תהליכונים (Threads)

תוכן עניינים

מבוא: תקציר מונחים

נושא התהליכים (המשימות) והתהליכונים נידון בהרחבה בקורס מערכות הפעלה. תהליכון – יחידה נפרדת של תהליך (Process) או משימה (Task) המשתמש במשאבים משותפים של המשימה אך מהווה יחידת ריצה נפרדת עם מצביע פקודה משל עצמו. הפעלת תהליכון וניהול פעולתו מתבצעים ע"י מערכת הפעלה או ע"י מכונה וירטואלית (למשל - CLR). תהליכונים שונים של אותה משימה (ושל משימות שונות) יכולים לרוץ במקביל ממש (בסביבה מרובת מעבדים או מרובת ליבות). הם גם יכולים להתבצע במערכות הפעלה עם שיתוף זמן (Time Sharing System) המשתמשות בשיטת הפקעה זמנית ומחזורית של מצב ריצה מתהליכונים ע"פ פסיקות מחזוריות של שעון המערכת – השיטה שמדמה ריצה מקבילית בהרבה היבטים. זאת אומרת – תהליכונים יכולים להיות מופסקים בכל נקודה באמצע ריצה וזמן המעבד יועבר לתהליכון אחר של אותה משימה או של משימה אחרת - ע"פ החלטת מערכת ההפעלה וכמעט ללא שליטה מצד המתכנת. נושאי סנכרון, קטע קריטי, שיטות שונות של העברת מידע בין תהליכים אינם חלק מהקורס שלנו. הנושאים האלה נלמדים בהרחבה בקורס הנדסת מערכות זמן אמת. אנחנו נשתמש בתהליכונים בצורה מוגבלת ונעביר מידע ביניהם בשיטת הודעות (בפועל נעביר מידע בפרמטרים של דלגטים).

שימוש פשוט בתהליכונים

דוגמה להדמיית חשבון בנק ללא תהליכונים

נתחיל מיצירת מחלקה פשוטה עבור חשבון בנק. נכתוב שתי פונקציות ציבוריות – עבור הפקדה ומשיכה של כסף. נכתוב פונקציה פרטית להוספת ריבית עו"ש ע"פ שער מוגדר ופונקציה ציבורית המפעילה אותה כל שלוש שניות בלולאה אין סופית. נייצר בנאי שיריץ את פונקציית הריבית המחזורית וגם נוסיף קצת פונקציות פלט כדי לעקוב אחרי ריצת התוכנית לחלקיה.

```
class Account
{
    private int balance;
    private readonly int interestRate;
    public void Deposit(int amount)
    {
        timeOutput(); out1("Deposit");
        balance += amount;
        out2();
    }
    public bool Withdraw(int amount)
    {
        timeOutput();
        if (amount > balance)
        {
            out1("No withdraw"); out2();
            return false;
        }
        out1("Withdraw");
        balance -= amount;
        out2();
        return true;
    }
    private void applyInterest()
    {
        timeOutput(); out1("applyInterest");
        balance *= 100 + interestRate
        balance /= 100;
        out2();
    }
    public void interestLoop()
    {
        while (true)
        {
            applyInterest();
            Thread.Sleep(3000); // 3sec
        }
    }
}
```

```
public Account (int initBalance, int interestRate)
{
    balance = initBalance;
    interestRate = interestRate;
}
private void timeOutput()
{
    Console.Write("{0}: ",
            DateTime.Now.ToString("HH:mm:ss"));
}
private void out1(string loc)
{
    Console.Write("{0}: old balance = {1}, ",
                loc, balance);
}
private void out2()
{
    Console.WriteLine("new balance = {0}, ",
                    balance);
}
```

בסוף נכתוב תוכנית ראשית שתייצר מופע של חשבון בנק. שימו לב שאין אפשרות לבצע כל פעולה – בגלל שהתהליך מועסק בהרצת הלולאה האין סופית של ריבית מחזורית ולא נותנת אפשרות לכל פעולה נוספת בעזרת קלט כלשהו. וכל זה כי התוכנית רצה בתהליך בעל תהליכון בודד שמוקצה אוטומטית עבור התוכנית הראשית (הרצת פונקציית Main).

```
class Program
{
    static void Main(string[] args)
    {
        Account myAccount = new Account(1000, 2);
        myAccount.interestLoop();
    }
}
```

נשכתב את הבנאי של Account שיכלול את ההפעלה של לולאת הריבית. זו הצורה העדיפה מבחינה הנדסית מכיוון שהריבית הזו הנה פונקציונליות פנימית של החשבון ועדיף להפעילה אוטומטית בבניית החשבון ולא מבחוץ (בתוכנית הראשית) כמו קודם. עכשיו הבנאי והתוכנית הראשית ייראו בצורה הבאה.

```
public Account (int initBalance, int interest)
{
    balance = initBalance;
    interestRate = interest;
    interestLoop();
}
```

```
    static void Main(string[] args)
    {
        Account myAccount = new Account(1000, 2);
    }
```

הוספת שימוש תהליכון להדמיית חשבון בנק – הכרת מחלקת Thread

איך נוכל לפתור את הבעיה? בואו נראה איך אפשר להריץ את פונקציית הריבית המחזורית במקביל לתוכנית הראשית – ונפנה את התוכנית הראשית לקליטת הנחיות הפקדה ומשיכה. נשתמש בשביל זה במחלקה מיוחדת שנקראת Thread (נקרא לה בינתיים "המחלקה"). כבר ראינו אותה כשהיינו צריכים להשהות את התוכנית לשלוש שניות – ע"י פונקציה סטטית של המחלקה בשם Sleep שמשהה את התהליכון הנוכחי לפרק זמן מוגדר. כדי שנוכל להשתמש במחלקה זאת הוספנו כבר קודם "using System.Threading;".

בשביל לבנות תהליכון חדש – נצטרך ליצור מופע חדש של המחלקה. בואו נתמקד קודם כל בבנאים של המחלקה. למחלקה יש ארבעה בנאים – נתעלם משניים מהם שמגדירים בצורה מפורשת את גודל המחסנית עבור תהליכון.

```
public Thread(ThreadStart start);
public Thread(ParameterizedThreadStart start);
```

שימו לב שהבנאים מקבלים פרמטר דלגט – זאת תהיה הפונקציה שתרוץ בתהליכון (בדומה לפונקציית Main בתהליכון הראשי של התוכנית). הנה הגדרות הדלגטים האלה:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object obj);
```

מה שאנחנו רואים, שבשביל הבנאי הראשון נוכל לתת כל פונקציית void ללא פרמטרים. בשביל הבנאי השני נוכל להשתמש בפונקציית void עם פרמטר מסוג object – למעשה זה מאפשר להעביר פרמטר מכל סוג. בעזרת יצירת אובייקט והפעלת הבנאי אנחנו מקבלים מופע של תהליכון אך עדיין זה לא גורם להרצתו. נוכל להגדיר משתנה לשמירת ההפניה למופע זה. נריץ אותו בעזרת פונקציית מופע של מחלקת Thread – פונקציית Start. אם השתמשנו בדלגט ללא פרמטרים – לא נעביר ארגומנטים נוספים גם ל-()Start. במקרה השני שכולל פרמטר נוסף – נעביר את הארגומנט דרך פונקציית (Start(myParameter. מכיוון שכל סוג ב-C# יורש בפועל מ-Object – נוכל להעביר ארגומנט מכל סוג. הפונקציה שהעברנו לבנאי תקבל אותו לפרמטר ותפעל בהתאם. ברור

שנעביר ארגומנט מהסוג שהפונקציה שלנו מצפה לקבל. בתוך הפונקציה נרצה להמיר את הפרמטר מסוג Object לסוג שאנחנו נרצה להעביר.

```
Thread myThread = new Thread(interestLoop);
MyThread.Start();
```

בהרבה המקרים לא נזדקק לעשות שום דבר עם המשתנה הזה. לכן נפעיל את התהליכון בצורה מקוצרת:

```
new Thread(interestLoop).Start();
```

מהרגע הזה נוצר ורץ לנו תהליכון נפרד שמתחיל בפונקציית interestLoop במקביל להתוכנית הנוכחית – היא בתורה ממשיכה את ריצתה מהביטוי הבא. הבנאי שלנו ייראה מעכשיו כדלקמן.

```
public Account (int initBalance, int interest)
{
    balance = initBalance;
    interestRate = interest;
    new Thread(interestLoop()).start();
}
```

עכשיו נעדכן את הבנאי עוד פעם ונפעיל את הפונקציה בתהליכון נפרד כפי שתיארנו קודם.

אחרי זה נשאר לנו רק להשלים את התוכנית הראשית כדלקמן. נוסיף לה פונקציונליות שתאפשר פעולות נוספות במקביל לריצת לולאת הריבית המחזורית. בלחיצת מקש '1' נפקיד סכום רנדומלי בין 0 ל-99, ובלחיצת מקש '2' נמשוך סכום רנדומלי בין 0 ל-199.

```
class Program
{
    static private Random rand = new Random();

    static void Main(string[] args)
    {
        Account myAccount = new Account(1000, 2);

        while (true)
        {
            ConsoleKeyInfo keyInfo = Console.ReadKey(true);
            switch (keyInfo.KeyChar)
            {
                case '1':
                    myAccount.Deposit(rand.Next(100));
                    break;
                case '2':
                    myAccount.Withdraw(rand.Next(200));
                    break;
            }
        }

    }
}
```

עכשיו נריץ ונוכל לראות את כל השינויים קורים במקביל. (נלחץ מידי פעם על "1" או על "2" בתוך החלון של קונסול).

בשיטה שלמדנו נוכל ליצור כל כמות נדרשת של תהליכונים. התהליכון שאנחנו מייצרים לא חייב לבצע לולאה אין-סופית. זאת יכול להיות גם פונקציה שמבצעת את משימתה ומסתיימת ב-return או סתם בסוף הפונקציה. במקרה כזה התהליכון יסתיים גם. המופע של ה-Thread ייאסף וייהרס ע"י Garbage Collector אלא אם שמרנו הפניה אליו. אבל גם במקרה כזה לא נוכל להריץ מחדש את התהליכון שכבר עשה את משימתו והסתיים. קריאה לפונקציית Start במצב כזה תגרום לחריגה ThreadStateException.

עצירת תהליכון – עצירה כוחנית ועצירה "חיננית"

השאלה הבאה היא – איך נעצור את התוכנית (ואת התהליכון) כשנרצה. הדרך הפשוטה היא לשמור את ההפניה לאובייקט Thread של התהליכון שרוצים לעצור, ולקרוא ממנו פונקציה Abort()... אך זו שיטה כוחנית שלא מאפשרת לסגור ולשחרר את המשאבים שבשימוש התהליכון שסוגרים אותו. לכן הפונקציה הזו הוכרזה כ-

```
private volatile bool _shouldStop;
public void Close()
{
    timeOutput(); Console.WriteLine("closing");
    _shouldStop = true;
}
public bool threadFinished(bool sync)
{
    timeOutput();
    if (myThread == null)
    {
        Console.WriteLine("threadFinished: no thread");
        return true;
    }
    if (sync)
    {
        Console.WriteLine("threadFinished: joining");
        myThread.Join();
        timeOutput();Console.WriteLine("threadFinished: true");
        return true;
    }
    bool t = !myThread.IsAlive;
    Console.WriteLine("threadFinished: {0}", t);
    return t;
}
```

deprecated (בתרגום ישיר לעברית – מגונה) והשיטה מקובלת לעצירת תהליכון המריץ לולאה אין סופית היא ע"י "דגל" – שדה שנבדק כל פעם בלולאה כתנאי יציאה. נגדיר את השדה בצורה שתוודא גישה בטוחה אליו – נשתמש במילת מפתח ***volatile***. משמעות מילת המפתח הזו (שמשמעותה "נדיף") היא למנוע מהקומפיילר לעשות אופטימזציה של שימוש בשדה ומכריחה את הקומפיילר לבצע קריאת ערך השדה מהזיכרון של פעם. הכוונה היא שבלי זה המהדר יכול להחליט שאם הלולאה קצרה כדאי לו להחזיק את הערך של השדה ברגיסטר (אוגר) של המעבד ולא לטעון אותו כל פעם מהזיכרון כדי לחסוך זמן ריצה.

```
public void interestLoop()
{
    myThread = Thread.CurrentThread;
    _shouldStop = false;
    while (!_shouldStop)
    {
        applyInterest();
        Thread.Sleep(3000); // 3 seconds
    }
    Thread.Sleep(5000);  // 5 seconds delay
}
```

נלמד באותה הזדמנות איך לבדוק האם התהליכון הסתיים או לאו. נתחיל ממאפיין סטטי של מחלקת Thread שמאפשר לקבל את ההפניה לתהליכון המתבצע – **CurrentThread**. למופע של תהליכון יש שתי פונקציות – פונקציה ראשונה סינכרונית – הממתינה לסיום התהליכון – **Join()**. והפונקציה השנייה – אסינכרונית – שמחזירה ***true*** אם התהליכון עדיין פועל ו-***false*** אם התהליכון הסתיים. יש לבדוק שהתהליכון קיים לפני הפעלת הפונקציות.

```
    static void Main(string[] args)
    {
        Account myAccount = new Account(1000, 2);

        while (!myAccount.threadFinished(false))
        {
            ConsoleKeyInfo keyInfo = Console.ReadKey(true);
            switch (keyInfo.KeyChar)
            {
                ...
                case '0':
                    myAccount.Close();
                    break;
            }
        }
    }
```

נוכל עתה להוסיף אפשרות עצירה לתוכנית הראשית – אך נמתין עד שהתהליכון של ריבית מחזורית יסתיים. נעשה זאת בעזרת פונקציה **Close** של ה-**Account**, שבעצם מדליקה את הדגל שדיברנו עליו: **_shouldStop**.

תהליכון "אנונימי" – שימוש בביטוי לאמבדה

אפשרות נוספת ויותר מתקדמת עם שימוש בביטויי למבדה שמאפשר להריץ קטעי קוד בתהליכון נפרד בלי לכתוב פונקציה מיוחדת – בשיטת in-line.

```
new Thread(() =>
{
    // Code to be performed in a separate thread
    ............
}).Start();
```

למשל, נוכל להכניס את הקוד של לולאת ריבית מחזורית לתוך בנאי מחלקת חשבון בנק ללא פונקציה מיוחדת (שאפשר לבטלה עתה):

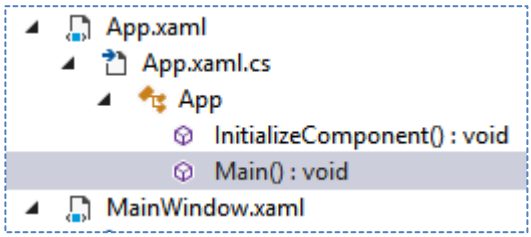
```
public Account (int initBalance, int interest)
{
    balance = initBalance;
    interestRate = interest;
    new Thread(() =>
    {
        myThread = Thread.CurrentThread;
        _shouldStop = false;
        while (!_shouldStop)
        {
            applyInterest();
            Thread.Sleep(3000); // 3 seconds
        }
        Thread.Sleep(5000);  // 5 seconds delay
    }).Start();
}
```

עכשיו נוכל להריץ את התוכנית הקטנה של הדמיית פעילות חשבון בנק. במסך של הקונסול נוכל ללחוץ מקשים 0 עד 2 (0 יסיים את התוכנית – בהדרגה כפי שתיארנו).

תכנות אסינכרוני בסביבת WPF

תהליכון ראשי של אפליקציה WPF ומושג STA

בסביבת הממשקים הגרפים יש חשיבות מיוחדת לעבודה עם התהליכונים. כפי שראיתם קודם – כשאתם בונים פרוייקט של WPF, התוכנית הראשית (פונקציה **Main** במודול **App**, בקובץ שמייוצר ע"י המערכת **App.g.i.cs** – ניתן לראות אותו עם ב-Solution Explorer לוחצים על המשולש להרחבת **App.xaml**, תחתיו מרחיבים **App.xaml.cs**, בתוכו מרחיבים את המחלקה **App**, ואז עושים הקשה כפולה על הפונקציה **Main** שבמחלקה) בנוייה בצורה מיוחדת.



קודם כל לפני פונקציית **Main** אנחנו רואים אטריבוט **[STAThread]** שמשמעותו קשור לצורת התהליכון.

```
/// <summary>
/// Application Entry Point.
/// </summary>
[System.STAThreadAttribute()]
[System.Diagnostics.DebuggerNonUserCodeAttribute()]
[System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks",
"4.0.0.0")]
public static void Main() {
    WpfLesson1.App app = new WpfLesson1.App();
    app.InitializeComponent();
    app.Run();
}
```

ככה נראה ההגדרה של האטריבוט **STAThread**:

```
// Summary:
//     Indicates that the COM threading model for an application is single-
threaded
//     apartment (STA).
[AttributeUsage(AttributeTargets.Method)]
[ComVisible(true)]
public sealed class STAThreadAttribute : Attribute
{
    public STAThreadAttribute();
}
```

WPF משתמש בתבנית STA (Single-Threaded Apartment) בגלל שזה המבנה הנדרש עבור Win32 אשר בבסיסו Apartment-Threaded. ההגדרה הזו אומרת שכל חלון יכול להיבנות בכל תהליכון – אבל מאותו הרגע החלון לא יכול לעבור בין תהליכונים וכל פונקציות הפקדים שלו והגישה למאפייניהם חייבים להתבצע באותו התהליכון בלבד. התהליכון הראשי של פרויקט WPF– הוא גם מתפקד כתהליכון UI של החלון הראשי.

כמו כן נראה את השורה הבאה בסוף פונקציית **Main**: "`app.Run();`". מחלקת **App** יורשת את הפונקציה **Run** ממחלקה **Application**. נלמד מה המשמעות שלה.

```
// Summary:
//     Starts a Windows Presentation Foundation (WPF) application.
// ...
public int Run();
```

ובהערות על הפונקציה שבתיעוד באתר המפתחים של מיקרוסופט:

Remarks

Run is called to start a WPF application. ...

*When Run is called, Application **attaches a new Dispatcher instance to the UI thread**. Next, the **Dispatcher object's Run method is called**, **which starts a message pump to process windows messages**. Finally, the Dispatcher object calls the Application object's the OnStartup method to raise the Startup event. ...*

או בלשוננו, הפונקציה הנ"ל מייצרת מופע של **מַשְגֵר** (**Dispatcher**) עבור התהליכון הראשי של האפליקציה.

מחלקה ואובייקט Dispatcher

התפקיד של האובייקט הזה הנו לנהל ולשגר הודעות מערכת החלונות ופריטים נוספים לתור האירועים של החלון, לנהל את התור הזה ולהפעיל את כל events handlers המתאימים (כולל כאלה שהוספתם ב-xaml ו-\או בקוד שלכם) בשביל כל אירוע. כך למעשה לכל תהליכון UI, כולל התהליך הראשי של האפליקציה, יש משגר צמוד שאנחנו יכול לגשת אליו בעזרת מאפיין **Dispatcher** של האפליקציה או של החלון. המאפיין זה משוכפל לכל הפקדים שבהיררכיית האלמנטים הגרפיים של החלון. זאת אומרת בכל אובייקט של פקד יש לנו מאפיין **Dispatcher** שנותן גישה למשגר של תהליכון UI האחראי על הפקד. שימו לב שלמחלקה הזו ולשם המאפיין שבפקדים יש אותו שם בדיוק. אל תתבלבלו.

מחלקה **Dispatcher** מוגדרת במרחב השמות **System.Windows.Threading**, וכדאי להוסיף אותו ברשימת ה-**using** של הקבצים שלנו ששם נרצה להשתמש בפונקציונליות ריבוי התהליכונים במודולים של WPF. כפי שהזכרנו קודם, המחלקה אחראית על ניהול תור הפריטים שצריך לעבד בתהליכון שהמשגר מוצמד אליו, כפי שמופיע בתיעוד: "*Provides services for managing the queue of work items for a thread.*". כאשר אנחנו בונים ומצמידים מופע של משגר לתהליכון – המופע הזה הופך להיות המשגר היחיד בשביל התהליכון, גם עם המופע מת בטעות... המופע נוצר כאשר אנחנו פונים למאפיין סטטי CurrentDispatcher של מחלקה Dispatcher שמחזיר לנו את המשגר של התהליכון. המאפיין בודק האם כבר נוצר מופע המשגר עבור התהליכון הנוכחי. אם עדיין לא נוצר המופע – הוא ייוצר. למעשה המחלקה Dispatcher והמאפיין CurrentThread שלה מממשים תבנית מולטיטון (Multiton) עבור מופעי המשגר לפי תהליכון.

כפי שכבר אמרנו קודם – בהפעלת אפליקציית WPF מופעלת פונקציה **Run** של המשגר של התהליכון הראשי שלה. התפקיד שך הפונקציה – הפעלת לולאת ביצוע על תור הפריטים של התהליכון (שגם נקרא תור המשגר – Dispatcher's event queue):

*Pushes the **main execution frame** on the event queue of the Dispatcher.*

Remarks

*The Dispatcher processes the event queue in a loop. **The loop is referred to as a frame**. ... The main execution frame will continue until the Dispatcher is shutdown.*

שימו לב שאם אנחנו רוצים לסיים את התהליכון – עלינו לעצור את המשגר שלו לפני כן. אנחנו נבקש את סיום המשגר ע"י אחת משתי הפונקציות הבאות של מופע המשגר: **BeginInvokeShutdown(***DispatcherPriority***)** לשליחת בקשת סיום אסינכרונית או **InvokeShutdown()** לעצירה סינכרונית של המשגר. כאשר מדובר בסגירת כל האפליקציה, המערכת של WPF דואגת בעצמה לסגירת משגר האפליקציה.

בחזרה לולאת הביצוע של המשגר. זאת לולאה אין סופית (כמעט – עד שהיא מקבלת הוראת סיום כנ"ל) שבתחילת הלולאה היא נכנסת להמתנה להודעה\לפריט הבא לעיבוד התהליכון. לאחר קבלתו, מנותחת ההודעה\הפריט ומופעלים ה-event-ים המתאימים של הפקדים המתאימים ומופעלות הפונקציות המתאימות. לפעמים מדובר בהפעלת סדרה שלמה של event-ים – כפי שרואים באירוע הלחיצה על כפתור העכבר. תור הפריטים של המשגר הנו תור עם עדיפויות. זאת אומרת – פריט בעל עדיפות גבוהה יותר יילקח מהתור לפני פריט בעל עדיפות נמוכה יותר, גם אם הראשון הגיע לתור אחרי השני.

רמות העדיפות של אירועי Dispatcher

קיימות מספר רמות עדיפות בתור הזה. ערך מספרי של העדיפות גבוה יותר מהוה עדיפות גבוהה יותר. העדיפויות מוגדרות ע"י סוג enum בשם **DispatcherPriority** כדלקמן (יש גם ערך Invalid – המספר שמאחוריו זה 1-):

0. Inactive – הפריטים לא מעובדים (לא עושים אתם כלום)
1. SystemIdle –המערכת במצב Idle (זאת אומרת כאשר אין משהו אחר לעשות במערכת)
2. ApplicationIdle – האפליקציה במצב "Idle" – אין לאפליקציה כרגע מה לעשות
3. ContextIdle – הביצוע לאחר כל משימות הרקע של המשגר
4. Background – משימות הרקע עבור המשגר
5. Input – אותה העדיפות כמו באירועי קלט בפקדים – לחיצות, הקלדות וכו'
6. Loaded – בין השלמת התסדירים וציור (rendering) ה-GUI לבין אירועי קלט, למשל אירוע Loaded של הפקדים ושל החלון
7. Render – באותה רמת העדיפות כמו ביצוע ציור (rendering) ה-GUI
8. DataBind – אותה רמת העדיפות כמו ביצוע קישור הנתונים לפקדים – אכלוס הנתונים מהנתונים המקושרים לתוך מאפייני הפקדים ולהפך
9. Normal – רמת ברירת המחדל לפעולות שתבקשו בקוד שלכם (נראה בהמשך) אם לא פירשתם עדיפות
10. Send – העדיפות הגבוהה ביותר לבקשות מיוחדות שתתבצענה לפני הבקשות הרגילות

נ.ב. בגרסה הנוכחית של WPF אין כרגע שום אירועי ם פנימיים המיוחסים לעדיפות ApplicationIdle ו-ContextIdle ואין הגדרה ממשית למצבים שמתוארים עבורם. עדיין אלו עדיפויות חוקיות לסדר את בקשותיכם בקוד בעדיפות הנמוכה מעדיפות משימות רקע.

אם אתם שולחים למשגר פעולה\בקשה בעדיפות Send כאשר אתם בתהליכון המקושר למשגר הזה – הפעולה תתבצע באופן ישיר ומידי ללא הכנסה לתור.

מהרשימה הנ"ל אנחנו גם למדים את סדר התייחסות המערכת לאירועי גרפיקה וקלט במערכת ה-WPF.

מה הצורך בשימוש בתהליכונים ב-WPF?

שימו לב: המשגר לא יעצור באמצע שום פעולה שלוקחת יותר מידי זמן. לכן עם בקוד של event handlers שלכם או בפונקציות שאתם תשלחו לביצוע במשגר יש פעולות כבדות, המתנה למשהו, כניסה למצב Sleep לפרק זמן – הדבר יגרום תקיעה של כל המערכת ושל ה-GUI עד להשלמת הפעולה!

לכן אם יש לנו פעולות כאלה שאנחנו רוצים לבצע באפליקציית WPF – אנחנו נעשה זאת בתהליכונים נפרדים – וזאת הסיבה העיקרית שבגללה אנחנו לומדים את הנושא! נוכל לעשות זאת ע"י השיטות שלמדנו בפרק הראשון וע"י שיטות נוספות שנרחיב עליהם בהמשך.

נוכל גם לפתוח חלונות חדשים בתהליכון נפרד. למשל אפשר לעטוף את הקוד הכולל הקפצת חלון הודעה (MessageBox) בתוך תהליכון, ואז החלון שלנו לא ייחסם עד לסגירת ההודעה הזו והקוד שכתוב אחרי השורה הבאה ימשיך להתבצע במקביל לפתיחה, עיון וסגירה של ההודעה:

```
new Thread(() => {MessageBox.Show(......);}).Start();
```

נוכל גם להפריד משימות האורכות זמן מסוים והשהיות לתהליכון נפרד כפי שהזכרנו קודם. למשל, דוגמת פונקציה לביצוע משימה במקרה של אזעקה קריטית (בשביל תרגיל 3):

```
private void criticalEvent(PrinterUserControl printer, Action action)
{
    // Action1: Send technician to fill ink or paper - it takes some time to get to
    //          the printer therefore lets do it in a separate thread...
    new Thread(() =>
    {
        Thread.Sleep(random.Next(MIN_TECHNICIAN_DELAY, MAX_TECHNICIAN_DELAY));
        action();
    }).Start();

    // Action2: After a sanity check (to avoid event collisions *),
    //          move to the next available printer by enqueuing
    //          the current printer and dequeuing the next one.
    // *) Printer can generate multiple critical events (when both ink and pages are zero)
    //    We don't want to do it multipletimes. We do it just in the first event.
    if (printer == CurrentPrinter)
    {
        queue.Enqueue(CurrentPrinter);
        CurrentPrinter = queue.Dequeue();
    }
}
```

הפרמטר הדלגט action בדוגמה הזאת יקבל כארגומנט פונקציית הוספת דיו או פונקציית הוספת נייר. במקרה של אזעקה קריטית מהמדפסת – במקום שנשכפל את הקוד הזה ב-handlers של אירועי המדפסת, נוכל לקרוא לפונקציה הנ"ל. עכשיו נחשוב מה יקרה... אנחנו הפעלנו את הפעולה הנ"ל מתוך תהליכון שאיננו התהליכון של חלון המכיל בקרים של דיו ושל נייר. בתוך פונקציית הפעולה ננסה לעדכן את כמות הנייר או כמות הדיו. כתוצאה – נפר את הדרישה של STA שדיברנו קודם – שגישה לכל הפונקציות והמאפיינים של הפקדים חייבים להתבצע רק מתהליכון UI של החלון בלבד.

הפיכת הדמיית חשבון לאפליקציה WPF

לפני שנמשיך בהסבר – נשכלל את תוכנית חשבון הבנק שלנו. קודם כל בואו נהפוך את התוכנית של חשבון הבנק לתוכנית WPF. נבטל את כל הפלט לקונסול. נוסיף כמה בקרות שמועילות בהמשך (לא נתמקד בהם בשיעור – תחקרו בעצמכם). נוסיף גם תבנית Observer – אירוע (Event) עבור עדכון של הצגת החשבון בחלון. ראו את מודול המחלקה של חשבון בנק להלן. נעתיק את מחלקת הארגומנטים של אירוע המדפסת:

```
namespace ThreadsWindow
{
    class AccountEventArgs : EventArgs
    {
        private int balance;
        public int Balance { get { return balance; } private set { balance = value; } }
        public AccountEventArgs(int balance)
        {
            Balance = balance;
        }
    }
}
```

נעתיק ונרחיב את פונקציונליות מחלקת החשבון כדי שנוכל לסגור אותו עם ובלי סגירת החלון (נבין בהמשך):

```
namespace ThreadsWindow
{
    class Account
    {
        public enum AccountState { RUNNING, STOPCLOSED, STOP }
        private const int CLOSED = -999999;

        private int balance;
        private int Balance
        {
            get {  return balance; }
            set
            {
                if (balance == CLOSED) return;
                if (balance != value)
                {
                    balance = value;
                    BalanceChangedHandler(value);
                }
            }
        }

        private readonly int interestRate; // integer % number
        private Thread myThread;
        private volatile AccountState _shouldStop;
        public Account(int initBalance, int interestRate)
        {
            this.Balance = initBalance;
            this.interestRate = interestRate;
            new Thread(() =>
            {
                myThread = Thread.CurrentThread;
                _shouldStop = 0;
                while (_shouldStop == AccountState.RUNNING)
                {
                    applyInterest();
                    Thread.Sleep(3000); // 3 secs
                }
                Thread.Sleep(5000); // 5 secs delay
                if (_shouldStop == AccountState.STOPCLOSED)
                    Balance = CLOSED;
            }).Start();
        }

        public void Deposit(int amount)
        {
            Balance += amount;
        }

        public bool Withdraw(int amount)
        {
            if (amount > Balance) return false;
            Balance -= amount;
            return true;
        }
```

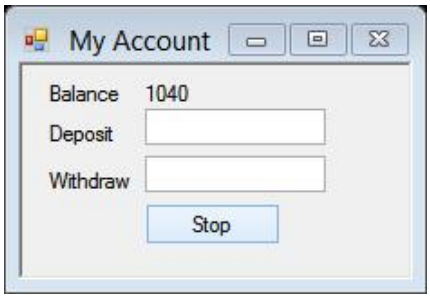
```
        private void applyInterest()
        {
            Balance = (Balance * (100 + interestRate)) / 100;
        }

        public void Close(bool upd)
        {
            _shouldStop = upd ? AccountState.STOPCLOSED : AccountState.STOP;
        }

        public bool threadFinished(bool sync)
        {
            if (myThread == null)
                return true;
            if (sync)
            {
                myThread.Join();
                return true;
            }
            bool t = !myThread.IsAlive;
            return t;
        }

        public event EventHandler<AccountEventArgs> BalanceChanged;
        void BalanceChangedHandler(int balance)
        {
            if (BalanceChanged != null)
            {
                new Thread((object obj) => BalanceChanged(this, (AccountEventArgs)obj)
                           ).Start(new AccountEventArgs(balance));
            }
        }
    }
}
```

ונגדיר חלון פשוט עבור התצוגה. הוא יכיל תצוגת מאזן, שתי שדות עריכה של הפקדה ומשיכה (ע"י הכנסת ערך ולחיצת מקש Enter) וכפתור Stop לעצירת התהליכון של ריבית מחזורית. להלן תמונת החלון ורשימת הפקדים שבו (ככה ייראו הגדרות הפקדים בקובץ האוטומטי שמיוצר ע"י WPF עבור החלון):




```
private System.Windows.Button btnStop;
private System.Windows.TextBox textBox1;
private System.Windows.Label lblBalance;
private System.Windows.Label txtBalance;
private System.Windows.Label lblDeposit;
private System.Windows.Label lblWithdraw;
private System.Windows.TextBox textBox2;
```

נוסיף לחלון כמה פונקציות. נתחיל מפונקציה באירוע טעינת החלון – הפונקציה תאתחל את חשבון הבנק ותרשום את "המשקיף" (Observer) לאירוע שינוי מאזן, פונקציית לחיצה על הכפתור שתעצור את תהליכון הריבית המחזורית תוך עדכון המאזן ל-999999- ע"פ מה שראיתם לעיל במחלקת חשבון הבנק. נייצר גם

פונקציה של אירוע סגירת החלון – שגם תנסה לעצור את תהליכון הריבית המחזורית וגם תמתין עד סיומו של התהליכון הזה. נוסיף את "המשקיף" שהזכרנו.

```
namespace ThreadsWindow
{
    public partial class WindowAccount : Window
    {
        public WindowAccount()
        {
            InitializeComponent();
        }

        private Account myAccount;

        private void Window_Loaded(object sender,
                                   RoutedEventArgs e)
        {
            myAccount = new Account(1000, 2);
            myAccount.BalanceChanged +=
                            windowAccountObserver;
        }

        private void btnStop_Click(object sender,
                                   RoutedEventArgs e)
        {
            if (myAccount != null)
                new Thread(() =>
                                myAccount.Close(true)
                            ).Start();
        }
        private void Window_Closing(object sender,
                                    System.ComponentModel. CancelEventArgs e)
        {
            if (myAccount != null)
            {
                myAccount.Close(false);
                myAccount.threadFinished(true);
            }
        }

        private void windowAccountObserver(object sender, AccountEventArgs args)
        {
            updateBalance(args.Balance);
        }

        private void updateBalance(int balance)
        {
            txtBalance.Text = String.Format("{0}", balance);
        }

    }
}
```

לפני שנוסיף יישום לשדות עריכה – ננסה להריץ את התוכנית... התוכנית תקרוס – נקבל חריגה מתוך פונקציה **updateBalance**. חריגה הנקראת **InvalidOperationException**. כמו כן מידע נוסף שנעשה ניסיון גישה לפקד **txtBalance** מתהליכון שונה מזה שבנה את הפקד. נתקלנו בבעיה של STA.

פונקציות שיגור משימות לתור של Dispatcher

מחלקה DispatcherObject כאב קדמון לכל הפקדים של WPF

כדי לפתור את הבעיה, נשתמש בפונקציות נוספות שקיבלנו מ-WPF. חלק ראשון מהפונקציות קיבלנו בכל פקד WPF כתוצאה מירושה ממחלקה DispatcherObject (שמוגדרת במרחב השמות System.Windows.Threading). המחלקה הזו מוסיפה כמה דברים מועילים לקישור בין הפקד למשגר שלו:

```
public abstract class DispatcherObject
{
    protected DispatcherObject();
    // Summary:
    //     Gets the System.Windows.Threading.Dispatcher this DispatcherObject
    //     is associated with.
    [EditorBrowsable(EditorBrowsableState.Advanced)]
    public Dispatcher Dispatcher { get; }

    // Summary:
    //     Determines whether the calling thread has access to this DispatcherObject.
    // Returns:
    //     true if the calling thread has access to this object; otherwise, false.
    [EditorBrowsable(EditorBrowsableState.Never)]
    public bool CheckAccess();

    // Summary:
    //     Enforces that the calling thread has access to this DispatcherObject.
    // Exceptions:
    //   T:System.InvalidOperationException:
    //     the calling thread does not have access to this DispatcherObject.
    [EditorBrowsable(EditorBrowsableState.Never)]
    public void VerifyAccess();
}
```

הסבר:

1. קודם כל כאן אנחנו רואים מעין מקבל כל פקד את הגדרת המאפיין **Dispatcher** שדיברנו עליו קודם.
2. פונקציות **CheckAccess** ו-**VerifyAccess** מאפשרות לבדוק האם הקוד כרגע מתבצע בתוך תהליכון UI של הפקד המבוקש (או של החלון שלנו – אם אנחנו בתוך הקוד שמוגדר במחלקת החלון שלנו). ההבדל ביניהם שהפונקציה הראשונה מחזירה ערך **true** או **false** בהתאם לתשובה, כאשר הפונקציה השנייה לא מחזירה כלום אלא זורקת חריגה **InvalidOperationException** אם אנחנו כרגע לא בתוך תהליכון UI.
3. שימו לב על האטריבוט EditorBrowsable(EditBrowsable.Never). האטריבוט הזה גורם לכך שה-IntelliSense של ויז'ואל סטודיו לא מראה לנו את הפונקציות ברשימת פונקציות האובייקט ולא משלים את השם שלהן כשאנחנו מתחילים להקליד אותו... מיקרוסופט החליטה שהונקציות האלה רק למתכנתים מתקדמים... מפתחים רבים מאד בעולם חולקים על הקביעה הזו של מיקרוסופט. פשוט אתם צריכים להיות מודעים שצריך לזכור את הפונקציות במדויק או להסתכל ב-DispatcherObject לתזכורת כשאתם צריכים להשתמש בהן, כי ויז'ואל סטודיו לא יעזור לכם במקרה הזה.

הרחבה על מחלקת Dispatcher – השדות והפונקציות

חלק שני מהפונקציות קיבלנו במופע המשגר המקושר לתהליכון UI של הפקד. אנחנו מגיעים אליהם בעזרת המאפיין הנ"ל Dispatcher של כל פקד ופקד (לא הצגתי חלק מהאיברים המתקדמים שלא חשובים לנו בקורס, קצת שינית את הסדר, מחקתי אטריבוטים הקשורים להצגה או השלמה של האבירים בעריכה או קשורים לאבטחת תכנה):

```
// Summary:
//     Provides services for managing the queue of work items for a thread.
public sealed class Dispatcher
{
    public static Dispatcher CurrentDispatcher { get; }
    public bool CheckAccess();
    public void VerifyAccess();
    public static void Run();
    public Thread Thread { get; }

    public void BeginInvokeShutdown(DispatcherPriority priority);
    public void InvokeShutdown();
    public bool HasShutdownFinished { get; }
    public bool HasShutdownStarted { get; }
    public event EventHandler ShutdownStarted;
    public event EventHandler ShutdownFinished;

    public DispatcherOperation BeginInvoke(Delegate method, params object[] args);
    public DispatcherOperation BeginInvoke(DispatcherPriority priority, Delegate method);
    public DispatcherOperation BeginInvoke(Delegate method, DispatcherPriority priority,
                                           params object[] args);
    public DispatcherOperation BeginInvoke(DispatcherPriority priority, Delegate method,
                                           object arg);
    public DispatcherOperation BeginInvoke(DispatcherPriority priority, Delegate method,
                                           object arg, params object[] args);
    public object Invoke(Delegate method, params object[] args);
    public object Invoke(DispatcherPriority priority, Delegate method);
    public object Invoke(Delegate method, DispatcherPriority priority, params object[] args);
    public object Invoke(DispatcherPriority priority, Delegate method, object arg);
    public object Invoke(DispatcherPriority priority, Delegate method,
                         object arg, params object[] args);
    public object Invoke(Delegate method, TimeSpan timeout, params object[] args);
    public object Invoke(DispatcherPriority priority, TimeSpan timeout, Delegate method);
    public object Invoke(Delegate method, TimeSpan timeout, DispatcherPriority priority,
                         params object[] args);
    public object Invoke(DispatcherPriority priority, TimeSpan timeout, Delegate method,
                         object arg);
    public object Invoke(DispatcherPriority priority, TimeSpan timeout, Delegate method,
                         object arg, params object[] args);

    public DispatcherOperation InvokeAsync(Action callback);
    public DispatcherOperation InvokeAsync(Action callback, DispatcherPriority priority);
    public DispatcherOperation InvokeAsync(Action callback, DispatcherPriority priority,
                                           CancellationToken cancellationToken);

    public DispatcherOperation<TResult> InvokeAsync<TResult>(Func<TResult> callback);
    public DispatcherOperation<TResult> InvokeAsync<TResult>(Func<TResult> callback,
                                                 DispatcherPriority priority);
    public DispatcherOperation<TResult> InvokeAsync<TResult>(Func<TResult> callback,
                                                 DispatcherPriority priority,
```

```
                                                CancellationToken cancellationToken);
    public void Invoke(Action callback);
    public void Invoke(Action callback, DispatcherPriority priority);
    public void Invoke(Action callback, DispatcherPriority priority,
                       CancellationToken cancellationToken);
    public void Invoke(Action callback, DispatcherPriority priority,
                       CancellationToken cancellationToken, TimeSpan timeout);

    public TResult Invoke<TResult>(Func<TResult> callback, DispatcherPriority priority);
    public TResult Invoke<TResult>(Func<TResult> callback, DispatcherPriority priority,
                                   CancellationToken cancellationToken);
    public TResult Invoke<TResult>(Func<TResult> callback, DispatcherPriority priority,
                                   CancellationToken cancellationToken, TimeSpan timeout);
}
```

הסבר:

- CurrentDispatcher – מאפיין סטטי לקבלת המשגר של **<u>התהליכון הנוכחי</u>** וייצורו אם עוד לא קיים כפי שדיברנו קודם
- CheckAccess\VerifyAccess – פונקציות של המשגר שמופעלות בפועל ע"י פונקציות באותם שמות של DispatcherObject כפי שדיברנו קודם
- Run – הפעלת לולאת טיפול בתור הפריטים\האירועים של המשגר
- Thread – אובייקט של התהליכון שהמשגר מקושר אליו
- BeginInvokeShutdown/InvokeShutdown – פונקציות להעברת הוראת סיום ללולאת המשגר כפי שהזכרנו קודם
- HasShutdownStarted/HasShutdownFinished – מאפיינים שמראים האם התחיל\הסתיים טיפול בבקשת סיום לולאת המשגר
- ShutdownStarted/ShutdownFinished – events שמאפשרים למפתח להוסיף קוד בתחילת ובסיוף הטיפול בבקשת סיום לולאת המשגר
- BeginInvoke – פונקציות (ישנות) לשליחת בקשה אסינכרונית למשגר לביצוע של פונקציה (בלי או עם פרמטרים) בתוך התהליכון המקושר למשגר לפי עדיפות מסוימת או בעדיפות רגילה
  - מחזיר אובייקט DispatcherOperation, בעזרתו ניתן לבצע כמה פעולות לאחר שליחת הבקשה:
    - לשנות את העדיפות של הבקשה
    - לבטל את הבקשה
    - להמתין להשלמת הטיפול בבקשה
    - לקבל את תוצאת הבקשה – הערך שמוזחר מתוך הפונקציה שביקשנו לבצע
  - מכיוון שהפרמטר מוגדר כ-Delegate, נצטרך לעשות המרה מפורשת לסוג דלגט המתאים – גם בהעברת שם פונקציה וגם בשימוש בביטוי לאמבדה – הדבר מסרבל את הקוד ולכן נוצרו פונקציות InvokeAsync
- Invoke עם פרמטר method מסוג Delegate – פונקציות (ישנות) שדומות ל-BeginInvoke לשליחת בקשה סינכרונית (זאת אומרת עם המתנה לסיום ביצועָה) לביצוע של פונקציה (בלי או עם פרמטרים) בתוך התהליכון המקושר למשגר לפי עדיפות מסוימת או בעדיפות רגילה, מחזירה את הערך שמוחזר מהפונקציה
- InvokeAsync - פונקציות לשליחת בקשה אסינכרונית למשגר לביצוע של פונקציה (בלי פרמטרים) בתוך התהליכון המקושר למשגר לפי עדיפות מסוימת או בעדיפות רגילה, מחזירה אובייקט של DispatcherOperation בדומה ל-BeginInvoke, הפונקציה יכולה להחזיר ערך רק בשימוש בפונקציה גנרית InvokeAsync<TResult>
- Invoke עם פרמטר callback מסוג Action – פונקציות שדומות ל-InvokeAsync לשליחת בקשה סינכרונית (זאת אומרת עם המתנה לסיום ביצועָה) לביצוע של פונקציה (בלי פרמטרים) בתוך התהליכון

המקושר למשגר לפי עדיפות מסוימת או בעדיפות רגילה, מחזירה את הערך שמוחזר מהפונקציה - רק בשימוש בפונקציה גנרית InvokeAsync<TResult>

יישום בהדמיית חשבון

לפי כך, נשתמש בפונקציות הנ"ל על מנת לוודא שהקוד שנדרש להתבצע בתהליכון UI בלבד (קוד שניגש ל- או משנה את הפקדים) יישלח למשגר של הפקדים האלה.

```
private void updateBalance(int balance)
{
    txtBalance.Text = String.Format("{0}", balance);
}

private bool updateBalanceCond(int balance)
{
    updateBalance();
    return balance < 500;
}
```

לצורך השימוש הנכון אנחנו נעשה הכנה. אנחנו נשנה את שם הפונקציה (האות הראשונה לקטנה, פונקציה פרטית) ונעשה בעצם שני יישומים עבורה (לצורך עתידי). הפונקציה השנייה תחזיר **true** אם המאזן קטן מ-500.

```
public void UpdateBalance(int balance)
{
    if (CheckAccess())
        updateBalance(balance);
    else
        Dispatcher.BeginInvoke((Action<int>)
                  (x => updateBalance(x)), balance);
}
```

נכתוב עכשיו פונקציה שאותה נפעיל מ"המשקיף" שלנו והיא תבדוק האם אנחנו נמצאים בתוך התהליכון שיצר את הפקד ותפעל בהתאם. איך נעשה זאת? קודם כל נבדוק בעזרת הפונקציה **CheckAccess** של הפקד שלנו. אם ערכו יהיה **false** – פשוט נפעיל את הפונקציה הפנימית הנ"ל. אם ערכו יהיה **true** – נפעיל פונקציה **Invoke** ונעביר אליה את הפונקציה הפנימית דרך ביטוי למבדה והמרה ל-**Action** הגנרי המתאים. הפונקציה לא תתקמפל ללא המרה בגלל שהפרמטר שלה אינו מוגדר כדלגט מסויים אלא ע"י מילת מפתח **Delegate** של C#. שימו לב שהשתמשנו בהעמסה תוך העברת פרמטרים נוספים לדלגט. אם נרצה להעביר יותר פרמטרים, נרשום את כולם לפי הסדר ב-<> של תבנית ה-**Action** ונתאים את ביטוי הלמבדה. קריאה לפונקציה **BeginInvoke** גורמת להעברת הדלגט בהודעה למשגר של תהליכון UI והפעלתה בתהליכון הזה. הפונקציה הינה אסינכרונית – אין המתנה לסיום הביצוע שלה. אם נרוצה להמתין להשלמת המשימה בתהליכון שלנו – נשמור את הערך המוחזר מתוך **BeginInvoke** במשתנה מסוג **DispatcherOperation** ונפעיל ממנו את פונקציית ההמתנה:

```
BeginInvoke((Action<int>)(x => innerUpdateBalance(x)), balance)).Wait();
```

או פשוט על ידי הפעלת פונקציה **Invoke** אשר עושה בשבילנו את העבודה הנ"ל (זאת פונקציה סינכרונית):

```
Invoke((Action<int>)(x => innerUpdateBalance(x)), balance);
```

מה נוכל לעשות אם אנחנו זקוקים לתוצאה של ביצוע הדלגט? נוכל להשתמש בכל סוג דלגט אחר שמחזיר ערך – ונקבל אותו כערך מוחזר מ-**Invoke** (למשל נשתמש ב-Predicate שמחזיר ערך בולאני ונפעיל את הפונקציה הפנימית השנייה שבנינו קודם – שתחזיר לנו האם המאזן נהיה נמוך מ-500). נגדיר שדה **warned** ונקפיץ חלון הודעה בפעם שאנחנו עוברים ממאזן עם יותר או שווה מ-500 למאזן עם פחות מ-500.

```
private bool warned = false;
private void warnLowBalance(bool check)
{
    if (check)
        if (!warned)
        {
            warned = true;
            new Thread(() => MessageBox.Show("You are going low on balance!",
                                             "Account warning",
                                             MessageBoxButton.OK,
                                             MessageBoxImage.Exclamation)
                        ).Start();
        }
    else
        warned = false;
}
public void UpdateBalanceCond(int balance)
{
    if (CheckAccess())
        warnLowBalance(updateBalanceCond(balance));
    else
        warnLowBalance((bool)Dispatcher.Invoke(
                                 (Predicate<int>)(x => updateBalanceCond(x)), balance));
}
```

נוכל לבדוק עכשיו את התוכנית – שתעבוד יפה הפעם.

פונקציות חדשות ופשוטות יותר לשימוש ב-Dispatcher

כאשר מדובר בהפעלת פונקציות ללא פרמטרים (או שמקבלים את המידע בדרך עקיפה), נשתמש בפונקציות החדשות (הקוד נראה הרבה יותר טוב בלי המרות מיותרות!):

```
private void myAction() { }
public void MyAction()
{
    if (CheckAccess())
        myAction();
    else
        Dispatcher.InvokeAsync(myAction);
}
private int myFunc() { return 0; }
public int MyFunc()
{
    if (CheckAccess())
        return myFunc();
    else
        return Dispatcher.Invoke<int>(myFunc);
}
```

נספח א – השלמת קלט ב-TextBox ללא כפתור

מה שחסר זה רק יישום הפקדים של עריכה עבור הפקדה ומשיכה. ב-WPF אין אירוע מיוחד לסיום קלט בתיבת טקסט. לכן ניתן להשתמש בכפתור ליצירת אירוע השלמת קלט. כמו כן נרצה לוודא שמקלידים רק ספרות...

ועוד – אנחנו רוצים לוודא שבדות לקלט מספרים מקלידים רק ספרות. מכיוון שעוד לא למדנו DataBinding וכללי בדיקה (ValidationRule), נרצה להתערב תוך כדי הקלדה בשדה TextBox.

אפשרות שתעזור לנו לעשות זאת היא ליצור פונקציה לאירוע של לחיצת מקש בשדה. הפונקציה מקבלת פרמטר `KeyEventArgs` שמכיל את הקוד ואת התו של המקש שנלחץ. נשתמש באירוע המקדים בשביל לתפוס אותו בשלב מוקדם - **PreviewKeyDown**. הקוד מובא להלן. נשתמש בפונקציה לשתי השדות – של ההפקדה ושל המשיכה. משימה נחמדה ללמוד איך לעשות את הפונקציה ואיך להשתמש בפונקציות סטטיות של מחלקת **Char**. שימו לב שערך טקסט של פקד העריכה אינו כולל את תוצאות הלחיצה האחרונה. התו יתווסף לפקד אם נחזיר בפרמטר הנ"ל שדה **Handled** שווה ל-**false**. זאת אומרת – לא סיימנו טיפול בתו ואנחנו מבקשים ממערכת החלונות להשלים את הטיפול. במקרה הפשוט – הוספת התו לטקסט בפקד העריכה. אנחנו נאפשר למערכת להשלים טיפול רק עבור ספרות ומקשי בקרה (חצים, מחיקה, וכו'). אחרת נחזיר **true**. אם נלחץ המקש "**Enter**" – נעשה עדכון בחשבון בנק ונמחק את תוכן הפקד. וכמובן נחזיר **true**. הקוד להלן. תרגיל נחמד לשיפור שליטה ב-WPF (הקוד בעמוד הבא).

```
private void textBox_PreviewKeyDown(object sender, KeyEventArgs e)
{
    TextBox text = sender as TextBox;
    if (text == null) return;
    if (e == null) return;

    if (e.Key == Key.Enter || e.Key == Key.Return)
    {
        if (text.Text.Length > 0)
        {
            int amount = int.Parse(text.Text);
            text.Text = "";
            if (sender == txtDeposit)
                myAccount.Deposit(amount);
            else if (sender == txtWithdraw)
                myAccount.Withdraw(amount); // Can handle no money here…
        }
        e.Handled = true;
        return;
    }
    // It`s a system key (add other key here if you want to allow)
    if (e.Key == Key.Escape || e.Key == Key.Tab || e.Key == Key.Back ||
        e.Key == Key.Delete || e.Key == Key.CapsLock ||
        e.Key == Key.LeftShift || e.Key == Key.RightShift ||
        e.Key == Key.LeftCtrl || e.Key == Key.RightCtrl ||
        e.Key == Key.LeftAlt || e.Key == Key.RightAlt ||
        e.Key == Key.LWin || e.Key == Key.RWin || e.Key == Key.System ||
        e.Key == Key.Left || e.Key == Key.Up ||
        e.Key == Key.Down || e.Key == Key.Right) return;

    char c = (char)KeyInterop.VirtualKeyFromKey(e.Key);
    if (Char.IsControl(c)) return;
    if (Char.IsDigit(c))
    {
        if (!(Keyboard.IsKeyDown(Key.LeftShift) || Keyboard.IsKeyDown(Key.RightShift) ||
              Keyboard.IsKeyDown(Key.LeftCtrl) || Keyboard.IsKeyDown(Key.RightCtrl) ||
              Keyboard.IsKeyDown(Key.LeftAlt) || Keyboard.IsKeyDown(Key.RightAlt)))
            return;
    }

    e.Handled = true;
    MessageBox.Show("Only numbers are allowed", "Account",
                    MessageBoxButton.OK, MessageBoxImage.Error);
}
```