**الفصل الرابع عشر**

**كتابة تطبيقات متعدّدة المهامّ**

**Writing Multithreaded Applications**

**·** [**ما هي العمليّات الفرعيّةThreads ؟:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\01-%20ما%20هي%20العمليّات%20المستقلّة.htm)

**·** [**إنشاء عمليّة فرعيّة Thread:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\02-%20إنشاء%20عمليّة%20مستقلّة.htm)

**·** [**نافذة العمليّات الفرعيّة Threads window:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\03-%20نافذة%20العمليّات%20المستقلّة.htm)

**·** [**العمليّات الخلفيّة Background، والرئيسيّة Foreground:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\04-%20العمليّات%20الخلفيّة.htm)

**·** [**وسائل العمليّة الفرعيّة:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\05-%20وسائل%20العمليّة%20المستقلّة.htm)

**·** [**حول استخدام العمليّات الفرعيّة:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\06-%20حول%20استخدام%20العمليّات%20المتعدّدة.htm)

**·** [**النموذج سريع التحميل:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\07-%20النموذج%20سريع%20التحميل.htm)

**·** [**التحكّم في العمليّات:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\08-%20التحكّم%20في%20العمليّات.htm)

**·** [**منبّه العمليّات Thread.Timer:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\09-%20منبّه%20العمليّات.htm)

**·       [العمليّات الآمنة:](file:///C:\\Users\\Administrator\\Desktop\\vb.net\\d-%20%20مواضيع%20متقدّمة\\14-%20المهامّ%20المتعدّدة\\14-%20المهامّ%20المتعدّدة\\10-%20العمليّات%20الآمنة.htm)**

**·** [**التعامل مع الأدوات من العمليّات الفرعيّة:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\11-%20التعامل%20مع%20الأدوات%20من%20العمليّات%20المستقلّة.htm)

**·** [**الاتصال بالعمليّة الفرعيّة:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\12-%20الاتصال%20بالعمليّة%20المستقلّة.htm)

**·** [**إيقاف العمليّة انتظارا لحدث ما:**](file:///C:\Users\Administrator\Desktop\vb.net\d-%20%20مواضيع%20متقدّمة\14-%20المهامّ%20المتعدّدة\14-%20المهامّ%20المتعدّدة\13-%20إيقاف%20العمليّة%20انتظارا%20لحدث%20ما.htm)

**ما هي العمليّات الفرعيّة؟:**

تعتمد فكرة نظام تشغيل الويندوز على منح المستخدم القدرة على التعامل مع عدد كبير من البرامج في نفس الوقت.. ولكن كيف يمكن أن يحدث ذلك رغم أنّ للكمبيوتر مشغّلا دقيقا Micro-processor واحدا، لا يستطيع أن يقوم بأكثر من عمليّة واحدة في نفس الوقت؟

الفكرة كالتالي: إنّ الويندوز يقسّم البرامج إلى مجموعات من العمليّات Processes، كلّ عمليّة منها لها رقمها وموقعها في الذاكرة وقيم متغيّراتها الخاصّة بها، ولها أولويّة في التنفيذ Priority.. وكلّ ما يحدث، هو أن يتمّ تقسيم وقت التنفيذ بين هذه العمليّات بما يتناسب مع أولويّة كلّ عمليّة.. فالعمليّة الأكثر أولويّة تأخذ جزءا أكبر من الوقت لتنفيذها.. فإذا انتهى جزء الوقت الخاصّ بها، انتقل التنفيذ لجزء آخر من عمليّة أخرى، ثمّ أخرى، حتّى آخر عمليّة.. بعد هذا يعود الكمبيوتر ليواصل تنفيذ جزء آخر من أوّل عمليّة، فالثانية فالثالثة وهكذا، حتّى يتمّ الانتهاء من هذه العمليّات جميعا.

وطبعا لا تشعر أنت بأيّ من هذا، فالوقت المخصّص لتجزئة العمليّات هو أجزاء ضئيلة من الثانية.

ليس هذا فحسب.. بل يمكن تقسيم كلّ عمليّة Process إلى عمليّات فرعيّة Threads، بحيث يمكن أن ينفّذ البرنامج أكثر من مهمّة في نفس الوقت.

وسيتساءل مبرمج VB: جميل.. ولكن ما دخلي أنا بكلّ هذا؟.. كلّ ما عليّ فعله هو كتابة برنامجي، وعلى نظام التشغيل أن يفعل عند تنفيذه ما حلا له.

معك حقّ، وهذا هو الوضع في معظم البرامج التي ستكتبها.. ولكنّ الجديد هنا، هو أنّ VB.Net يمنحك القدرة على تقسيم برنامجك نفسه إلى عمليّات فرعيّة Threads.. إنّ هذا معناه أن مستخدم برنامجك يمكنك أن ينفّذ أكثر من عمليّة معا في نفس الوقت.

ما زلت تشعر بعدم أهمّيّة ما أقول؟

حسنا.. افترض أنّك كتبت برنامجا ينفّذ عمليّة طويلة، مثل ضغط ملفّ كبير الحجم مثلا.. في هذه الحالة سيكون على مستخدم برنامجك أن يضع يده على خدّه، وينتظر حتّى تنتهي هذه العمليّة، دونَ أن يستطيع أن يفعل أيّ شيءٍ آخر ببرنامجك.. هنا تأتي فائدة تعدّد العمليّات Multithreading، فلو جعلت ضغط الملفّ عملية فرعيّة (بالأولويّة التي تريدها)، فسيتمّ إجراؤها في خلفيّة البرنامج، بينما سيكون بإمكان المستخدم في هذه الحالة أن يواصل استخدام برنامجك، فيقوم بعرض ملفّات أخرى وقراءتها أو تحريرها.. إلخ.. وعندما تنتهي عمليّة ضغط الملفّ، سيتمّ إخبار برنامجك بذلك، لتعرض للمستخدم أيّ رسالة تفيد بانتهاء العمليّة.

أعتقد أنّك قد بدأت تهتمّ.. جميل.. تعال معي نتعرّف في هذا الفصل على كيفيّة كتابة المهامّ المتعدّدة، فلربّما تحتاج لهذه الإمكانيّة الرائعة في تطبيقاتك.

ولكن عليك أن تلاحظ أولا، أنّ تقسيم البرنامج إلى عمليّات فرعيّة ليس جائزا في كلّ الأحوال، فمثلا، لو كنت تقوم بإجراء عمليّة على صورة معيّنة ـ كعكس ألوانها مثلا ـ فليس من المنطقيّ أن تسمح للمستخدم بأداء عمليّة أخرى على نفس الصورة في نفس اللحظة، وإلا فسيحصل على نتائج عبارة عن "لخبطة"!!

دُعَّ عنّا الكلام، وتعالَ نرى مثالا بسيطا.

**إنشاء عمليّة فرعيّة Thread:**

عندما تعرض للمستخدم رسالة ـ باستخدام الدالة MsgBox ـ فإنّ البرنامج يتوقّف عن الاستجابة لك إلى أن تغلق هذه الرسالة.. ما رأيك أن نجرّب عرض الرسالة كعمليّة فرعيّة، لنرى ماذا سيحدث؟

أوّل خطوة هي أن تبدأ بتعريف الإجراء الذي ستنفّذه العمليّة.. هذا الإجراء يجب ألا تكون له أيّ معاملات:

**Private Sub BackgroundProcess()**

**MsgBox("هذه عمليّة فرعيّة ولا تؤثّر على البرنامج الأصليّ")**

**End Sub**

والآن نريد جعل هذا الإجراء مهمّة فرعيّة.. ابدأ بكتابة جملة الاستيراد التالية في بداية الملفّ:

**Imports System.Threading**

والآن ها هو ذا الكود الذي يستدعى الإجراء BackgroundProcess كعمليّة فرعيّة:

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

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

**' عرّف متغيّر للعمليّة**

**Dim t As Thread**

**' أرسل لهذه العمليّة عنوان الإجراء الذي ستنفّذه**

**t = New Thread(AddressOf BackgroundProcess)**

**' ابدأ تنفيذ العمليّة**

**t.Start()**

**End Sub**

الآن جرّب تشغيل هذا الكود في مشروع MsgThread.. ستكتشف أنّك تستطيع الانتقال بينَ النموذج والرسالة بلا عوائق.. ولو ضغطت الزرّ مرّة أخرى، فسيتمّ عرض رسالة ثانية!



ليس هذا فحسب، بل إنّك تستطيع إغلاق النموذج دونَ أن يتم إغلاق الرسالة!.. شيء طبيعيّ، فكلاهما عمليّة مستقلّة تماما عن الأخرى.. وطبعا لن ينتهي تنفيذ البرنامج ما دامت إحدى عمليّاته ما زالت تعمل.. لهذا لو أردت إغلاق البرنامج نهائيّا، فعليك أن تغلق كلّ الرسائل المعروضة على الشاشة.

أضف زرّا جديدا للنموذج، واستخدمه لعرض رسالة عاديّة:

**Private Sub Button2\_Click(ByVal sender As System.Object, \_**

**ByVal e As System.EventArgs) Handles Button2.Click**

**MsgBox("أغلق هذه الرسالة أولا")**

**End Sub**

ستجد أنّ عرض هذه الرسالة التقليديّة سيمنعك من التعامل مع النموذج، ولكنّه لن يؤثّر على تعاملك مع الرسالة المعروضة من العمليّة الفرعيّة.

**نافذة العمليّات الفرعيّة Threads window:**

إذا أردت متابعة العمليّات الفرعيّة وهي تنفّذ، فافتح نافذة العمليّات الفرعيّة من قائمة Debug بضغط الأمر Threads.. بهذا يمكنك في أثناء وقت التوقّف Break Time أن تعرّف العمليّة التي يتمّ تنفيذها حاليّا.

**العمليّات الخلفيّة Background Threads، والعمليّات الرئيسيّة Foreground Threads:**

هذان النوعان من العمليّات متماثلان، إلا في شيءٍ واحد: العمليّة التي يتمّ تنفيذها في خلفيّة البرنامج لا يمكن أن تستمرّ بمفردها، لهذا إذا تمّ إنهاء كلّ العمليّات الرئيسيّة، فإنّ العمليّات الخلفيّة سيتمّ إنهاؤها في الحال وإغلاق البرنامج.

ولكي تحدّد نوع العمليّة، يمكنك استخدام الخاصيّة IsBackground، فلو جعلتها True فسيتمّ تنفيذ العمليّة في الخلفيّة، ولو جعلتها False فستصبح العمليّة عمليّة رئيسيّة، وهو الوضع الافتراضيّ للعمليّات التي تبدأها.

الآن لو عدّلت كود البرنامج السابق لتظهر الرسالة كعمليّة خلفيّة، فسيتمّ إغلاقها عند إغلاق البرنامج:

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

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

**Dim t As New Thread(AddressOf BackgroundProcess)**

**' اجعل هذه العمليّة تتمّ في خلفيّة البرنامج**

**t.IsBackground = True**

**t.Start()**

**End Sub**

**وسائل العمليّة الفرعيّة:**

**ابدأ Start:**

تبدأ تنفيذ العمليّة.

**إجهاض Abort:**

لإنهاء تنفيذ العمليّة.. وبعد أن تفعل ذلك، لا تحاول استكمال تنفيذ العمليّة أو بدئها مرّة أخرى، فإنّها ستكون قد انتهت تماما ولن تحصل إلا على خطإ!.. في هذه الحالة عليك أن تعيد تعريف العمليّة مرّة أخرى وبدئها من جديد.. ولا يعني ذلك أنّك ستضطرّ لأن تعرّف متغيّرا جديدا.. يكفي أن تضع في نفس المتغيّر نسخة جديدة من العمليّة كالتالي:

**متغيّر العمليّة = New Thread(AdressOf إجراء العمليّة)**

**تأجيل Suspend:**

لإيقاف العمليّة مؤقّتا.

**إكمال Resume:**

لإكمال العمليّة التي تمّ إيقافها.

**تعطيل Sleep:**

لإيقاف العمليّة للعدد الذي تريده أنت من أجزاء الثانية Milliseconds.. لاحظ أنّك لا تستطيع إنهاء العمليّة بينما هي متوقّفة، لهذا يجب أن تعيدها للعمل أولا، قبل أن تقوم بإنهائها.

**متواصل Join:**

لتنفيذ العمليّة الفرعيّة الحاليّة باستمرار لوقت تحدّده أنت أو لحين انتهائها، على حساب إيقاف العمليّة الأصليّة أثناء ذلك.. ويمكن الاستفادة من هذه الوسيلة في شيء هامّ: فمجرّد استدعاء الدالة Abort قد لا ينهي العمليّة في الحال وقد يستهلك بعض الوقت.. فلو أردت أن تتأكّد من أنّ العمليّة قد انتهت نهائيّا قبل مواصلة الكود، فاستخدم الوسيلة Join في السطر التالي مباشرة للسطر الذي يستدعي الوسيلة Abort.

**Th.Abort()**

**Th.Join()**

**حول استخدام العمليّات المتعدّدة:**

لا تحاول أن توقع نفسك في العنت، بمحاولة استخدام العمليّات الفرعيّة في كلّ تطبيق تكتبه، لمجرّد أنّ هذه الإمكانيّة متاحة، فوجود أكثر من عمليّة في برنامجك سيعقّد عمليّة تصحيح الأخطاء Debugging، هذا بخلاف البطء الذي سيلمّ بالبرنامج نتيجةً لتنفيذ أكثر من عمليّة في نفس الوقت، فانتقال نظام التشغيل بين عمليّة وأخرى Context Switching عمليّة معقّدة، وتستهلك بعض الوقت.. فلا تحاول التمادي في تشغيل عدد كبير من العمليّات في برنامجك.

ولكنّ هناك مواقف ستحتاج فيها لاستخدام عمليّات فرعيّة، مثل تحريك مجموعة من الأشكال على نموذج، بحيث يكون كل منها مستقلا عن الآخر في حركته.. في هذه الحالة يمكنك تنفيذ الكود الذي يحرّك كلّ شكل في عمليّة فرعيّة.. لاحظ أن مبرمجي VB6 كانوا يحلّون هذه المشكلة باستخدام المنبّه Timer، فكانوا يجعلون لكلّ واحد من الأشكال منبها مستقلا مسئولا عن تحريكه.

موضع آخر لاستخدام العمليّات الفرعيّة، هو عندما تريد أن تسمح للمستخدم بإنهاء الجمل التكراريّة الطويلة.. فإمّا أن تكتب داخل الجملة التكراريّة تعبير Application.DoEvents للسماح للتطبيق بالاستجابة لأحداث الفأرة ولوحة المفاتيح.. في هذه الحالة لو ضغط المستخدم زرّ إلغاء العمليّة، فسيتمّ تغيير قيمة أحد المتغيّرات ليشير إلى هذا.. مثلا:

**Abort = True**

في هذه الحالة يجب أن يكون السطر التالي لتعبير DoEvents هو السطر التالي:

**If Abort Then Exit For**

هذه طريقة.. الطريقة الأخرى هي تنفيذ الجملة التكراريّة كعمليّة فرعيّة، وعند ضغط زرّ الإلغاء يتمّ إنهاء العمليّة باستدعاء الوسيلة Abort.. واضح أنّ هذه الطريقة أبسط من الطريقة الأولى، هذا بخلاف أنّ DoEvents تمثّل عبئا على البرنامج وقد تتسبّب في بطئه.

**النموذج سريع التحميل:**

لو كان على برنامجك أن يقوم بتحميل كمّيّة كبيرة من البيانات قبل أن يعرض النموذج الرئيسيّ، ففي هذه الحالة سينتظر المستخدم لفترةٍ طويلة قبل أن يستطيع استخدام البرنامج.. أحد الحلول الطريفة لهذه المشكلة هو تحميل البيانات في عمليّة فرعيّة، بحيث تظهر النافذة الرئيسيّة للبرنامج أولا، بينما ما زال تحميل البيانات يتمّ في الخلفيّة.. لاحظ أنّ استخدام المنبّه Timer في هذه الحالة لن يجدي، لأنّ المنبّه يتوقّف تماما عن العمل إذا كان البرنامج مشغولا في تنفيذ إجراء ما، ولا يعود للعمل إلا بعد انتهاء الإجراء.

وفي المثال التالي سنرى كيف نحمّل نموذجا عليه قائمة بها مئة ألف عنصر.. طبعا هذا رقم ضخم، ولو جرّبت تحميله في حدث تحميل النموذج، فسيستغرق النموذج وقتا طويلا قبل أن يظهر.. لهذا فسنحمّل هذه العناصر في عمليّة فرعيّة:

**Private Sub FillList()**

**Dim i As Integer = 1**

**For i = 1 To 100000**

**ListBox1.Items.Add("Item: " + i.ToString)**

**Next**

**MsgBox("done!")**

**End Sub**

الآن يمكننا أن نطلق عمليّة ملء القائمة من حدث تحميل النموذج:

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

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

**Dim bgThread As New Thread(AddressOf FillList)**

**bgThread.Start()**

**End Sub**

وعند ضغط أيّ عنصر، ستظهر رسالة تمثّل رقم العنصر في القائمة:

**Private Sub ListBox1\_SelectedIndexChanged(ByVal sender As Object, \_**

**ByVal e As System.EventArgs) \_**

**Handles ListBox1.SelectedIndexChanged**

**MsgBox(ListBox1.SelectedIndex.ToString)**

**End Sub**

الآن جرّب تشغيل تطبيق FastLoad Form في مجلد برامج هذا الفصل.. سرعان ما سيظهر لك النموذج بينما القائمة ما زالت لم تمتلئ بكلّ العناصر.. ويمكنك أن تضغط أيّ عنصر لتظهر لك رسالة تحمل رقمه.. كلّ ذلك بينما ملء القائمة ما زال يترى.

**التحكّم في العمليّات:**

في تطبيق MovingImages يمكنك أن ترى كيفية استخدام العمليّات الفرعيّة Threads لتحريك ثلاثة من مربّعات الصور ImageBoxes رأسيّا على النموذج.. في هذا التطبيق ستتعلّم استخدام الخصائص التالية للعمليّات:

**حالة العمليّة ThreadState:**

تعيد لك هذه الخاصيّة الحالة التي عليها العمليّة الحاليّة.. وقيمها كالتالي:

|  |  |
| --- | --- |
| تمّ استدعاء العمليّة، ولكنّها لم تبدأ بعد. | Unstarted |
| بدأ تنفيذ العمليّة. | Running |
| تمّ إيقاف العمليّة لبعض الوقت، أو تمّت مواصلة العمليّة باستمرار لبعض الوقت. | WaitSleepJoin |
| مطلوب إيقاف العمليّة، ولكنّها لم تتوقّف بعد. | SuspendRequested |
| تمّ إيقاف العمليّة. | Suspended |
| يتمّ تنفيذ العمليّة. | Running |
| مطلوب إنهاء العمليّة، ولكنّها لم تنته بعد. | AbortRequested |
| تمّ إنهاء العمليّة. | Aborted |

**اسم العمليّة Name:**

تسمح لك هذه الخاصيّة بوضع اسم للعمليّة، كما تسمح لك بقراءة هذا الاسم فيما بعد.

**الأولويّة Priority:**

تحدّد هذه الخاصيّة الوقت الذي سيخصّص لتنفيذ أجزاء كلّ عمليّة.. وطبعا العمليّة ذات الأولويّة الأعلى هي التي تأخذ حظّا أكبر من غيرها في التنفيذ، بحيث يتمّ تنفيذها أسرع من أيّ عمليّة أخرى لها نفس الحجم ولكنّ أولويّتها أقلّ.. وتأخذ هذه الخاصيّة القيم التالية:

|  |  |
| --- | --- |
| Highest | أعلى أولويّة ممكنة. |
| AboveNormal | أولويّة فوق الأولويّة العاديّة. |
| Normal | أولويّة عادية. |
| BelowNormal | أولويّة أقل من العاديّة. |
| Lowest | أقل أولويّة ممكنة. |

كما يوضّح لك تطبيق MovingImages كيفيّة استخدام الخاصيّة Thread.CurrentThead للتعامل مع العمليّة التي يقوم الكمبيوتر بتنفيذها حاليّا.. استمتع بدراسة هذا المشروع.

**منبّه العمليّات Thread.Timer:**

هذا المنبّه يسمح لك باستدعاء أيّ دالّة كلّ فترة من الوقت، بمجرّد تمرير اسمها إلى المنبّه عند تعريفه.. هذه الدالة سيتمّ تنفيذها في عمليّة فرعيّة بخلاف تلك التي يوجد بها المنبّه.

طبعا هذا المنبه مختلف عن ذلك الذي عرفناه من قبل System.Windows.Forms.Timer، كما أنّ هناك نوعا ثالثا من المنبّهات تحت فضاء الاسم System.Timers.Timer، فلا ترتبك بين هذه الأنواع!

وفي المثال التالي ترى كيف نستدعي الإجراء CheckStatus كلّ ثانيتين.. هذا الاستدعاء سيبدأ بعد ثلاث ثواني من تشغيل المنبّه:

**Dim timer As Timer**

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

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

**timer = New Timer(AddressOf CheckStatus, Nothing, 3000, 2000)**

**End Sub**

**Private Sub Form1\_KeyDown(ByVal sender As Object, ByVal e \_**

**As KeyEventArgs) Handles MyBase.KeyDown**

**If e.KeyCode = Keys.Escape Then**

**' إنهاء عمل المنبّه**

**timer.Dispose()**

**End If**

**End Sub**

**' لاحظ أنّ الإجراء الذي ستستدعيه لا بدّ أن يكون له معامل من النوع كائن**

**Shared Sub CheckStatus(ByVal state As Object)**

**MsgBox("Esc هذه الرسالة ستظهر كلَّ ثانيتين، إلى أن تضغط زر" + \_**

**vbCrLf + "ولكن اضغط النموذج أولا ليكون فعالا، حتّى يستقبل ضغطة الزرّ")**

**End Sub**

**العمليّات الآمنة:**

عند التعامل مع عمليّات فرعيّة مختلفة قد تحدثُ بعض المشاكل.. من هذه المشاكل التسابق على المعلومات Race Condition، فقد يحدث مثلا أنّ تحاول أكثر من عمليّة فرعيّة الكتابة في ملفّ واحد.. في هذه الحالة لن تكون النتائج مضمونة.

ولحلّ هذه المشكلة، يمكنك إجبار كلّ عمليّة على تنفيذ مقطع من الكود بالكامل، بحيثُ لا ينتقلُ التنفيذ إلى عمليّة أخرى قبل أن ينتهي تنفيذ هذا المقطع.. وبهذا يمكنك أن تضع جمل الكتابة في الملفّ في مقطع واحد، بحيث تنفّذ بالكامل ويتمّ إغلاق الملفّ أولا، قبل أن تسمح لأيّ عمليّة أخرى بالتعامل مع نفس الملفّ.

ولتحديد مقطع متزامن التنفيذ، ضعه بين جملتي SyncLock.. End SyncLock، مع وضع كائن قراءة الملفّ Stream بعد كلمة SyncLock، ليتمّ منع التعامل معه.

والمثال التالي يمنع أيّ عمليّة أخرى من التعامل مع النموذج، أثناء تنفيذ مقطع الكود:

**SyncLock Me**

**' كود التعامل مع النموذج**

**End SyncLock**

والمثال التالي يمنع أيّ عمليّة فرعيّة أخرى من التعامل مع الزرّ Button1، أثناء تغيير عنوانه:

**SyncLock (Button1)**

**button1.Text = "Something Else"**

**End SyncLock**

واضح طبعا أنّ الكائن الذي تحدّده في بداية مقطع جملة التنفيذ غير المتزامن، يتمّ قصر التعامل معه على العمليّة الفرعية الحاليّة، بينما يتم تعطيل باقي العمليّات التي تحاول التعامل معه، إلى أن يتمّ انتهاء مقطع التنفيذ المتزامن.

**نصائح هامّة:**

- لا تستخدم المقطع المتزامن إلا للضرورة القصوى، فزيادة عدد هذه المقاطع يؤثّر على كفاءة الأداء.

- لا تستدعِ أيّ عمليّات فرعيّة Threads أخرى من داخل مقطع متزامن التنفيذ، خاصّة لو كانت العمليّة الجديدة بدورها بها مقطع متزامن، فقد يوقعك هذا في دائرة مغلقة، ويجمّد تنفيذ البرنامج.

- إذا كان لديك أكثر من عمليّة فرعيّة تستخدم نفس الكائن (زرّ أو ملفّ أو ....)، فضع الكود الذي يستخدم هذا الكائن في مقطع متزامن، في كلّ عمليّة فرعيّة، بحيث تتجنّب أي نتائج غير متوقّعة.

إنّ تطبيق SyncLockMsgThread هو نفسه تطبيق MsgThread، ولكنّنا جعلنا عمليّة عرض الرسالة تحدث في مقطع متزامن التنفيذ.

جرّب هذا التطبيق.. اضغط زر "رسالة مستقلّة" ستظهر لك الرسالة.. لا تغلقها واضغط الزرّ مرّة أخرى.. ستكتشف أنّ الرسالة الثانية لن تظهر إلا إذا أغلقت الرسالة الأولى.. هذه هي الفكرة: أنّك تضمن عدم تنفيذ المقطع المتزامن أكثر من مرّة في نفس الوقت.

**التعامل مع الأدوات من العمليّات الفرعيّة:**

لا يمكن استخدام وسائل وخصائص أدوات الويندوز إلا من داخل العمليّة التي تنتمي إليها الأداة، وإلا فلن تضمن النتائج.. ولا يستثنى من هذا إلا الوسيلة CreateGraphics، فاستخدامها مباشرة عبر العمليّات المختلفة هو استخدام آمن.. ويمكنك التحقّق من قيمة الخاصيّة "وسيلة الاستدعاء مطلوبة" InvokeRequired، فلو كانت True، ففي هذه الحالة عليك أن تستخدم وسائل الاستدعاء غير المباشر التالية:

**بدء الاستدعاء BeginInvoke:**

تبدأ استدعاء إحدى وسائل الأداة، سواء من نفس عمليّتها أو من عمليّة أخرى.. وعند استدعاء هذه الوسيلة أرسل لها كمعامل مندوبا Delegate عن الوسيلة التي ستستدعيها.. كما يمكنك أن ترسل كمعامل ثانٍ مصفوفة من النوع Object تحتوي على قيم معاملات الوسيلة ـ إن وجدت.. اتبع معنا هذه الخطوات:

* ابدأ بتعريف Delegate يمثّل صيغة الوسيلة التي ستستدعيها، وكمثال:

**Public Delegate Sub XX (X As Integer, Y As Integer)**

* ثمّ عرّف متغيّرا من هذا المندوب:

**Dim CallXX As New XX(AddressOf MySub)**

حيث MySub هو اسم الإجراء الذي ستستدعيه.. ويشترط أن يكون متناسبا مع صيغة المندوب، بمعنى أن يكون له نفس عدد المعاملات وبنفس الأنواع.

* عرّف مصفوفة كائنات وضع فيها قيم المتغيّرات كالتالي:

**Dim args() As Object = {100, 200}**

* الآن يمكنك استخدام الوسيلة Invoke لاستدعاء الإجراء MySub كالتالي:

**Dim R As IAsyncResult = Me.BeginInvoke(CallXX, args)**

لاحظ أنّ استخدام كلمة Me يمكّنك من استدعاء وسائل وخصائص النموذج، سواء الأساسيّة منها أو التي عرّفتها أنت على أنّها عامّة Public.. ولو شئت استدعاء وسائل وخصائص أيّ أداة، فاستبدل اسم الأداة بكلمة Me، فكلّ الأدوات تمتلك وسائل الاستدعاء غير المباشر.

ما يجب أن تلاحظه، هو أنّه عند استدعاء هذه الوسيلة لن ينتظر البرنامج حتّى ينتهي تنفيذها، بل سيستمرّ تنفيذ ما يليها من كود.. أي أنّ هذه الوسيلة تنفّذ بطريقة غير متزامنة Asynchronous.. وتعيد هذه الوسيلة كائنا من النوع "واجهة النتيجة غير المتزامنة" IAsyncResult.

**إنهاء الاستدعاء EndInvoke:**

أرسل لهذه الدالة القيمة المعادة من الوسيلة BeginInvoke، لتعيد لك القيمة المعادة من الوسيلة التي تمّ استدعاؤها:

**Dim ReturnValue As Object = Me.EndInvoke( R )**

**استدعاء Invoke:**

هذه الوسيلة متزامنة Synchronous، بمعنى أنّها توقف البرنامج إلى أن يتمّ تنفيذ الوسيلة التي تستدعيها، بعد ذلك يتمّ مواصلة باقي الكود، وإذا كانت للوسيلة المنفّذة قيمة معادة، فإنّ بإمكانك استقبالها كقيمة معادة من الوسيلة Invoke.. ولا يمكنك استخدام Invoke إذا كانت الوسيلة المراد تنفيذها موجودة في نفس العمليّة.

ولاستخدام هذه الوسيلة اتبع نفس الخطوات السابق ذكرها مع الوسيلة BeginInvoke.. الفارق الوحيد أنّك لن تحتاج هنا لاستخدام الوسيلة EndInvoke، فالقيمة المعادة من الدالة ـ إن وجدت ـ ستعيدها لك الوسيلة Invoke مباشرة:

**Dim R As Object = Me.Invoke(CallXX, args)**

ويمكنك التدريب على استخدام الوسيلة Invoke في تطبيق Image Processing، في مجلّد برامج هذا الفصل، حيث جعلنا التأثيرات التي يتمّ إجراؤها على الصورة تنفّذ في عمليّة فرعيّة.. وأثناء تنفيذ العمليّة يجب استدعاء الإجراء ShowPixels لتحديث الصورة، والإجراء DoneProcessing للإعلان عن انتهاء التنفيذ.. ولاستدعاء هذين الإجراءين اتبعنا هذه الخطوات:

* تعريف Delegate لكلّ منهما:

**Public Delegate Sub ImageDisplayCallback(ByVal Y As Integer, \_**

**ByVal pixels() As Color, ByVal Refresh As Boolean)**

**Public Delegate Sub DoneProcessingCallback()**

* تعريف متغير من كلّ Delegate:

**Private CallShowPixels As New \_**

**ImageDisplayCallback(AddressOf ShowPixels)**

**Private CallDoneProcessing As New \_**

**DoneProcessingCallback(AddressOf DoneProcessing)**

* استخدام الوسيلة Invoke لاستدعاء الإجراء DoneProcessing عن طريق المتغيّر CallDoneProcessing:

**Me.Invoke(CallDoneProcessing)**

ولاستدعاء الإجراء ShowPixels يجب إعداد مصفوفة تحتوي على قيم معاملاته.. لهذا عرّفنا المصفوفة Args كالتالي:

**Dim args(2) As Object**

حيث وضعنا في الخانة الأولى قيمة المعامل الأوّل (وهو رقم الصفّ الذي سنحدّثه في الصورة):

**args(0) = i**

ووضعنا في الخانة الثانية مصفوفة نقاط هذا الصفّ:

**args(1) = pixels**

حيث pixels هي مصفوفة ألوان، تمّ تعريفها وملؤها في إجراء العمليّة (راجع التطبيق المشار إليه).

أمّا الخانة الثالثة فقيمتها True أو False على حسب رغبتنا في إنعاش الصورة المرسومة (كلّ عشرة صفوف).

وأخيرا استدعينا الإجراء ShowPixels كالتالي:

**Me.Invoke(CallShowPixels, args)**

**الاتصال بالعمليّة الفرعيّة:**

من الطبيعيّ أن تحتاج للاتصال بالعمليّة بعد تشغيلها، على الأقلّ حتّى تعرف متى ينتهي تنفيذها.. تعال نتعرّف على كيفيّة الاتصال بالعمليّة.

إن إجراء العمليّة لا يمتلك أيّ معاملات ولا يعيد أيّ قيمة.. ولحلّ هذه المشكلة، يمكنك أن تخصّص خليّة Class لكلّ عمليّة، وتجعل لها من الخصائص ما يمكّنك من قراءة البيانات من العمليّة وتمريرها إليها.. في هذه الخليّة ستعرّف أيضا حدثا يدلّ على انتهاء العمليّة.. هذه الخليّة هي التي ستحتوي على الإجراء الذي ينفّذ العمليّة، والذي من خلاله يجب أن تحدّث قيم خصائص الخليّة حتّى تستطيع قراءتها من برنامجك.. كما يجب أن يكون آخر سطر في هذا الإجراء هو حدث إطلاق حدث انتهاء العمليّة.. انظر للمثال التالي:

**Public Class ThreadedCalculator**

**Public Value1 As Double**

**Public Value2 As Double**

**Public Event ThreadComplete(ByVal result As Double)**

**Public Sub Calculate()**

**' كود العمليّة**

**If Value1 =قيمة معيّنة Then Value2 = القيمة الجديدة**

**RaiseEvent ThreadComplete(القيمة العائدة)**

**End Sub**

**End Class**

وقبل بدء العمليّة، يجب وضع قيمة الخاصيّة Value1 وبهذا فهي تعمل كأنّها معامل للعمليّة:

**Dim WithEvents TCalc As New ThreadedCalculator()**

**TCalc.Value1 = 99.09**

**Dim oThread As New Thread(AddressOf TCalc.Calculate)**

**OThread.Start()**

فإذا أردت أن تعرف قيمة الخاصيّة Value2 بعد انتهاء العمليّة، فعليك بقراءتها في حدث انتهاء العمليّة كما يلي:

**Sub ThreadHandler(ByVal result As Double) \_**

**Handles TCalc.ThreadComplete**

**Console.WriteLine("القيمة المعادة من العمليّة هي " & result.ToString)**

**End Sub**

ولمزيد من التدريب على استخدام العمليّات الخاصّة، افحص التطبيق Image Processing في مجلّد برامج هذا الفصل، وفيه ترى كيف طوّرنا نفس التطبيق الذي أنشأناه في الفصل الحادي عشر، ليقوم بأداء العمليّات على الصورة في عمليّة فرعيّة، ممّا يمنحنا القدرة على التعامل مع النموذج أثناء تنفيذ العمليّة، كما يمنحك القدرة على إلغاء العمليّة بضغط زر الإلغاء Esc من لوحة المفاتيح.

**إيقاف العمليّة انتظارا لحدث ما:**

رغم أنّ الطريقة السابقة سهلة، إلا إنّها قد ترهقك في كتابة كود طويل، خاصّة عندما يتعامل برنامجك مع عدد كبير من العمليّات.. لهذا فإنّ لديك طريقة أخرى قد تكون أصعب قليلا، ولكنّها تريحك من كتابة كود طويل.. حيث يمكنك أن تجبر العمليّة على انتظار انطلاق حدث ما، مثل انتهاء عمليّة أخرى، باستخدام الخليّة Thread.WaitHandle، التي تمتلك الوسائل التالية:

|  |  |
| --- | --- |
| WaitAll | انتظر حدوث جميع العناصر، الموجودة في المصفوفة التي ترسلها كمعامل. |
| WaitAny | انتظر حدوث أيّ عنصر من العناصر، الموجودة في المصفوفة التي ترسلها كمعامل. |
| WaitOne | انتظر حدوث حدث معيّن، من العناصر الموجودة في المصفوفة التي ترسلها كمعامل. |

ولكن ما هي تلك المصفوفة التي نرسلها كمعامل؟

هذه المصفوفة ستحتوي على عناصر من نوع خلايا خاصّة، مشتقّة من الخليّة WaitHandle، مثل AutoResetEvent و ManualResetEvent و Mutex.

وكمثال، لو أردت جعل عمليّة رئيسيّة تنتظر انتهاء كلّ العمليّات التي انطلقت من خلالها، اتبع الخطوات التالية:

1. عرّف خليّة للعمليّة، بحيث يقبل حدث إنشائها New معاملا من نوع الخليّة AutoResetEvent.

**Public Sub New(done As AutoResetEvent)**

1. اكتب الأمر التالي بحيث يكون آخر سطر في إجراء العمليّة:

done.Set()

إنّ استخدام الوسيلة Set يدلّ على أنّ العمليّة قد انتهت.. ويمكن استخدام الوسيلة Reset للإشارة إلى أنّ العمليّة قد عادت للعمل.

ها هو ذا كود الخليّة كاملا:

**Imports System.Threading**

**Class TClass**

**Private done As AutoResetEvent**

**Public Sub New(done As AutoResetEvent)**

**Me.done = done**

**End Sub**

**Public Sub TMethod()**

**' كود العمليّة ................................................**

**done.Set() ' العمليّة الحاليّة قد انتهت**

**End Sub**

**End Class**

3. عرّف مصفوفة من الخليّة AutoResetEvent عدد خاناتها مساوٍ لعدد العمليّات التي ستتعامل معها.

**Dim waitEvents(NorOfThreads - 1) As AutoResetEvent**

4. يجب أن تكون حالة العمليّة False عند بدايتها، للدلالة على أنّها لم تنتهِ بعد:

**waitEvents(i) = New AutoResetEvent(False)**

5. عند تعريف كل نسخة من خليّة العمليّة، أرسل لها كمعامل أحد عناصر المصفوفة waitEvents، حتّى يتمّ تغيير حالة العمليّة من خلال كود إجرائها:

**TClass = New TClass(waitEvents(i))**

6. الآن يمكنك إرسال المصفوفة waitEvents لأيّ وسيلة من وسائل الخليّة WaitHandle لجعل العمليّة الحاليّة تنتظر انتهاء بعض أو كلّ العمليّات الجديدة:

**WaitHandle.WaitAll(waitEvents)**

أعرف أنّ هذه الخطوات تبدو لك صعبة، ولكنّك سرعان ما ستألفها.

والآن تعال نكتب الإجراء الذي يطبّق الخطوات السابقة.. لاحظ أنّه مسبوق بالسمة Attribute المسمّاة MTAThread، حيث إنّ الحروف MTA هي اختصار لتعبير Multithreaded Apartment، ولن نخوض الآن في هذا الأمر، ولكن يكفيك أن تعلم أنّ كتابة هذه السمة أمر واجب!

**<MTAThread()> Shared Sub Main()**

**Dim NorOfThreads As Integer = 3**

**Dim waitEvents(NorOfThreads - 1) As AutoResetEvent**

**Dim ts As Thread**

**Dim TClass As TClass**

**Dim i As Integer**

**For i = 0 To NorOfThreads – 1**

**waitEvents(i) = New AutoResetEvent(False)**

**TClass = New TClass(waitEvents(i))**

**ts = New Thread(AddressOf TClass.TMethod)**

**ts.Start()**

**Next i**

**' انتظر انتهاء كلّ العمليّات.**

**WaitHandle.WaitAll(waitEvents)**

**End Sub**

وهناك طريقة بديلة لانتظار انتهاء كلّ العمليّات، هي استدعاء الوسيلة Join لكلّ عمليّة!

إنّ بحر العمليّات الفرعيّة Threads شاسع وعميق، ولا يمكن أن نمخر كلّ عُبابِه في هذا الفصل، لهذا فسنكتفي هنا بهذه النبذة التي تتناول بعض الأساسيّات البسيطة، وعلى من يريد الاستزادة أن يفرد شراعه ويبدأ رحلته بنفسه!