integer()، لنخبر VB أنّ القيمة المعادة هي مصفوفة.

ثم بعد ذلك عرّفنا مصفوفة موضعيّة داخل الدالة، وملأناها بالأرقام المطلوبة، ثم جعلناها هي القيمة المعادة.

لاحظ مدى مرونة VB.Net، فقد عرّفنا عدد خانات المصفوفة X بقيمة المعامل (المتغيّر) N، ولم يكن مسموحا بمثل هذا من قبل.

والآن استخدم الكود التالي لاستدعاء هذه الدالة:

**Dim S( ) As Integer**

**S = Series(2, 3, 10)**

**' بعد تنفيذ هذه الجملة، ستحتوي المصفوفة على متتابعة حسابية**

**' حدها الأول 2، ومعدل زيادتها 3، وعدد حدودها 10**

ويمكن تحسين كود الدالة السابقة، ليصبح كالتالي:

**Function Series(Start As Integer, B As Integer, N As Integer) \_**

**As Integer()**

**Dim X(N) As Integer, I As Integer**

**For I = 0 To N - 1**

**X(I) = Start**

**Start += B ' الحد التالي للمتتابعة**

**Next**

**Return X ' القيمة المعادة مصفوفة**

**End Function**

**الدوال والسجلات**

هناك علاقات متعدّدةٌ بين الدوالّ والسجلات.. هنا نتعرّف عليها:

* + [**الدوال التي تعيد سجلات Functions Returning Structures:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\6-%20الدوال%20والسجلات\1-%20الدوال%20التي%20تعيد%20سجلات.htm)
  + [**السجلات كمعاملات للإجراءات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\6-%20الدوال%20والسجلات\2-%20السجلات%20كمعاملات%20للإجراءات.htm)
  + [**الإجراءات كأعضاء للسجلات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\6-%20الدوال%20والسجلات\3-%20الإجراءات%20كأعضاء%20للسجلات.htm)
  + [**مشروع السجلات Records Project:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\6-%20الدوال%20والسجلات\4-%20مشروع%20السجلات.htm)

**الدوال التي تعيد سجلات Functions Returning Structures:**

طريقة أخرى للحصول على قيم عديدة معادة من دالة، هي جعل القيمة المعادة سجلا من تعريفك، حيث تستطيع أن تجعل له العدد الذي تريده من الحقول، لنقل كل ما تريد من بيانات.

وهذه الطريقة تتفوّق على جعل القيمة المعادة مصفوفة في كثير من الأحيان، فليس في كل مرّة تكون القيم المعادة من نفس النوع، وحتّى لو كانت من نفس النوع، فليس شرطا أن تدلّ على نفس المعنى والوظيفة.. فمثلا، المصفوفة المعادة في المثال السابق هي متتالية حسابيّة، وهو مناسب تماما، فليس هناك معنى محدّد لكل عنصر في المتتالية.. ولكن الوضع سيختلف عندما تكون القيم المعادة مثلا، هي رقم الطالب، ورقم فصله، ونتيجته في الامتحان... إلخ، فوضع هذه القيم في مصفوفة يقتضي من المبرمج أن يتذكّر معنى ووظيفة كل خانة حتّى يستخدمها في باقي البرنامج، وهو ما يجعل الكود أصعب في القراءة، وأعقد عند إصلاح الأخطاء.

في مثل هذه الحالة إذن، يصبح السجلّ هو الحلّ الأمثل، لأنّ خاناته معرّفة بوضوح، وتجعل الكود أيسر في قراءته وإصلاحه:

**Structure Student**

**Dim No As Integer**

**Dim ClassNo As Integer**

**Dim Grade As Integer**

**End Structure**

والآن يمكن تعريف أيّ دالة للبحث في سجلات الطلاب، بحيث يكون ناتج البحث هو سجل الطالب المستوفي لشروط البحث.. افترض مثلا أنّنا سنكتب دالة للبحث عن الطالب الأوّل في الامتحان:

**Function FirstStudent() As Student**

**Dim S As Student**

**' الكود المناسب للبحث عن الطالب الحاصل على أعلى درجة**

**' S يتم وضع بيانات الطالب في المتغيّر**

**Return S**

**End Function**

ولاستدعاء هذه الدالّة، يمكنك استخدام كود كالتالي:

**Dim FS As Student = FirstStudent()**

**MsgBox (FS.No)**

**MsgBox(FS.Grade.ToString)**

وليس هذا كلّ ما في الأمر، فما زالت لديك إمكانيّة رائعة، للحصول على مصفوفة من السجلات كقيمة معادة.. انظر لهذه الدالة، التي ترجع لنا أسماء كل الطلبة الحاصلين على درجات أعلى من قيمة معيّنة:

**Function StudentGrade(ByVal GreaterThan As Integer) As Student()**

**Dim S() As Student**

**' الكود المناسب للبحث عن الطلاب**

**Return S**

**End Function**

ولاستدعاء هذه الدالة:

**Dim X() As Student = StudentGrade (60)**

**السجلات كمعاملات للإجراءات:**

لقد ذكرنا هذا من قبل: إنّ السجلات هي إحدى الأنواع التي يمكن أن تستخدم كمعاملات للإجراءات الفرعيّة والدوالّ.. افترض أنّنا نريد أن نكتب إجراء لطباعة بيانات طالب معيّن:

**Sub PrintStudentData( ThisStudent As Student)**

**Console.WriteLine (ThisStudent.No)**

**Console.WriteLine(ThisStudent.ClassNo)**

**Console.WriteLine(ThisStudent.Grage)**

**End Sub**

ولاستدعاء هذا الإجراء، استخدم كودا كالتالي، وهو يطبع بيانات الطالب الأوّل:

**Dim FS As Student = FirstStudent()**

**StudentData(Fs)**

أو استخدم ـ كبديل ـ الجملة المختصرة التالية لو أردت:

**StudentData(FirstStudent())**

ولا يتوقّف الأمر عند هذا الحدّ، فإنّ بوُسعك أيضا أن تستخدم مصفوفة من السجلات كمعامل، كما ستفعل عند تعريف الإجراء الذي يطبع أسماء مجموعة من الطلبة:

**Sub PrintStudentsData(Students() As Student)**

**Dim I As Integer**

**For I = 0 To Students.GerUpperBound(0)**

**PrintStudentData(Students(I))**

**Next I**

**End Sub**

وحينما تريد استخدام هذا الإجراء، جرّب الكود التالي لطباعة بيانات الطلبة الحاصلين على درجات أعلى من 50:

**PrintStudentsData (StudentGrade (50))**

لاحظ المرونة الشديدة للغة، فنظرا لأنّ الدالة StudentGrade تُرجع مصفوفة، فإنّ بإمكاننا استخدامها كمعامل للإجراء PrintStudentsData، الذي هو مصفوفة بدوره.

**الإجراءات كأعضاء للسجلات:**

لو عرّفت متغيّرا كالتالي:

**Dim X As Student**

ثم كتبت X. فستظهر لك قائمةٌ تعرض كل حقول هذا السجلّ، وتعرض كذلك بعض وسائل هذا السجلّ، مثل الدالة الجاهزة ToString.

إذن فالسجلّ كباقي أنواع الكائنات، له أعضاء: خصائص، ووسائل (إجراءات).

الطريف في الأمر، أنّ بإمكانك أن تُعرّف أعضاء جديدة للسجلّ، تنفّذ بها ما تشاء من العمليّات على حقوله.. والأمر بسيط: كل ما عليك فعله، هو أن تعرّف إجراءً عاديّا تماما، يتبع نفس قواعد تعريف الإجراءات ومعاملاتها وقيمها المعادة، لكن داخل مقطع السجلّ، بحيث يصبح هذا الإجراء عضوا من أعضاء السجلّ، ويظهر اسمه في قائمة الأعضاء.. تعالَ نعيد تعريف سجل الطالب كمما يلي:

**Structure Student**

**Dim No As Integer**

**Dim ClassNo As Integer**

**Dim Grade As Integer**

**Function StudentName() As String**

**Dim Name As String**

**' No الكود المناسب للبحث عن الطالب في قاعدة بيانات الطلاب بمعلوميّة رقمه**

**Return Name**

**End Function**

**Sub Print()**

**Console.WriteLine(StudentName)**

**Console.WriteLine (No)**

**Console.WriteLine(ClassNo)**

**Console.WriteLine(Grage)**

**End Sub**

**End Structure**

لاحظ قدرتك على استخدام أسماء الحقول مباشرةً بدون أن تكون مسبوقة باسم السجلّ، وذلك لأنّ هذه الحقول مُعرّفة على مستوى السجلّ ككلّ، وتبعا لقواعد مجال المتغيّرات، فإنّها قابلة للاستخدام المباشر داخل مقطع السّجلّ.. نفس الأمر ينطبق على الدوالّ والإجراءات، فإجراء الطباعة Print، يستخدم الدالة StudentName مباشرة، لأنّها معرّفة ضمن نطاق السّجلّ.. أمّا المتغيّرات التي تعرّف داخل أي إجراء تابع للسّجلّ، كالمتغيّر Name في الدالّة StudentName، فلا يمكن استخدامه من أي مكان في السّجلّ، غير الإجراء الذي ينتمي إليه.

والآن يمكنك استخدام تعبيرات كالتالية:

**Dim N As String = FirstStudent().StudentName**

**MsgBox(s)**

**FirstStudent().Print**

والتي يمكن كتابتها في صورة أفضل كالتالي:

**Dim S As Student = FirstStudent ()**

**MsgBox (S.StudentName)**

**S.Print**

وتكفي هنا هذه الفكرة المختصرة عن تعريف أعضاء السجلات، إلى أن نتعمّق في هذا الموضوع أكثر فيما بعدُ إن شاء الله.

**مشروع السجلات Records Project:**

أعتقد أنّ كتابة مشروع نستخدم فيه هذه المفاهيم، سيكون فكرة جيّدة:

افترض أنّ لديك قاعدة بيانات تحتوي على أسماء مجموعة من الشركات وبعض المعلومات عنها.. نريد الآن عرض بيانات هذه الشركات لمستخدم البرنامج.. طبعا لن نستطيع عرضها كلّها معا.. لهذا سنصمّم نموذجا يعرض بيانات شركة واحدة فقط في كلّ مرّة، وعند ضغط زرّ "عرض التالي" View Next، يتمّ عرض بيانات الشركة التالية... وهكذا.



وعندما يصل البرنامج لسجلّ آخر شركة، يعود مرّة أخرى لأوّل سجلّ.

ابدأ بتعريف الأنواع التالية، في الجزء الخاصّ بتعريف متغيّرات النموذج، خارج أيّ إجراء:

**Structure Customer**

**Dim Company As String**

**Dim Manager As String**

**Dim Address As String**

**Dim City As String**

**Dim Country As String**

**Dim CustomerSince As Date**

**Dim Balance As Decimal**

**End Structure**

**Private Customers(8) As Customer ' مصفوفة تحتوي على 9 زبائن**

**Private Cust As Customer ' مخزن مؤقّت لقيم السجلّ الحالي**

**Private CurrentIndex As Integer ' رقم السجلّ الحالي في مصفوفة سجلات الزبائن**

سيقوم الإجراء المستجيب لحدث ضغط زر عرض السجلات بما يلي:

- يتحقّق ممّا إذا كان السجلّ الحاليّ هو آخر السجلات، وذلك بمقارنته بالدالة CountCustomers، التي تحسب عدد الزبائن الموجودين في مصفوفة الزبائن Customers.. فإذا كان السجلّ الحالي هو السجلّ الأخير فعلا، يتم العودة إلى أوّل سجلّ وعرضه، بجعل رقم السجلّ الحالي صفرا.

- يستدعي الدّالّة GetCustomer()، حيث سيمنحها رقم السجلّ، ليحصل منها على السّجلّ الحالي من سجلات الزبائن.

- يستدعي الدالة ShowCustomer لتعرض محتويات هذا السجلّ في اللافتات الموضوعة على النموذج.

- يقوم بزيادة رقم الخانة الحاليّة CurrentIndex، لتشير إلى السجلّ التالي.

هذا هو الإجراء:

**Private Sub Button1\_Click(ByVal sender As System.Object, \_**

**ByVal e As System.EventArgs) Handles Button1.Click**

**If CurrentIndex = CountCustomers() Then CurrentIndex = 0**

**Dim aCustomer As Customer = GetCustomer(CurrentIndex)**

**ShowCustomer(CurrentIndex)**

**CurrentIndex += 1**

**End Sub**

وهذه هي الدوال الثلاثة التي يستدعيها:

**Function CountCustomers() As Integer**

**Return(Customers.Length)**

**End Function**

**Function GetCustomer(ByVal Id As Integer) As Customer**

**Return(Customers(Id))**

**End Function**

**Sub ShowCustomer(ByVal Id As Integer)**

**Dim aCustomer As Customer = GetCustomer(Id)**

**lblCompany.Text = aCustomer.Company**

**lblSince.Text = aCustomer.CustomerSince**

**lblAddress.Text = aCustomer.Address**

**lblCity.Text = aCustomer.City**

**lblCountry.Text = aCustomer.Country**

**lblBalance.Text = aCustomer.Balance**

**End Sub**

بقيَت خطوة هامّة: ملء مصفوفة الزبائن عند تحميل النموذج.. إنّ المنطقيّ أن يتمّ تحميل هذه البيانات من قاعدة بيانات أو من ملفّ.. ولكن نظرا لأنّنا لم نتعرّف على كيفيّة التعامل مع هذين النوعين بعد، فسنضع كلّ بيانات الشركات في مصفوفة، نملؤها في حدث تحميل النموذج.. وسنكتفي هنا بكتابة بيانات سجلين، كمثال على كيفية فعل ذلك، ويمكنك الرجوع للبرنامج كاملا في مجلّد برامج هذا الفصل:

**Private Sub Form1\_Load(ByVal sender As System.Object, \_**

**ByVal e As System.EventArgs) Handles MyBase.Load**

**Dim Cust As Customer**

**Cust.Company = "Bottom-Dollar Markets"**

**Cust.Manager = "Elizabeth Lincoln"**

**Cust.Address = "23 Tsawassen Blvd."**

**Cust.City = "Tsawassen"**

**Cust.Country = "Canada"**

**Cust.CustomerSince = #10/20/1996#**

**Cust.Balance = 33500**

**Customers(0) = Cust**

**' ………………………………**

**' ………………………………**

**Cust = New Customer()**

**Cust.Company = "Wilman Kala"**

**Cust.Manager = "Matti Karttunen"**

**Cust.Address = "Keskuskatu 45"**

**Cust.City = "Helsinki"**

**Cust.Country = "Finland"**

**Cust.CustomerSince = #1/3/2000#**

**Cust.Balance = 2500**

**Customers(7) = Cust**

**Button1\_Click(sender, e) ' استدعاء إجراء عرض السجلات، لعرض السجلّ الأوّل**

**End Sub**

**تعدد التعريفات Overloading**

* + [**مفهومه:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\7-%20تعدد%20التعريفات\1-%20مفهومه.htm)
  + [**ولكن كيف تتعدد التعريفات؟:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\7-%20تعدد%20التعريفات\2-%20كيفيّته.htm)
  + [**الشروط التي يجب مراعاتها عند تعدّد التعريفات:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\7-%20تعدد%20التعريفات\3-%20الشروط%20التي%20يجب%20مراعاتها%20عند%20تعدّد%20التعريفات.htm)

**مفهومه:**

هناك حالات تحتاج فيها أن تعرّف دالة تتعامل مع عدد مختلف من المعاملات.. ربما يبدو لك الحلّ في استخدام مصفوفة المعاملات ParamArray أو المعاملات المصفوفات Array Parameters، أو حتّى تعريف الدالة بأكبر عدد من المعاملات وجعل بعضها اختياريّا.

كل هذه الحلول واردة وجميلة، ولكنّ المشكلة تتفاقم لو كنت تريد تعريف دالة تقبل نفس العدد من المعاملات في كلّ مرّة، ولكنّ أنواع هذه المعاملات تختلف من مرّة لأخرى!

هنا سيبدو لك الحلّ في استخدام متغيّرات عامّة من النوع "كائن" Object، ولكنّه حل لا ننصحك به، فهو سيقلل من سرعة وكفاءة البرنامج، وسيضع على كاهلك أعباء أثقل عند كتابة الكود (مثلا استخدام دالة CType للتحويل بين الأنواع، وكتابة كودًا إضافيّا لمعالجة الأخطاء التي قد تنجم عن إرسال بيانات غير مناسبة).

وطبعا لن أقول لك اكتب دالتين أو ثلاثة، بحيث تختلف كل منها عن الأخرى في الاسم، وفي عدد المعاملات أو في نوع البيانات، فهذه طريقة عقيمة لأقصى درجة، وستجعل من الصعب عليك تذكر أسماء الدوال أثناء كتابة الكود.

إذن فما الحلّ؟

الحلّ هو استخدام مفهوم "تعدّد التعريفات" OverLoading، الذي عن طريقه يمكنك تعريف الدالة الواحدة أي عدد من المرات، بحيث تحمل في كلّ مرّة نفس الاسم، وتختلف في عدد المعاملات أو أنواعها أو تختلف في نوع القيمة المعادة.

وعند كتابة الجملة التي تستدعي الدالة، سيظهر لك تلميح على الشاشة بعد فتح قوس المعاملات، ليعرض لك صيغة الدالة كما أنت معتاد، لكنك ستلاحظ في جانبه الأيسر ظهور تعبير كالتالي "1 Of 4" محصور بين سهمين، أحدهما يشير لأعلى، والآخر لأسفل.. إنّ هذا يعني أنّ هذه الدالة لها أربع صيغ مختلفة، وأنّ الصيغة الحاليّة هي الصيغة الأولى منها.. ولعرض الصيغ الأخرى، اضغط بالفأرة على أيّ من السهمين، أو باستخدام سهمي الأعلى والأسفل من لوحة المفاتيح.



**ولكن كيف تتعدد التعريفات؟:**

يمكنك استخدام الكلمة الأساسيّة Overloads، أمام كل صيغة من صيغ الإجراء، وبذلك تتمكّن من تعرف أكثر من صيغة لها نفس الاسم.

بل أتدري: ليس مطلوبا حتّى استخدام هذه الكلمة.. كلّ ما عليك فعله هو أن تعرّف الصيغ المختلفة بطريقة طبيعيّة تماما!

مثال:

نريد الآن كتابة دالة تبحث عن طالب في قاعدة بيانات، إمّا بمنحها الاسم، أو بمنحها الرقم، أو بمنحها المجموع ورقم الفصل (مع ملاحظة أن عددا كبيرا من الطلبة يمكن أن يشترك في المجموع ورقم الفصل، لهذا سيكون الناتج في هذه الحالة مصفوفة) .. وفي كلّ الأحوال سيكون الناتج من نوع سجلات الطلاب الذي عرّفناه سابقا في هذا الفصل:

**Function FindStudent(Name As String) As Student**

**Dim S As Student**

**' الكود المناسب للبحث عن اسم الطالب في قاعدة البيانات**

**Return S**

**End Function**

**Function FindStudent(No As Integer) As Student**

**Dim S As Student**

**' الكود المناسب للبحث عن رقم الطالب في قاعدة البيانات**

**Return S**

**End Function**

**Function FindStudent(Grade As Integer, ClassNo As Integer) As Student()**

**Dim S() As Student**

**' الكود المناسب للبحث عن كل طلاب هذا الفصل**

**' الحاصلين على هذه الدرجة في قاعدة البيانات**

**Return S**

**End Function**

والآن، جرّب هذه التعبيرات:

**Dim S As Student = FindStudent(100)**

**S = FindStudent ("محمد حمدي غانم")**

**S = FindStudent(50, 3)**

لاحظ أنّك لن تستطيع البحث عن الطالب بالدرجة وحدها، إلا إذا جعلت معامل هذه الصيغة من أي نوع آخر غير العدد الصحيح Integer، وإلا لصارت مماثلة لصيغة البحث عن الطالب بالرقم:

**Private Function FindStudent(Grade As Long)As Student()**

**Dim S() As Student**

**' الكود المناسب للبحث عن كل الطلاب الحاصلين على هذه الدرجة في قاعدة البيانات**

**Return S**

**End Function**

لاحظ كيف عرّفنا مجال الصيغة الأخيرة Private، بينما كل الصيغ الأخرى Public.. صحيح أنّنا لم نكتب كلمة Public في تعريفها، ولكنها افتراضيّة.

**الشروط التي يجب مراعاتها عند تعدّد التعريفات:**

- لا يمكن اعتبار الصيغتين مختلفتين، لمجّرد وجود اختلاف في أسماء المعاملات.

- يجب أن يختلف عدد المعاملات، أو يختلف نوع أحد المعاملات على الأقلّ، من صيغة لأخرى.

- لا يمكن اعتبار صيغتين مختلفتين، لمجرّد اختفاء أحد المعاملات من إحداهما، بينما هو معامل اختياريّ في الأخرى.

- لا يمكن تعريف إحدى الصيغ كدالة والأخرى كإجراء.

- لا يمكن اعتبار صيغتين مختلفتين، لمجرّد اختلاف القيمة المعادة من إحداهما عن الأخرى (وهو ما يسبّب لي شخصيا الكثير من التضايق، وإن كان من الممكن حلّ ذلك بجعل القيمة المعادة كائنا والأمر لله).

- يمكن اعتبار الصيغتين مختلفتين حتّى لو كانت كل معاملاتهما متماثلة في النوع والعدد، إذا كان أحد المعاملات مصفوفة والآخر لا.

- لا يمكن اعتبار صيغتين مختلفتين، لمجرّد اختلاف مجال إحداهما عن الأخرى، كأن تكون إحداهما خاصة Private والأخرى عامّة Public.

- إذا كانت لديك صيغتان من الدالة مختلفتان في معاملاتهما، فإنّ بإمكانك أن تجعل مجال كل منهما مختلفا، كأن تجعل إحداهما خاصة Private والأخرى عامّة Public.

**الإجراءات الارتدادية Recursive Procedures**

* + [**استدعاء الإجراء لنفسه:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\8-%20الإجراءات%20الارتدادية\0-%20استدعاء%20الإجراء%20لنفسه.htm)
  + [**تنفيذ جملة تكراريّة بإجراء ارتداديّ:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\8-%20الإجراءات%20الارتدادية\1-%20تنفيذ%20جملة%20تكراريّة%20بإجراء%20ارتداديّ.htm)
  + [**حساب مجموع مصفوفة ارتداديّا:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\8-%20الإجراءات%20الارتدادية\2-%20حساب%20مجموع%20مصفوفة%20ارتداديّا.htm)
  + [**دالة المضروب Factorial:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\8-%20الإجراءات%20الارتدادية\3-%20دالة%20المضروب.htm)
  + [**الترتيب السريع Quick Sort:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\8-%20الإجراءات%20الارتدادية\4-%20الترتيب%20السريع.htm)
  + [**البحث الثنائيّ Binary Search:**](file:///C:\Users\Administrator\Desktop\vb.net\B-%20أساسيّات%20اللغة\04-%20مسار%20البرنامج%20وأجزاؤه\8-%20الإجراءات%20الارتدادية\5-%20البحث%20الثنائيّ.htm)

**استدعاء الإجراء لنفسه:**

هل يمكن أن يستدعي الإجراء نفسه؟

نعم بالتأكيد.. لا يوجد ما يمنع ذلك!

إنّ هذه الطريقة تسمّى البرمجة الارتداديّة (التكراريّة) Recursive Programming، وهي مفيدة جدا، وتختصر كتابة الكود.

والصيغة العامّة لهذا الإجراء هي كالتالي:

**Sub XX(معاملات الإجراء)**

**A. If شرط التوقّف Then Return**

**B.بعض الكود على معاملات الإجراء لأداء وظيفته**

**C.استدعاء الإجراء بقيم معاملات جديدة**

**Call XX(قيم المعاملات الجديدة)**

**D.ربما يكون هناك أي كود آخر يناسبك**

**End Sub**

ويمكن بالطبع أن يكون الإجراء دالّة ذات قيمة معادة.

ويمكن أن تضع الخطوة C قبل الخطوة B في الصيغة السابقة، على حسب ما يناسب وظيفة برنامجك.. وكثيرا ما تكون هاتان الخطوتان داخل جملة تكراريّة Loop.

ولكن كيف يستدعي الإجراء نفسه؟

1- يقوم VB بتخزين قيم جميع متغيرات الإجراء في مخزن يسمّى "الرصّة" Stack، وذلك لأنّ الاستدعاء الجديد للإجراء سيفسد القيم الحالية للمتغيّرات، ونحن ما زلنا نحتاجها لحين عودتنا للإجراء الأصليّ مرّة أخرى.

2- يستدعى VB الإجراء بالطريقة التقليديّة تماما.

3- يستمرّ تكرار الخطوتين السابقتين لمرات عديدة، وفي كلّ مرّة يتم حفظ متغيرات النسخة الحاليّة من الإجراء في الرصّة.

4- عندما يصل VB لشرط التوقّف ينتهي تنفيذ النسخة الحاليّة من الإجراء، ويعود VB للنسخة السابقة، حيث يستعيد قيم كل متغيراتها من الرصّة، ويواصل تنفيذ السطر رقم D (التالي لسطر استدعاء البرنامج لنفسه).

5- يتم تكرار الخطوة 4، حتّى يصل VB إلى النسخة الأولى التي بدأت كل هذه المتاهة، حيث يستعيد قيم متغيراتها من الرّصّة، ويواصل تنفيذها بطريقة عاديّة.

إنّ هذه الطريقة تحتوي على عيب خطير، هو أنّها مرتبطة بحجم الرصّة Stack، حيث كثيرا ما تؤدّي البرمجة الارتداديّة إلى استهلاك كل مساحة الرصّة، وبالتالي يحدث خطأ ويتم إغلاق التطبيق!

لهذا فمن المهمّ أن تتأكّد دائما من وجود شرطٍ دقيقٍ يستطيع أن يوقف عمليّة استدعاء الإجراء لنفسه (الخطوة A) ، حتّى لا تستمرّ هذه العمليّة إلى ما لا نهاية، فتستهلك المساحة المخصّصة في الذاكرة ويحدث خطأ يُنهي البرنامج.. عامّة اطمئن، فبينما كان مبرمجو VB6 محرومين من استخدام هذه التقنية، فقد كانت في VB6 تستهلك الرصّة بسرعة عجيبة، فإنّ بإمكانك الآن أن تستخدم هذه التقنية مرتاحا، ولملايين المرّات من الاستدعاءات!

طبعا ما زال الأمر مربكا بالنسبة لك.. تعالَ نرى بعض الأمثلة، وسيتّضح لك كلّ شيء.

**تنفيذ جملة تكراريّة بإجراء ارتداديّ:**

افترض أنّك تريد كتابة إجراء يعرض عددا من الرسائل، كلّ منها تحمل رقمها.. الطبيعيُّ أن تكتب هذا الإجراء كالتالي:

**Sub Msg(No As Integer)**

**Dim C As Integer**

**For C = 1 To No**

**MsgBox("رسالة رقم " + CStr(C))**

**Next**

**End Sub**

ما رأيك إذن لو كتبنا هذا الإجراء بطريقة ارتداديّة؟

ها هو ذا:

**Sub Msg(No As Integer)**

**' شرط التوقّف**

**If No = 0 Then Exit Sub**

**' استدعاء الإجراء لنفسه**

**Msg(No -1)**

**' وظيفة البرنامج**

**MsgBox("رسالة رقم " + CStr(No))**

**End Sub**

تعالَ نتتبع هذا البرنامج.. افترض أنّنا استدعيناه بجملة كالتالية:

**Msg(3)**

الجدول التالي يوضّح خطوات التنفيذ:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| رقم الاستدعاء | قيمة المتغيّر No | شرط التوقّف | استدعاء تالي | عرض الرسالة | انتهاء الإجراء |
| 1 | 3 | غير متحقّق | نعم |  |  |
| 2 | 2 | غير متحقّق | نعم |  |  |
| 3 | 1 | غير متحقّق | نعم |  |  |
| 4 | 0 | متحقّق.. عودة للإجراء السابق |  |  |  |
| 3 | 1 |  |  | رسالة رقم 1 | نعم.. عودة للإجراء السابق. |
| 2 | 2 |  |  | رسالة رقم 2 | نعم.. عودة للإجراء السابق. |
| 1 | 3 |  |  | رسالة رقم 3 | نعم.. انتهى الاستدعاء الارتداديّ. |

لاحظ أنّ الإجراء يستدعي نفسه قبل أن يعرض رسالته، لهذا ستظهر الرسائل بترتيب معاكس لترتيب استدعاء الإجراء الارتداديّ لنفسه (آخر إجراء يعرض أوّل رسالة، وأوّل إجراء يعرض آخر رسالة).. ولو أردت عكس هذا الترتيب، فغيّر موضع جملة عرض الرسالة، ليتمّ تنفيذها قبل استدعاء الإجراء لنفسه.

لا أدري لماذا أشعر بأنّك ما زلتَ غير مستسيغ للإجراءات الارتداديّة!.. حسنا.. افتح مشروعا جديدا، وأضف تعريف الإجراء Msg للنموذج.. ضع زرّا على النموذج، واكتب فيه السطر التالي:

**Msg(3)**

اضغط بالفأرة على الهامش الرماديّ الأيسر لمحرّر الكود بجوار السطر السابق، أو اضغط F9 من لوحة المفاتيح بينما مؤشّر الكتابة موجود في السطر السابق.. سيضع VB علامة توقّف Break Point على هذا السطر، وسيتوقّف عنده قبل تنفيذه.

شغّل البرنامج واضغط الزرّ.. سيتوقّف VB عند سطر استدعاء الإجراء.. اضغط زرّ F8 من لوحة المفاتيح.. سينفّذ VB سطرا واحدا من الكود، وستجد أنّ التنفيذ قد انتقل إلى أوّل سطر في الإجراء Msg.. حلّق بالفأرة فوق اسم المتغيّر No.. سيظهر لك تلميح يخبرك أنّ قيمته 3.. استمرّ في ضغط F8 وتابع مسار التنفيذ وقيم المتغير No.. أعتقد أنّ ذلك كفيل بجعلك تستوعب الأمرَ تماما.

**حساب مجموع مصفوفة ارتداديّا:**

هل تذكر كيف كنّا نحسب مجموع الأعداد الموجودة في مصفوفة؟

كنّا نكتب الكود التالي:

**Function Sum(No() As Integer) As Integer**

**Dim I, S As Integer**

**For I = 0 To UBound(No)**

**S += No(I)**

**Next**

**Return S**

**End Function**

الآن سنكتب نفس الدالة، ولكن بطريقة ارتداديّة:

**Function Sum(No() As Integer, I As Integer) As Integer**

**' شرط التوقّف هو الوصول إلى نهاية المصفوفة**

**If I = UBound(No) Then**

**Return No(I)**

**Else ' الخطوة التي ستتكرّر دائما**

**' اجمع الخانة الحاليّة على مجموع الخانات التالية لها**

**Return No(I) + Sum(No, I + 1)**

**End If**

**End Function**

تعالَ نتتبّع هذا الإجراء عند استدعائه كالتالي:

**Dim X() As Integer = {1, 2, 3, 4, 2}**

**MsgBox(Sum(X, 0))**

ولمن لم يحبّوا الجدول السابق، فسنشرح الخطوات هنا بالكلمات.. وإن كنت أنصح من يحاول فهمها، أن يضع نقطة توقّف Break Point على سطر استدعاء الإجراء، ويتتبّع شرح الخطوات عمليّا، بضغط F8 خطوةً بخطوة:

- نظرا لأنّ I = 0، إذن فشرط التوقّف غير متحقّق.. لهذا سيقفز التنفيذ للجملة الارتدادية، حيث سيستدعى الإجراء نفسه بقيمة جديدة للمعامل I = 1، وسينتظر الإجراء القيمة المعادة.

- نظرا لأنّ I = 1، إذن فشرط التوقّف غير متحقّق.. لهذا سيقفز التنفيذ للجملة الارتدادية، حيث سيستدعى الإجراء نفسه بقيمة جديدة للمعامل I = 2، وسينتظر الإجراء القيمة المعادة.

- نظرا لأنّ I = 2 (شرط التوقّف غير متحقّق)، فسيقفز التنفيذ للجملة الارتدادية، حيث سيستدعى الإجراء نفسه بقيمة جديدة للمعامل I = 3، وسينتظر الإجراء القيمة المعادة.

- نظرا لأنّ I = 3 (شرط التوقّف غير متحقّق)، فسيقفز التنفيذ للجملة الارتدادية، حيث سيستدعى الإجراء نفسه بقيمة جديدة للمعامل I = 4، وسينتظر الإجراء القيمة المعادة.

- نظرًا لأنّ I = 4 فقد وصلنا لنهاية المصفوفة.. الآن صار شرط التوقّف متحقّقا.. لهذا سيعيد الإجراء قيمة الخانة الخامسة في المصفوفة ( = 2) ويتمّ الخروج منه.

- سيعود التنفيذ للخطوة الرابعة، حيث ستكون القيمة المعادة = [2 + قيمة الخانة الرابعة في المصفوفة ( = 4)] أي 6.

- سيعود التنفيذ للخطوة الثالثة، حيث ستكون القيمة المعادة = [6 + قيمة الخانة الثالثة في المصفوفة ( = 3)] أي 9.

- سيعود التنفيذ للخطوة الثانية، حيث ستكون القيمة المعادة = [9 + قيمة الخانة الثانية في المصفوفة ( = 2)] أي 11.

- سيعود التنفيذ للخطوة الأولى، حيث ستكون القيمة المعادة = [11 + قيمة الخانة الأولى في المصفوفة ( = 1)] أي 12.

- سيعود التنفيذ لسطر استدعاء الدالة (من خارجها)، حيث ستظهر رسالة عليها الناتج النهائيّ (12).

ويمكن تحسين الدالّة السابقة، بحيث لا نحتاج للمعامل I، فبدلا من أن نجعله معاملا، يمكن أن نعرّفه كمتغيّر دائم القيمة Static، بحيث يظلّ محتفظا بقيمته حتّى بعد أن ينتهي الإجراء.. وبهذا نستطيع تغيير قيمته من داخل الدالّة نفسها، كالتالي:

**Function Sum(No() As Integer) As Integer**

**Static I As Integer**

**' شرط التوقّف هو الوصول إلى نهاية المصفوفة**

**If I = UBound(No) Then**

**Return No(I)**

**Else**

**Dim S As Integer**

**'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*استدعاء الإجراء لنفسه \*\*\*\*\*\*\*\*\*\*\***

**' يجب زيادة رقم الخانة قبل استدعاء الدالة مرّة أخرى**

**I + = 1**

**S = Sum(No)**

**' يجب إنقاص رقم الخانة عند العودة من الاستدعاء مرّة أخرى**

**I -= 1**

**'\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Return No(I) + S**

**End If**

**End Function**

كما يمكن تطوير الدالة السابقة، بحيث لا نحتاج نهائيّا للمعامل I .. والفكرة تقوم على جمع أوّل خانة من المصفوفة على مجموع مصفوفة جديدة تحتوي على باقي خانات المصفوفة الأصليّة.. طبعا هذا المجموع الجديد يحتاج لاستدعاء الإجراء لنفسه، حيث سيتمّ تكرار نفس العمليّة مرارا، إلى أن نصل إلى مصفوفة تحتوي على خانة واحدة فقط.. انظر للكود التالي:

**Function Sum(ByVal No() As Integer) As Integer**

**Dim L As Integer = No.Length - 1**

**If L = 0 Then**

**Return No(0)**

**Else**

**Dim NextNos(L - 1) As Integer**

**' نسخ كل خانات المصفوفة ما عدا الخانة الأولى**

**Array.Copy(No, 1, NextNos, 0, L)**

**' الناتج هو ناتج جمع الخانة الحاليّة مع مجموع باقي المصفوفة**

**Return No(0) + Sum(NextNos)**

**End If**

**End Function**

وإن كنت أنصحك بتجنّب فكرة إنشاء مصفوفة جديدة ونسخ القيم إليها داخل الإجراءات الارتداديّة، لأنّ ذلك سيلتهم الكثير من مساحة الذاكرة وسيبطئ من سرعة التنفيذ.. وسنرى ذلك بالتفصيل عند شرح إجراء الترتيب الثنائي.

على كلٍّ، كان المثالان السابقان للإيضاح فقط، ولكنّهما لا يُعتبران عمليّين، ولا تتّضح فيهما عبقريّة البرمجة الارتداديّة.. سيتّضح لك ذلك حينما تريد أن تكتب برنامجا يحسب لك مجموع أحجام كلّ الملفّات الموجودة في مجلّد، بما في ذلك تلك الموجودة في مجلداته الفرعيّة.. وستتّضح لك عندما تريد المرور عبر كلّ عناصر الشجرة TreeView.. وغير ذلك من الأمثلة التي سنقابلها في حينها في الفصول القادة بإذن الله.

وحتّى لا تشعر بعدم الرضا، سأريك مثالا جيّدا للدالة الارتداديّة وهو في ذات الوقت بسيط.. هذا المثال هو دالة المضروب.. ثمّ سنكمل مشروع Arrays الذي بدأناه في الفصل السابق، حيث سنكتب كود الترتيب السريع Quick Sort والبحث الثنائي Binary Search.

**دالة المضروب Factorial:**

تعرف من الرياضيّات أنّ:

مضروب 5 = 5 × 4 × 3 × 2 × 1.

وبملاحظة بسيطة يمكنك أن تستنتج أنّ:

مضروب 5 = 5 × (4 × 3 × 2 × 1) = 5 × مضروب 4.

مضروب 4 = 4 × (3 × 2 × 1) = 4 × مضروب 3.

مضروب 3 = 3 × (2 × 1) = 3 × مضروب 2.

مضروب 2 = 2 × (1) = 2 × مضروب 1.

مضروب 1 = 1 × مضروب 0.

مضروب 0 = 1.

وكصيغة عامّة، فإنّ مضروب العدد (ن) = ن × مضروب (ن – 1).

طبعا لاحظت أنّ خير ما يمكن أن يمثّل هذه الدالة هو إجراء ارتداديّ كالتالي:

**Function Factorial(n As Integer) As Double**

**If n = 0 Then ' شرط التوقّف**

**Factorial = 1**

**Else**

**Factorial = n \* Factorial(n - 1)**

**End If**

**End Function**

وأعتقده أنّه أوضح من أن يحتاج للشرح!

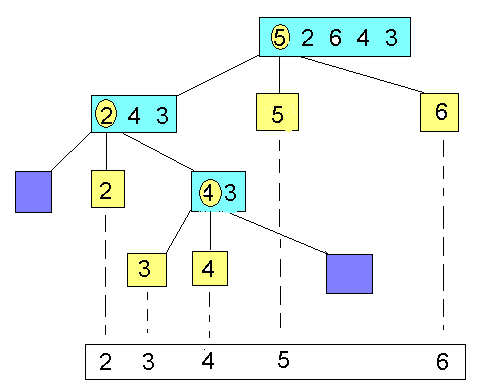
**الترتيب السريع Quick Sort:**

رأينا في الفصل السابق كيف نرتّب المصفوفة بطريقة الفقّاعة Bubble Sort، التي تتسّم بكثرة عمليّات المقارنة.. والآن سنرى طريقة أخرى لترتيب المصفوفة.

وتتلخّص هذه الطريقة في أنّنا سنأخذ الخانة الأولى في المصفوفة كمرجع، ونقارن بها باقي الخانات.. فالخانات التي تحتوي على قيم أصغر من قيمة الخانة الأولى سنضعها في مصفوفة مؤقّتة (سنسمّيها المصفوفة الصغرى)، وباقي الخانات نضعها في مصفوفة مؤقّتة أخرى (سنسمّيها المصفوفة الكبرى).. ثمّ سنرتّب كلا من المصفوفتين بنفس الطريقة.. وهكذا...

في النهاية سننقل المصفوفة المؤقّتة الصغرى إلى بداية المصفوفة، تليها الخانة المرجعيّة، تليها المصفوفة الكبرى.

وحتّى يتّضح الأمر، تعال نجرّب هذه الطريقة لترتيب مصفوفة تحتوي على الأرقام (5، 2، 6، 4، 3):



إنّ المستطيلات الملوّنة باللون السماويّ في الرسم السابق، هي المصفوفات التي سيتمّ استدعاء إجراء الترتيب السريع لترتيبها.. وطبعا المصفوفة العليا هي المصفوفة الأصليّة، أمّا الأخريات فهي أجزاء منها، سيستدعي الإجراء نفسه لترتيبها.

أمّا الخانات الصفراء، فهي الخانات الناتجة من الترتيب.. والخانات الزرقاء هي خانات فارغة لن تؤثّر في شيء.

لاحظ أنّ الترتيب تمّ على ثلاث خطوات فقط (ثلاث مستطيلات سماويّة)، في حين أنّ ترتيب نفس هذه المصفوفة بطريقة الفقّاعة تمّ على ستّ خطوات (راجع الفصل السابق).

تعال نكتب كود هذا الترتيب.. لاحظ أنّنا أسمينا المصفوفة الصغرى LeftArr، والمصفوفة الكبرى RightArr.. وحتّى نعرف عدد خانات كلّ منهما، استخدمنا المتغيّر LAC (اختصارا لتعبير LeftArr Counter) ليعبّر عن عدد خانات المصفوفة الصغرى، والمتغيّر RAC (اختصارا لتعبير RightArr Counter) ليعبّر عن عدد خانات المصفوفة الكبرى، وذلك بدلا من أن نستخدم الدالة UBound أكثر من مرّة داخل الإجراء.. وسنبدأ الإجراء بخانة واحدة في كلّ من المصفوفة الصغرى والكبرى، ثمّ سنضيف إليهما الخانات عند الحاجة:

**Sub QuickSort(ByRef Arr() As String)**

**Dim LeftArr(0), RightArr(0) As String**

**Dim LAC, RAC As Integer**

**Dim C As Integer**

**'----------------------- ملء المصفوفة الصغرى والكبرى---------------------------**

**For C = 1 To UBound(Arr)**

**If Arr(0) > Arr(C) Then ' انقل الخانة إلى المصفوفة الصغرى**

**' يجب زيادة خانة على المصفوفة الصغرى،**

**' مع الاحتفاظ بقيم خاناتها السابقة**

**ReDim Preserve LeftArr(LAC)**

**LeftArr(LAC) = Arr(C)**

**' بعد إضافة الخانة يجب زيادة المتغيّر المعبّر عن طول المصفوفة بواحد**

**LAC += 1**

**Else ' انقل الخانة إلى المصفوفة الكبرى**

**ReDim Preserve RightArr(RAC)**

**RightArr(RAC) = Arr(C)**

**RAC += 1**

**End If**

**Next**

**'-------------------------------------استدعاء الإجراء لنفسه------------------------------**

**' شرط التوقّف عن ترتيب المصفوفة الصغرى هو أن تتكون فارغة أو بها خانة واحدة**

**If LAC > 1 Then QuickSort(LeftArr)**

**' شرط التوقّف عن ترتيب المصفوفة الكبرى هو أن تتكون فارغة أو بها خانة واحدة**

**If RAC > 1 Then QuickSort(RightArr)**

**'------------------نقل المصفوفتين المرتّبتين إلى المصفوفة الأصليّة ---------------------**

**' تأكّد إذا كانت المصفوفة الصغرى تحتوي على خانات**

**If LAC > 0 Then**

**' ضع الخانة الأولى (المرجعيّة) بعد المصفوفة الصغرى**

**'LAC ونحن نعرف طول المصفوفة الصغرى من المتغيّر**

**Arr(LAC) = Arr(0)**

**' انسخ المصفوفة الصغرى إلى بداية المصفوفة الأصليّة**

**LeftArr.CopyTo(Arr, 0)**

**End If**

**' إذا كانت المصفوفة الكبرى تحتوي على خانات**

**' (فانسخها إلى نهاية المصفوفة الأصليّة، (بعد الخانة التي بها الخانة المرجعيّة**

**If RAC Then RightArr.CopyTo(Arr, LAC + 1)**

**End Sub**

أعتقد أنّ الإجراء واضح.. شيءٌ واحد فقط يجب أن يلفت نظرك: أين ناتج هذا الإجراء؟.. كيف سنحصل على المصفوفة المرتّبة؟

إنّ الترتيب سيتمّ على المصفوفة المرسلة مباشرة، وذلك لأنّنا عرّفنا معامل هذا الإجراء مرجعيّا ByRef، وبذلك سيؤثّر أيّ تغيير نحدثه في هذا المعامل على المصفوفة الأصليّة.. إنّ ذلك أفضل وأسرع بالطبع، فليس مستحبّا أن نمرّر المصفوفات بالقيمة في إجراء يستدعى نفسه مئات المرّات.

وعلى هذا يمكننا استدعاء هذا الإجراء بجملة كالتالية:

**QuickSort (TextBox1.Lines)**

وعندما ينتهي الإجراء، ستصبح مصفوفة سطور مربّع النصّ Lines مرتّبة.

جربّ الإجراء.. حاول كذلك مقارنته بالبحث بالفقّاعة.. وحتّى يظهر لك الفارق، انسخ محتويات مربّع النصّ، والصقها أكثر من مرّة (لا ترفع أصبعيك عن زري Ctrl+V لفترة طويلة)، حتّى تحصل على عدد ضخم من السطور.

ستواجهك مفاجأة مدهشة: فبينما يتمّ تنفيذ الترتيب بالفقّاعة في ثانيتين أو ثلاثة، يستهلك الترتيب السريع دقيقتين أو أكثر!!

نتيجة غير متوقّعة على الإطلاق!.. ولكنّ هناك ما يبرّرُها.. إنّ السبب في ذلك هو استخدام جملة Redim Preserve.. ففي كلّ مرّة نحاول إضافة خانة للمصفوفة، يقوم VB بحجز مساحة جديدة في الذاكرة، ونسخ المصفوفة القديمة إليها، وحذف المصفوفة السابقة، لتصبح المساحة المحجوزة هي المصفوفة الجديدة.. هذه العمليّة تحدث داخل جملة تكراريّة في إجراء يُستدعى لآلاف المرّات، ممّا يعنى أنّها تُستدعى لملايين المرّات!!.. إنّ دقيقتين وقت سريع جدّا بالنسبة للإرهاق الذي سبّبناه لـ VB!!

إذن فما هو الحلّ؟

بسيطة: نحجز للمصفوفتين الصغرى والكبرى طولا مبدئيّا مناسبا، وليكن مساويا لطول المصفوفة الأصليّة، فنحن واثقون أنّ أيّا من المصفوفتين الصغرى والكبرى لن يزيد طولها عن ذلك.

ولن نقلق بخصوص الطول الفعليّ للمصفوفة الصغرى أو الكبرى، فنحن بالفعل نستخدم المتغيّرين LAC و RAC لمعرفة عدد العناصر الموجودة في كلّ منهما.

ثمّ بعد أن ينتهي ملء المصفوفتين، نعيد تغيير حجميهما، بحيث نزيل الخانات الفارغة من نهايتيهما إن وُجدت.

وبهذا بدلا من تنفيذ جملة تغيير الحجم ReDim Preserve بعدد خانات المصفوفة، سيتمّ تنفيذها مرّتين فحسب في كلّ استدعاء للإجراء.

هذا هو الإجراء المعدّل:

**Sub QuickSort(ByRef Arr() As String)**

**Dim L As Integer = UBound(Arr)**

**' المصفوفة الصغرى والكبرى لهما نفس طول المصفوفة الأصليّة مبدئيّا**

**Dim LeftArr(L), RightArr(L) As String**

**Dim LAC, RAC As Integer**

**Dim C As Integer**

**For C = 1 To L**

**If Arr(0) > Arr(C) Then**

**' لم نعد نحتاج لجملة تغيير حجم المصفوفة**

**LeftArr(LAC) = Arr(C)**

**LAC += 1**

**Else**

**' لم نعد نحتاج لجملة تغيير حجم المصفوفة**

**RightArr(RAC) = Arr(C)**

**RAC += 1**

**End If**

**Next**

**' إعادة تغيير حجم المصفوفتين لإزالة الخانات الزائدة**

**' تذكّر أن رقم الخانة الأخيرة في المصفوفة أصغر من طول المصفوفة بـ 1**

**ReDim Preserve LeftArr(LAC - 1)**

**ReDim Preserve RightArr(RAC - 1)**

**If LAC > 1 Then QuickSort(LeftArr)**

**If RAC > 1 Then QuickSort(RightArr)**

**If LAC Then**

**Arr(LAC) = Arr(0)**

**LeftArr.CopyTo(Arr, 0)**

**End If**

**If RAC Then RightArr.CopyTo(Arr, LAC + 1)**

**End Sub**

والآن لو جرّبت هذا الإجراء المعدّل، فستجده في سرعة الصاروخ!

ولكنّ هناك تعديلا آخر يمكن القيام به لتسريع إجراء البحث السريع ليصبح اسمه على مسمّى!

ما رأيك لو تخلّصنا نهائيّا من تعريف المصفوفتين الصغرى والكبرى ومن الوقت المهدر في نقل البيانات إليهما؟.. أعتقد أنّ هذا سيوفّر الوقت ومساحة الذاكرة، خاصّة وأن هذه العمليّات تحدث مئات المرّات مع استدعاء الإجراء لنفسه.

ولكن ما البديل للمصفوفة الصغرى والكبرى؟

البديل هو أن نجري العمليّات على المصفوفة الأصليّة، مع استخدام مؤشّرين St و En، يحددان بداية المصفوفة الفرعيّة ونهايتها.. فمثلا في أوّل مرّة سنتعامل مع المصفوفة من الخانة رقم صفر إلى آخر خانة.. وبعد ذلك نتعامل مع جزء من المصفوفة يعتبر هو المصفوفة الصغرى، يبدأ من الخانة رقم صفر إلى الخانة رقم 10 مثلا... وهكذا...

تعال نرى كيف تتمّ هذه الطريقة.. افترض أنّ لدينا نفس المصفوفة السابقة ونريد ترتيبها:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LS** | **I** |  |  |  |
| 5 | 2 | 6 | 4 | 3 |

في البداية سنتعامل مع المصفوفة كلّها (St = 0، و En = 4).

سنأخذ الرقم 5 مرجعا لنا.. وسيكون لدينا مؤشّران:

- آخر أصغر رقم LS وستكون قيمته مساوية لخانة البداية St مبدئيّا.

- والعدّاد الذي يمرّ عبر خانات المصفوفة I، وسيبدأ في الخانة التالية لخانة البداية.

سنبدأ بمقارنة الخانة المرجعيّة بقيمة الخانة رقم I.. ونظرا لأنّ الرقم 2 أصغر من الرقم 5، فسنفعل الآتي:

- نزيد LS بواحد، ليشير إلى الخانة رقم 1.

- نُبادل قيمتي الخانتين رقمي I و LS، وذلك حتّى ننقل العدد الأصغر للنصف الأوّل من المصفوفة.. ونظرا لأنّ I و LS هنا متساويان فلن يحدث أيّ شيء.

بعد ذلك سيزيد I بمقدار 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **LS** | **I** |  |  |
| 5 | 2 | 6 | 4 | 3 |

نظرًا لأنّ 5 أصغر من 6، فلن نفعل شيئا.. سنكتفي بزيادة I.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **LS** |  | **I** |  |
| 5 | 2 | 6 | 4 | 3 |

ولكنّ الرقم 5 أكبر من الرقم 4.. إذن فسنقوم بالآتي:

- نزيد LS بواحد، ليشير إلى الخانة رقم 2.

- نبادل قيمتي الخانتين رقمي I و LS.

ثمّ نزيد I بواحد.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **LS** |  | **I** |
| 5 | 2 | 4 | 6 | 3 |

نظرًا لأنّ 5 أكبر من 3، فسنقوم بنفس الخطوتين المعهودتين:

- نزيد LS بواحد، ليشير إلى الخانة رقم 3.

- ننقل قيمتي الخانتين رقمي I و LS.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | **LS** | **I** |
| 5 | 2 | 4 | 3 | 6 |

الآن وصل I إلى نهاية المصفوفة.. في هذه اللحظة يشير LS إلى آخر رقم من الأرقام الأصغر من 5 في المصفوفة.. الآن كلّ ما علينا هو تبادل قيمتي الخانة المرجعيّة وخانة LS معا.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **المصفوفة الصغرى** | | | **LS** | **المصفوفة الكبرى** |
| 3 | 2 | 4 | 5 | 6 |

تلاحظ الآن أنّ كلّ الخانات التي تسبق الخانة LS بها أرقام أقلّ من 5 (المصفوفة الصغرى)، والخانات التي تليها بها أرقام أكبر من 5 (المصفوفة الكبرى).

الآن وبمنتهى البساطة سنستدعي الإجراء مرّة أخرى لكلّ من هاتين المصفوفتين.. المصفوفة الصغرى (St = 0، و En = 2)، والمصفوفة الكبرى (St = 4، و En = 4).. ولكنّ لأنّ المصفوفة الكبرى تتكوّن من خانة واحدة، فلن يتمّ استدعاء الإجراء لها.

تعال نرى ما سيحدث في المصفوفة الصغرى:

|  |  |  |
| --- | --- | --- |
| **LS** | **I** |  |
| 3 | 2 | 4 |

نظرا لأنّ 3 أكبر من 2، فستتم زيادة LS، ثم لن تؤثّر عمليّة التبادل في شيء.. بعد ذلك نزيد I.

|  |  |  |
| --- | --- | --- |
|  | **LS** | **I** |
| 3 | 2 | 4 |

نظرا لأنّ 3 أصغر من 4 فلن يحدث شيء.

وصلنا لنهاية الخانات، ويجب أن نبادل الخانة المرجعيّة مع الخانة رقم LS.

|  |  |  |
| --- | --- | --- |
| **المصفوفة الصغرى** | **LS** | **المصفوفة الكبرى** |
| 2 | 3 | 4 |

تلاحظ أنّ كلا المصفوفتين الصغرى والكبرى عبارة عن خانة واحدة.. لقد انتهى الترتيب.

تعال نكتب هذا الإجراء:

**Sub QuickSort(ByRef Arr() As String, Optional ByVal St \_**

**As Integer = 0, Optional ByVal En As Integer = -1)**

**If En = -1 Then En = UBound(Arr)**

**Dim I As Integer, LS As Integer = St, Temp As String**

**For I = St + 1 To En**

**' لو غيّرت علامة (أصغر من) إلى (أكبر من) لتمّ الترتيب من الأكبر للأصغر**

**If Arr(I) <= Arr(St) Then**

**LS += 1**

**' تبادل قيمتي الخانتين، إذا كانا بالفعل خانتين مختلفتين**

**If LS <> I Then**

**Temp = Arr(LS)**

**Arr(LS) = Arr(I)**

**Arr(I) = Temp**

**End If**

**End If**

**Next**

**' تبادل قيمتي الخانة المرجعيّة مع آخر أصغر رقم**

**If LS <> St Then**

**Temp = Arr(LS)**

**Arr(LS) = Arr(St)**

**Arr(St) = Temp**

**End If**

**' استدعاء الإجراء لنفسه**

**' لن نستدعي ترتيب المصفوفة الصغرى أو الكبرى إلا إذا كان بها خانتان على الأقلّ**

**If LS - St > 1 Then QuickSort(Arr, St, LS - 1)**

**If En - LS > 1 Then QuickSort(Arr, LS + 1, En)**

**End Sub**

لاحظ أنّنا نحتاج لإرسال بداية ونهاية المقطع الذي نرتّبه من المصفوفة (المصفوفة الصغرى أو الكبرى) في كلّ استدعاء ارتدادي.. ولكن عند استدعاء هذا الإجراء لترتيب المصفوفة الأصليّة، فإنّنا لا نحتاج لذكر هذين المعاملين، لهذا فقد جعلناهما اختياريّين Optional، بحيث لو لم يتمّ إرسالهما تكون قيمة St = 0، وقيمة En تساوي -1.. ولكن لماذا -1؟

إنّ ذلك يعود إلى عدم قدرتنا على كتابة جملة كالتالية في تعريف الدالة:

**Optional En As Integer = Ubound(Arr)**

لهذا جعلنا القيمة الافتراضيّة لهذا المعامل أيّ رقم لا يمكن أن يصلح كرقم خانة في المصفوفة (مثل الأعداد السالبة)، وكتبنا جملة شرط في أوّل الإجراء، بحيث إذا وجدنا قيمته المتغيّر En تساوي -1، نجعل قيمته مساوية لرقم الخانة الأخيرة في المصفوفة.

وبهذا نستطيع استدعاء هذا الإجراء كالتالي:

**QuickSort (TextBox1.Lines)**

هذا بالإضافة لقدرتك على اختيار جزء فقط من المصفوفة لترتيبه:

**QuickSort (TextBox1.Lines, 1, 4)**

الآن لو جرّبت إجراء الترتيب السريع في صيغته النهائيّة، فستجده أسرع بالفعل من الترتيب بالفقّاعة، حتّى إنك تستطيع استخدامه لترتيب سطور مربّع نصّ يحتوي على 5200 سطر في أقلّ من ثانية، في حين يفعلها الترتيب بالفقّاعة في حوالي 7 ثواني!

وستجد الكود الكامل للترتيب السريع في مشروع Arrays في مجلّد برامج الفصل السابق.

**البحث الثنائيّ Binary Search:**

رأينا كيف نبحث عن نصّ في المصفوفة بالمرور عبر خاناتها واحدة تلو أخرى، ومقارنتها بنصّ البحث.. هذه الطريقة عقيمة، وقد تتسبب في بطء شديد للبرنامج.

تعالَ نرى طريقة أسرع بكثير من تلك الطريقة.

والفكرة كلّها تتلخّص في الآتي:

لو كانت لدينا مصفوفة مرتّبة، فسنقارن نصّ البحث بقيمة الخانة الموجودة في منتصف المصفوفة.. فلو كان النصّ أكبر، إذن فلا بدّ أنّه يوجد في النصف الأخير من المصفوفة.. ولو كان أصغر، إذن فلا بد أنّه موجود في النصف الأوّل من المصفوفة.

في أيّ من الحالتين، سنأخذ نصف المصفوفة المناسب ونبحث فيه بنفس الطريقة.. معنى هذا أنّنا اختصرنا نصف المقارنات المطلوبة، لأنّنا أهملنا أحد نصفي المصفوفة.

وبدلا من أن ننسخ نصف المصفوفة كلّ مرّة لإرسالها كمعامل للإجراء الارتداديّ، سنستخدم المتغيّرين St و En لنضع فيهما رقمي خانتي بداية ونهاية الجزء الذي نبحث فيه من المصفوفة.. وسنجعل هذين المتغيّرين ثابتي القيمة Static، حتّى نستطيع استخدامهما عبر الاستدعاءات المختلفة.. ولو حدث أن صار St = En فإنّ ذلك يعني أنّ المصفوفة الجزئيّة الحاليّة عبارة عن خانة واحدة وأنّ البحث قد انتهى.

فإذا وجدنا عنصر البحث في أيّ خطوة لا نتابع العمليّة ونعيد رقم الخانة.. وإذا لم نجد، فيمكن أن نعيد -1، أو سالب رقم آخر خانة وصلنا إليها، فهي الخانة التي تحتوى على النصّ التالي أبجديّا لعنصر البحث المفقود.

تعال نرَ مثالا أولا قبل أن نكتب الكود.. وسنجعل هذا المثال رقميّا للتسهيل:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| 1 | 3 | 4 | 5 | 7 | 8 | 10 | 11 | 12 | 16 | 18 | 20 | 21 | 22 |

نريد الآن أن نبحث عن الرقم 18 في المصفوفة السابقة:

- سنبدأ بالمتغيّر St = 0، والمتغيّر En = 13 (رقم الخانة الأخيرة في المصفوفة).

- ومن الأفضل أن نتأكّد أن عنصر البحث ليس في خانة البداية أو النهاية، قبل الاستمرار.

- خانة منتصف المصفوفة هي Mdl = (0 + 13) ÷ 2 = 6، حيث سنأخذ العدد الصحيح فقط، وسنهمل باقي القسمة.

- في الخانة رقم 6 سنجد الرقم 10.. وبما أنّ 18 أكبر من 10، إذن فسنوجّه بحثنا للنصف الأخير من المصفوفة.

- الآن سنجعل خانة البداية St = 7، أي أكبر من خانة المنتصف بواحد، وسنجعل خانة النهاية En = 12 أي أصغر من خانة النهاية السابقة بواحد.

ثمّ سنعيد نفس العمليّة في هذا الجزء الجديد:

- عنصر البحث ليس في خانة البداية أو النهاية.

- خانة منتصف الجزء الجديد تحمل رقم (7 + 12) ÷ 2 = 9، ويوجد بها الرقم 16.. ما زال عنصر البحث أكبر.

- سنجعل خانة البداية St = 10، أي أكبر من خانة المنتصف بواحد، وسنجعل خانة النهاية En = 11 أي أصغر من خانة النهاية السابقة بواحد، ونعيد الكرّة.

- سنجد أن الرقم 18 موجود في خانة البداية.. إذن فلا داعي للاستمرار.. لقد عثرنا على ضالتنا في الخانة رقم 10.

ويمكنك أن تجرّب البحث عن أرقام أخرى ـ حتّى لو كانت غير موجودة ـ بنفس الطريقة السابقة.

تعال الآن نرى كود هذا الإجراء:

**Function BinarySearch(ByRef Arr() As String, ByVal Target As String\_**

**Optional ByVal Reset As Boolean = False) As Integer**

**Static St As Integer = 0, En As Integer = UBound(Arr)**

**If Reset Then**

**St = 0**

**En = UBound(Arr)**

**End If**

**' شرط التوقف: العثور على عنصر البحث في خانة البداية أو النهاية**

**If Arr(St) = Target Then Return St**

**If Arr(En) = Target Then Return En**

**Dim Mdl As Integer = (St + En) \ 2**

**Select Case Arr(Mdl)**

**Case Target 'شرط التوقّف: العثور على نصّ البحث في خانة المنتصف**

**Return Mdl**

**Case Is < Target ' الهدف أكبر من قيمة خانة المنتصف**

**' شرط التوقّف: خانة المنتصف هي خانة النهاية**

**' يحدث هذا عندما يساوي رقم خانة البداية رقم خانة النهاية**

**If Mdl = En Then Return -1**

**' البحث في الجزء الأخير من المصفوفة الحاليّة**

**St = Mdl + 1**

**En -= 1**

**' نتيجة البحث هي الناتجة عن البحث في الجزء الأخير من المصفوفة**

**Return BinarySearch(Arr, Target)**

**Case Else ' الهدف أصغر من قيمة خانة المنتصف**

**' شرط التوقّف: خانة المنتصف هي خانة البداية**

**' يحدث هذا عندما يساوي رقم خانة البداية رقم خانة النهاية**

**If Mdl = St Then Return -1**

**' البحث في الجزء الأوّل من المصفوفة الحاليّة**

**St += 1**

**En = Mdl – 1**

**' نتيجة البحث هي تلك الناتجة عن البحث في النصف الأوّل من المصفوفة**

**Return BinarySearch(Arr, Target)**

**End Select**

**End Function**

لاحظ استخدامنا للمتغيّر الاختياري Reset.. لماذا يا ترى؟

لو أجريت أوّل بحث فستحصل على نتيجة، ولكن بعد ذلك لن تضمن النتيجة، لأنّ المتغيّرين St و En ثابتي القيمة Static وسيظلان محتفظين بآخر قيمتين أخذاهما من البحث السابق.. لهذا عند إجراء أيّ بحث جديد، لا بدّ أن نضع قيمة المتغيّر Reset بـ True، وذلك حتّى نعيد هذين المتغيّرين إلى قيمتيهما الابتدائيّتين، في جملة If الموجودة في أوّل الإجراء.

وكان بإمكانك استخدام المتغيّرين St و En كمعاملين اختيارين للدالة ـ كما فعلنا في إجراء البحث السريع ـ وبذلك تتخلّص من مشكلة إعادة المتغيّرات ثابتة القيمة Static Variables إلى قيمها الابتدائيّة، كما ستمكّن المستخدم من تحديد المنطقة التي يريد البحث فيها من المصفوفة.. طبعا الحلّ الأخير أفضل، ولكنّي أردت أن أريك طرقا مختلفة للتفكير وحلّ المشكلات.

ولاستدعاء هذا الإجراء، استخدم الكود التالي:

**Dim S As String = InputBox("اكتب نصّ البحث")**

**If S = "" Then Exit Sub**

**QuickSort(TextBox1.Lines) ' يجب ترتيب المصفوفة أولا**

**Dim R As Integer = BinarySearch(TextBox1.Lines, S, True)**

**If R = -1 Then**

**MsgBox("لم يتمّ العثور على نصّ البحث")**

**Else**

**MsgBox("نصّ البحث موجود في السطر رقم " + Str(R + 1))**

**End If**

وستجد الكود الكامل للبحث الثنائيّ في مشروع Arrays في مجلّد برامج الفصل السابق.

**القوالب Modules:**

بقي شيءٌ صغير ولكنّه هامّ: ماذا تفعل لترتيب إجراءاتك؟

إن الإجراءات يمكن إعادة استخدامها أكثر من مرّة في عديد من المشاريع، ولن يكون من السهل عليك أن تتذكّر أين يوجد إجراء معيّن، إذا كانت كل إجراءاتك متناثرة داخل النماذج.

إذن فلا بد من وجود "قالب" معيّن، تُنظّم فيه الإجراءات، بحيث يمكن الوصول إليها بسهولة.. هذه هي وظيفة القالب Module، فهو لا يمتلك واجهة مستخدم كالنماذج، ولا يمكن تعريف نسخ منه كالكائنات المعرَّفة من الخلايا Classes.. إنّه فحسب مجرّد قالب يحتوي على مجموعة من المتغيّرات والإجراءات الفرعيّة والدوال، بحيث يمكن وضع الإجراءات التي تؤدّي وظائف متشابهة أو مرتبطة، في قالب له اسم واضح، وبهذا يمكن إضافته للمشروع بسهولة عند الاحتياج إليه.

ولإضافة قالب جديد لمشروعك، اضغط القائمة الرئيسيّة "مشروع" / "إضافة قالب" Project.Add Module، ليظهر لك مربع حوار "إضافة عنصر"، حيث سيكون العنصر المحدّد هو القالب Module، والاسم الافتراضي للقالب الجديد هو Module1.vb.. غيّر الاسم إلى ما يناسبك، واضغط موافق.

وهذه الطريقة تضيف ملفا جديدا لمشروعك، سيتم عرضه لك، وسترى به تعريف النموذج كالتالي:

**Module Module1**

**End Module**

هذا هو مقطع القالب، حيث من الممكن أن يكون Module1 أي اسم تريده.. في هذا المقطع يمكنك أن تضيف المتغيرات والإجراءات التي تريد.

ويمكن إضافة قالب جديد بطريقة أخرى، وذلك بإضافة مقطع القالب في أي ملف من ملفات المشروع.. مثلا في ملف النموذج، بعد جملة End Class.. إنّ هذه الإمكانيّة متاحة، لأنّ كل ملفات مكونات البرنامج لها الامتداد ".vb" أيّ أنّها لا ترمز لشيء في حدّ ذاتها، وإنّما يمكن أن تحتوي على العديد من المكوّنات المختلفة، بمجرد تعريف مقطع كل منها في نفس الملف.

ولإضافة قالب موجود مسبّقا للمشروع، اضغط "مشروع" / "إضافة عنصر موجود" Project.Add Existing Item ، حيث يمكنك إضافة الملفّ الذي يحتوي على القالب.

والآن تعال نكتب قالبا يحتوي على الدوال الخاصّة بالطلاب، التي تعاملنا معها من قبل.. كل ما علينا فعله هو نقل هذه الدوال إلى مقطع القالب:

**Module ModStudent**

**Structure Student**

**Dim No As Integer**

**Dim ClassNo As Integer**

**Dim Grade As Integer**

**Function StudentName() As String**

**Dim Name As String**

**' الكود المناسب للبحث عن الطالب في قاعدة بيانات الطلاب بمعلوميّة رقمه**

**Return Name**

**End Function**

**Sub Print()**

**Console.WriteLine(StudentName)**

**Console.WriteLine (No)**

**Console.WriteLine(ClassNo)**

**Console.WriteLine(Grage)**

**End Sub**

**End Structure**

**Function StudentGrade(ByVal GreaterThan As Integer) As Student()**

**Dim S() As Student**

**' الكود المناسب للبحث عن الطلاب**

**Return S**

**End Function**

**Sub PrintStudentData( ThisStudent As Student)**

**Console.WriteLine (ThisStudent.No)**

**Console.WriteLine(ThisStudent.ClassNo)**

**Console.WriteLine(ThisStudent.Grage)**

**End Sub**

**Sub PrintStudentsData(Students() As Student)**

**Dim I As Integer**

**For I = 0 To Students.GerUpperBound(0)**

**PrintStudentData(Students(I))**

**Next I**

**End Sub**

**Function FindStudent(Name As String) As Student**

**Dim S As Student**

**' الكود المناسب للبحث عن اسم الطالب في قاعدة البيانات**

**Return S**

**End Function**

**Function FindStudent(No As Integer) As Student**

**Dim S As Student**

**' الكود المناسب للبحث عن رقم الطالب في قاعدة البيانات**

**Return S**

**End Function**

**Function FindStudent(Grade As Integer, ClassNo As Integer) As Student()**

**Dim S() As Student**

**' الكود المناسب للبحث عن كل طلاب هذا الفصل**

**' الحاصلين على هذه الدرجة في قاعدة البيانات**

**Return S**

**End Function**

**Private Function FindStudent(Grade As Long)As Student()**

**Dim S( ) As Student**

**' الكود المناسب للبحث عن كل الطلاب**

**' الحاصلين على هذه الدرجة في قاعدة البيانات**

**Return S**

**End Function**

**End Module**

الآن صار لديك قالب منظّم يحتوي على تعريف سجل الطالب والإجراءات التي تتعامل مع بيانات الطلاب، حيث يمكنك استدعاء هذه الإجراءات من أي موضع في أي إجراء، في هذا القالب أو في قالب آخر أو في نموذج أو خليّة، وذلك لأنّها عامّة Public.. اكتب هذه الجملة في أيّ إجراء في أيّ موضع في البرنامج:

**Dim X As Student = FindStudent("عادل حنفي")**

أمّا لو أحببت أن تكتب متغيرات وإجراءات تُستخدم فقط في القالب الذي يحتويها ولا يمكن استدعاؤها من أي مكان غيره، فضع كلمة Private قبل تعريف الإجراءات أو المتغيرات (مع ملاحظة أن استخدام كلمة Dim في تعريف المتغيّر تعني أنّه Private).

ولهذا السبب لن يمكن استخدام الصيغة الأخيرة من دالة البحث عن الطالب (البحث بالدرجة) من خارج هذا القالب، فلن تكون مرئيّة إلا في داخله.

وهناك طريقة شيّقة لعرض أسماء القوالب والنماذج وما تحويه من إجراءات وسجلات، عن طريق استخدام نافذة "عارض الخلايا" Class View، والتي يمكن إظهارها بضغط "عرض" / "عارض الخلايا" View. Class View.

بقيت ملاحظة أخيرة:

افترض أنّ لديك أكثر من قالب، وحدث أنّ متغيرا أو إجراءً موجود في أكثر من قالب بنفس الاسم.. فكيف نتعامل مع هذه الحالة؟

الأمر بسيط، اكتب اسم القالب كمُعرِّف، واستخدم المتغير أو الإجراء كعضو من أعضائه:

**Dim X As ModStudent.Student = ModStudent.FindStudent("عادل حنفي")**