**תכנות מקבילי**

רוב האפליקציות צריכות להתמודד עם מספר מטלות המתרחשות בו בעת (Concurrency).

די שכיח שאפליקציה צריכה להתמודד עם מטלה גדולה (או מאוד גדולה) שפתרונה גוזל זמן (או הרבה זמן).

תכנות אסינכרוני ב-C# אינו שונה מהותית ממה שאתם מכירים מפייתון או משפת C++ רק התחביר שונה.

**רב-משימתיות (Multi-Threading):**

מודל תכנות נפוץ המאפשר ל- Threads רבים להתקיים במקביל בתוך יישום, ה-Threads חולקים את משאבי היישום אולם כל אחד רץ בנפרד.   
בפועל - יכולת של יישום, תשתית תכנותית או מערכת הפעלה להריץ בו זמנית מספר מטלות.

**קוד סינכרוני – קוד אסינכרוני**

קוד סינכרוני הוא קוד שכל מטלה שמתבצעת חוסמת את המשך הריצה ואת ביצוע שאר המטלות עד לסיומה (Blocking Code).

קוד הוא אסינכרוני אם הוא מסוגל להתמודד עם מספר מטלות בו בעת מבלי לחסום את שאר המטלות ומבלי להמתין עד שמטלה מסתיימת.  
**לדוגמה:** לקוח שולח בקשות לשרת, טיפול שרת בבקשות של לקוחות, גישה לדיסק, חישובים מורכבים, עיבוד תמונה .....

**קוד אסינכרוני (Async Programming) – קוד מקבילי (Parallel Programing)**

שני דברים שונים אבל מאוד דומים.  
קוד מקבילי הוא תמיד אסינכרוני.  
קוד אסינכרוני (Asynchronous) וקוד מקבילי (Parallel) ממומשים באמצעות Threads.

**בקוד אסינכרוני**, קוד מסוגל להתמודד עם מטלות כבדות באמצעות Threads עצמאיים שאינם חוסמים את שאר התוכנית, לדוגמה: הורדת קובץ גדול מהאינטרנט אינה חוסמת את ה- UI והוא נשאר זמין למתכנת (Responsive UI).

**בקוד מקבילי** הכוונה ביצוע של מטלה גדולה ומורכבת על ידי חלוקתה למספר Threads שרצים במקביל, לדוגמה: בעיבוד תמונה גדולה נחלק את התמונה ל-10 חלקים שווים, כל חלק יטופל על ידי Thread אחר, עם סיומם תוצג התוצאה.

**Threads בשפת C#**

יצירת מטלה הנה תהליך תכנותי פשוט ב- C#. בשלב ראשון נצרף לתוכנית את מרחב השמות System.Threading, מרחב שמות זה מגדיר מספר טיפוסים חשובים.

המחלקה הראשונה שנדון בה היא המחלקה Thread, מחלקה זו אמונה על יצירה וניהול ה- Thread.

מחלקה זו מכילה מספר מתודות ומאפיינים חשובים:

* **Thread.Start()** - הפעלת מתודה זו מייצרת ומריצה Thread חדש בתוכנית, הפעלת Thread משמעותה קריאה למתודה אשר תתבצע במקביל ל- Thread.
* **Thread.Suspend()**- משהה את ביצוע ה- Thread.
* **Thread.Resume()** - מחדשת את ביצוע ה- Thread אשר הושהה על ידי המתודה Thread.Suspend() .
* **Thread.Sleep(int)** - מתודה סטטית, המעכבת את ביצוע ה- Thread מספר milliseconds .
* **Thread.IsAlive** - מאפיין בוליאני המחזיר את מצבו של ה- Thread, true במידה והוא עדיין פעיל, ו- false במידה והוא הסתיים.
* **Thread.Priority** - מאפיין הקובע את סדר הקדימות של ה- Thread, מהטיפוס enum ThreadPriority המגדיר חמישה עדיפויות שונות.

המחלקה Thread דורשת פריט מידע החשוב לפעולתה.

פריט מידע זה הנה מתודה אשר מכילה את הקוד שעל ה- Thread להפעיל.   
המטלה רצה במסגרתה מתודה שמקובל לכנותה "מתודת עבודה" (Worker Method).  
כאשר מטלה מתחילה היא מפעילה את מתודת העבודה, כאשר המתודה מסתיימת, מסתיימת איתה גם המטלה.

המתודה מועברת לאובייקט מהמחלקה Thread על ידי Delegate מיוחד הנקרא **ThreadStart**.

המתודה מופעלת אסינכרונית באמצעות אותו ה-Delegate, התחביר של ה-Delegate:

**public delegate void ThreadStart() ;**

לראשונה נתקלנו ב- Delegate כאשר למדנו על אירועים, זוכרים שה-Delegate מגדיר את החתימה של המתודה?

ולכן במקרה דנן, החתימה של מתודת העבודה חייבת להיות:

void MethodName()

ולכן הגדרת ה- Delegate דורשת שהמתודה לא תקבל ערכים ולא תחזיר ערך.

דוגמה:

מתודת העבודה של ה- Thread

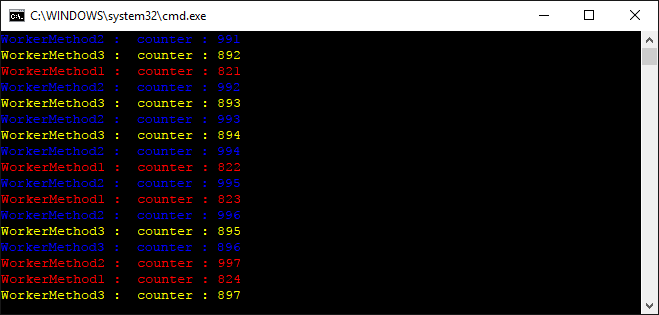
|  |
| --- |
| static void WorkerMethod1()  {  int counter = 0;  for (uint i = 0; i < 1000; i++)  {  Console.ForegroundColor = ConsoleColor.Red;  counter++;  Console.WriteLine("WorkerMethod1 : counter : " + counter);  }  הגדרת ה-Delegate  }  static void Main(string[] args)  {  ThreadStart ts1 = new ThreadStart(WorkerMethod1);  הגדרת ה-Thread  Thread t1 = new Thread(ts1);  t1.Start();  } |

הרצת ה-Thread

דוגמה נוספת:

|  |
| --- |
| static void WorkerMethod1()  מתודת העבודה ראשונה  {  int counter = 0;  for (uint i = 0; i < 1000; i++)  {  Console.ForegroundColor = ConsoleColor.Red;  counter++;  Console.WriteLine("WorkerMethod1 : counter : " + counter);  }  }  static void WorkerMethod2()  מתודת העבודה שניה  {  int counter = 0;  for (uint i = 0; i < 1000; i++)  {  Console.ForegroundColor = ConsoleColor.Blue;  counter++;  Console.WriteLine("WorkerMethod2 : counter : " + counter);  }  }  מתודת העבודה שלישית  static void WorkerMethod3()  {  int counter = 0;  for (uint i = 0; i < 1000; i++)  {  Console.ForegroundColor = ConsoleColor.Yellow;  counter++;  Console.WriteLine("WorkerMethod3 : counter : " + counter);  }  הפעלת Thread ראשון  }  static void Main(string[] args)  {  הפעלת Thread שני  ThreadStart ts1 = new ThreadStart(WorkerMethod1);  Thread t1 = new Thread(ts1);  t1.Start();  ThreadStart ts2 = new ThreadStart(WorkerMethod2);  Thread t2 = new Thread(ts2);  הפעלת Thread שלישי  t2.Start();  ThreadStart ts3 = new ThreadStart(WorkerMethod3);  Thread t3 = new Thread(ts3);  t3.Start();  } |

פלט:



**Foreground and Background Threads**

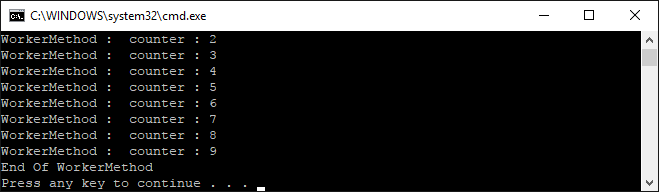
הגדרת Foreground Thread

תהליך יכול לפעול בשני מצבים:

1. Foreground Thread – ברירת המחדל, התוכנית נשארת בחיים כל עוד התהליכים רצים.
2. Background Thread – סיום התוכנית הורג את התהליך.

|  |
| --- |
| static void WorkerMethod()  {  for (int i = 0; i < 10; i++)  {  Console.WriteLine("WorkerMethod : counter : " + i);  }  Console.WriteLine("End Of WorkerMethod");  }  static void Main(string[] args)  {  ThreadStart ts = new ThreadStart(WorkerMethod);  Thread t = new Thread(ts);  t.Start();  Foreground Thread  Console.WriteLine("End Of Main");  } |

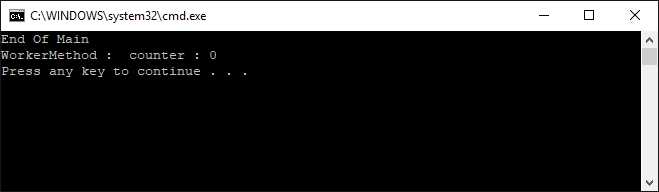
פלט:



התוכנית מסתיימת רק לאחר שה-Thread מסתיים

|  |
| --- |
| static void Main(string[] args)  {  ThreadStart ts = new ThreadStart(WorkerMethod);  Thread t = new Thread(ts);  t.IsBackground = true;  הגדרת Foreground Thread  t.Start();  Console.WriteLine("End Of Main");  } |

פלט:



סיום Main מאלץ גם את סיום ה- Thread

**Threads ב- WPF**

אפליקציה חלונאית במערכת ההפעלה Windows רצה על פי מודל שנקרא STA (Single Thread Apartment).

על פי המודל הזה רק ל-Thread הראשי (התהליך שגם מייצר את ה- UI) יש גישה לפקדים ורק הוא יכול לשנות את המאפיינים שלהם.

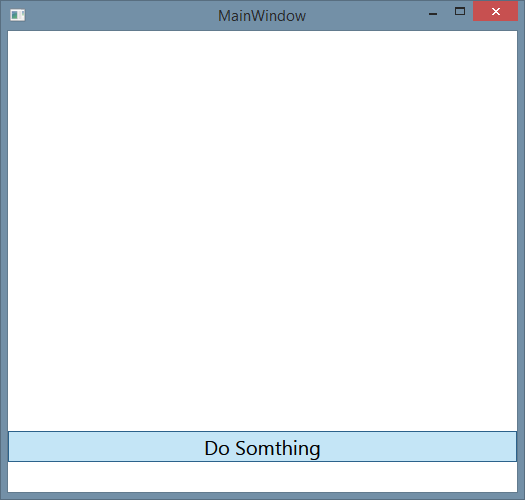
כל משתמש מחשב בעולם חווה ממשק משתמש שנתקע, לוחצים על כפתור שמבצע פעולה שגוזלת זמן ומקבלים את ההודעה "לא מגיב", המשתמש צריך להחליט האם לחכות או פשוט להרוג את התהליך.

מטלות ארוכות ידועות כ-"User Experience Killer"

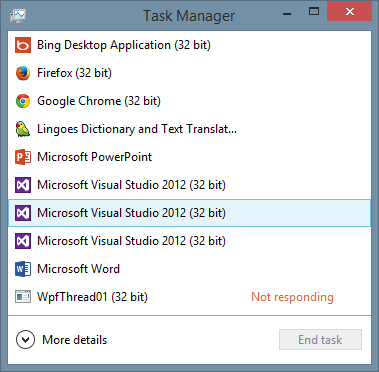
דוגמה מעצבנת - המשימה לצייר 1000 עיגולים על המסך:

|  |
| --- |
| private void Paint()  {  Random rnd = new Random();  Ellipse e;  for (int i = 0; i < 1000; i++)  {  e = new Ellipse();  e.Width = e.Height = rnd.Next(10, 100);  Color color = Color.FromRgb((byte)rnd.Next(256), (byte)rnd.Next(256),  (byte)rnd.Next(256));  e.Fill = new SolidColorBrush(color);  Canvas.SetTop(e, rnd.Next(300));  Canvas.SetLeft(e, rnd.Next(300));  canvas.Children.Add(e);  Thread.Sleep(10);  }  } |

הרצת התוכנית ולחיצה על הכפתור "Do Something" מפעילה את המתודה Paint:

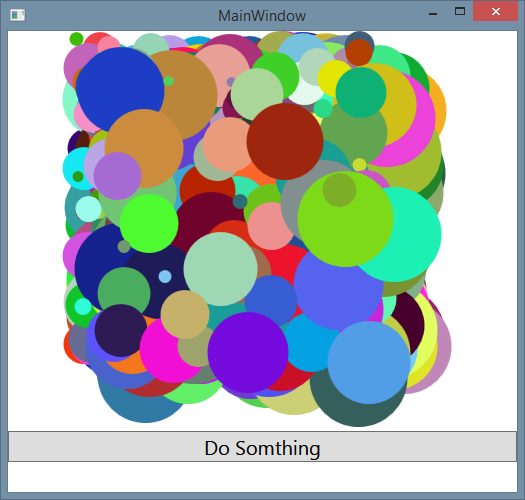


ובינתיים:



Not Responding

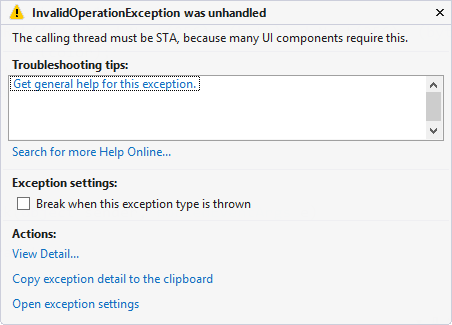
ואחרי כדקה של המתנה:



הפיתרון הוא כמובן להעביר את המטלה ל-Thread נפרד, אבל אם ננסה לממש את האירוע בצורה הבאה:

|  |
| --- |
| private void btnDoSomething\_Click(object sender, RoutedEventArgs e)  {  Thread thread = new Thread(Paint);  thread.Start();  } |

נקבל:



“The calling thread must be **STA**”

**STA** – זו לא מגבלה של הטכנולוגיה בה אנו משתמשים (WPF או WinForms),  
 אלא תכתיב של מערכת ההפעלה.

אז איך מתמודדים?  
באמצעות רכיב הנקרא **BackgroundWorker**.

**BackgroundWorker**

מחלקה שימושית המוגדרת במרחב השמות System.ComponentModel.  
המחלקה מאפשרת העברת מטלות כבדות וארוכות ב-WPF לתהליך נפרד.  
יכולה לדווח על התקדמות התהליך, על סיום או על ביטול.  
מכילה יכולת מובנית לטיפול בחריגים.  
מסוגלת לעדכן רכיבי UI למרות שהמטלה רצה ב-Thread נפרד.

**כיצד משתמשים ב- BackgroundWorker:**

1. הוספת אובייקט מהמחלקה בתכונה של מחלקת החלון (או ב-XAML אם רוצים):

|  |
| --- |
| private BackgroundWorker background\_worker = new BackgroundWorker(); |

1. **הגדרת התכונות** הבאות בבנאי של החלון: מאפשרים ביטול ומאפשרים קבלת דו"ח התקדמות:

|  |
| --- |
| background\_worker.WorkerSupportsCancellation = true;  background\_worker.WorkerReportsProgress = true; |

1. הגדרת האירועים הבאים:
   1. הגדרת האירוע DoWork אשר מועלה כאשר מופעלת המתודה RunWorkserAsync, הטיפול באירוע מממש את הפעולה האסינכרונית שרוצים לבצע ברקע.
   2. האירוע ReportProgress יופעל בכל פעם שנרצה לעדכן את הממשק הן בהתקדמות התהליך והן על מנת לעדכן תוצאות ביניים.
   3. האירוע השלישי, RunWorkerCompleted מועלה כאשר התהליך מסתיים או מבוטל.

|  |
| --- |
| background\_worker.DoWork += background\_worker\_DoWork;  background\_worker.ProgressChanged +=  background\_worker\_ProgressChanged;  background\_worker.RunWorkerCompleted +=  background\_worker\_RunWorkerCompleted; |

1. הרצת התהליך:

|  |
| --- |
| private void btnDoSomething\_Click(object sender, RoutedEventArgs e)  {  background\_worker.RunWorkerAsync(1000);  הפרמטר הנשלח לתהליך  } |

1. מימוש האירוע DoWork, מימוש התהליך אשר ירוץ ברקע.  
   ניתן להעביר פרמטר, הפרמטר הוא מטיפוס Object.  
   יש לבדוק במהלך הביצוע האם התהליך בוטל על מנת לעצור את ריצת התהליך.

הפרמטר שנשלח לתהליך

|  |
| --- |
| void background\_worker\_DoWork(object sender, DoWorkEventArgs e)  {  for (int i = 0; i < (int)e.Argument; i++)  {  if (background\_worker.CancellationPending)  {  e.Cancel = true;  בדיקה האם התהליך לא בוטל  break;  }  Parameters parameter = Paint();  background\_worker.ReportProgress((100 \* i) / (int)e.Argument +  1, parameter);  Thread.Sleep(10);  }  } |

1. הטיפול באירוע ProgressChanged – עדכון ממשק המשתמש:

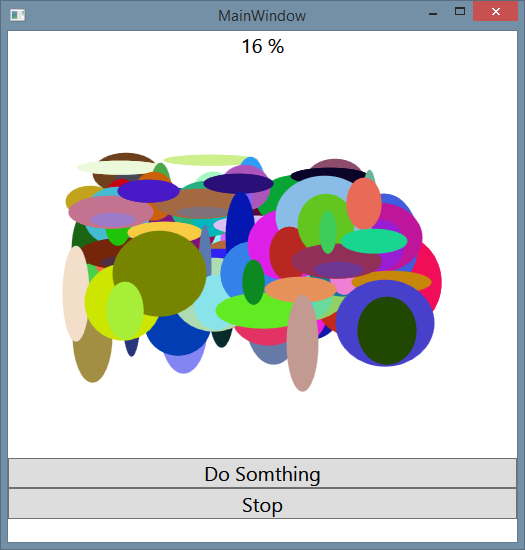
|  |
| --- |
| void background\_worker\_ProgressChanged(object sender,  ProgressChangedEventArgs e)  {  הפרמטר שנשלח לתהליך  Parameters parameter = e.UserState as Parameters;  Ellipse ellipse = new Ellipse();  ellipse.Width = parameter.Size.Width;  ellipse.Height = parameter.Size.Height;  ellipse.Fill = new SolidColorBrush(parameter.Color);  Canvas.SetTop(ellipse, parameter.Location.Y);  Canvas.SetLeft(ellipse, parameter.Location.X);  canvas.Children.Add(ellipse);  txtInterationCounter.Text = e.ProgressPercentage.ToString() +  " %";  }  התקדמות העיבוד. |

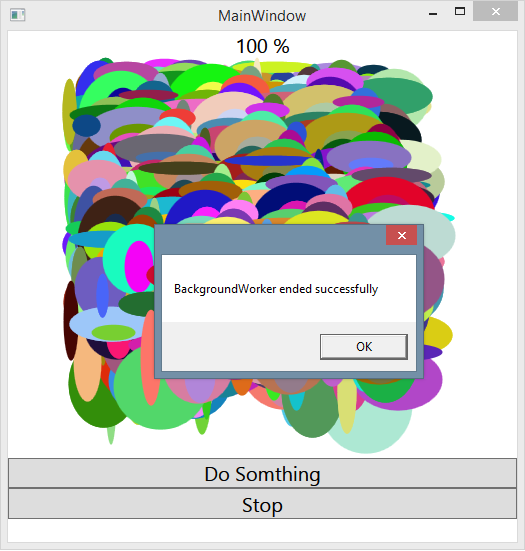
1. סיום התהליך:

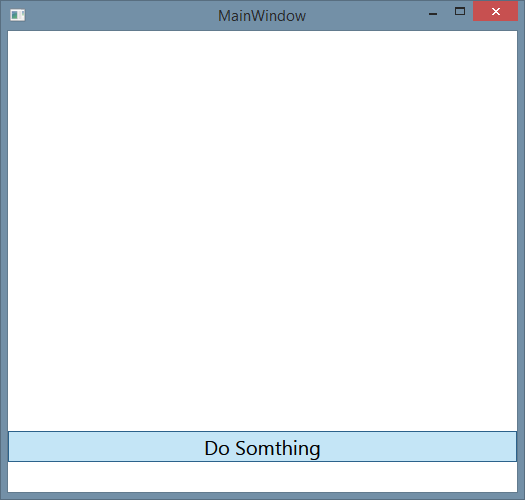
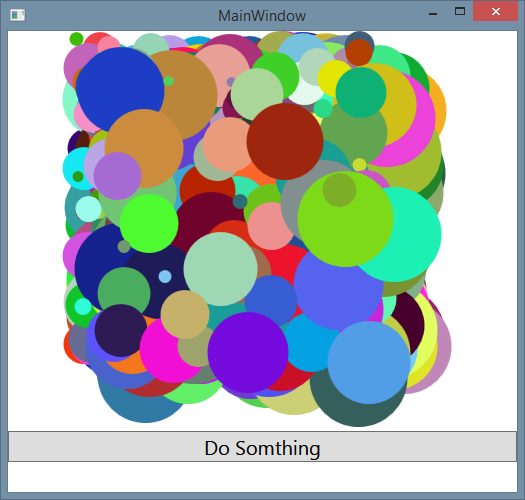
|  |
| --- |
| void background\_worker\_RunWorkerCompleted(object sender,  RunWorkerCompletedEventArgs e  {  if (e.Cancelled)  ביטול.  {  MessageBox.Show("BackgroundWorker canceled");  }  else  {  MessageBox.Show("BackgroundWorker ended successfully");  }  } |

סיום העיבוד.

פלט (הפעם בלי המתנה):







**תקשורת צד לקוח**

.NET Framework מגיעה עם סט מחלקות וטיפוסים גדול המספק מעטפת ליצירת תקשורת בין מחשבים.

החל מהרמה הבסיסית ביותר – סוקטים (Low Level) ועד לתשתיות מעטפת (כמו WCF –Windows Communication Foundation) המספקות יכולת תקשורת מתקדמת המקלה על המתכנת, אם כי מגבילה את היכולות שלו.

מכיוון שאיננו רוצים מגבלות כלשהן שמציבה בפנינו האבסטרקציה של תשתית המעטפת WCF, נתמקד ב- Low Level שמציאה לנו .NET Framework

המחלקה הראשונה שעלינו להכיר היא **TcpClient** המוגדרת במרחב השמות System.Net.Sockets ומספקת לאפליקציית הלקוח יכולות תקשורת ב-TCP.

במחלקה מוגדרים מספר מתודות די פשוטות המאפשרות התחברות לשרת, מאפשרות שליחת וקבלת מידע ממנו.

המחלקה השנייה היא **IPEndPoint** המוגדרת במרחב השמות System.Net, אובייקט מהמחלקה מייצג "נקודת קצה" (End Point) באמצעות כתובת IP ופורט.

המחלקה השלישית היא **NetworkStream** המוגדרת במרחב השמות System.Net.Sockets ומספקת אשר מנהלת עבור אפליקציית הלקוח את העברת הנתונים כמערך של בתים (Stream).

בקוד הדומה נתחבר לשרת, נשלח מידע לשרת, נאזין לו ונקבל ממנו תגובה:

יצירת End Point

התחברות

|  |
| --- |
| TcpClient client = new TcpClient();  IPEndPoint serverEndPoint=new IPEndPoint(IPAddress.Parse("127.0.0.1"), 4242);  client.Connect(serverEndPoint);  יצירת Stream  NetworkStream clientStream = client.GetStream();  byte[] buffer = new ASCIIEncoding().GetBytes("Hello Server!");  clientStream.Write(buffer, 0, buffer.Length);  clientStream.Flush();  המרת מידע למערך של בתים לטובת העברה לשרת  buffer = new byte[4096];  int bytesRead = clientStream.Read(buffer, 0, 4096); |

שליחה לשרת

האזנה וקבלת תגובה מהשרת