גירסה 1.01 – 24.3.2010
גירסה 1.00 – 1.1.2005

שפת C#
תכנות יעיל בשפה

Nir Adar

Email: nir@underwar.co.il

Home Page: http://www.underwar.co.il

אנא שלחו תיקונים והערות אל המחבר.

תוכן עניינים

פתיחה

מסמך זה מרכז מספר עצות בהן ניתן להשתמש על מנת לכתוב תוכניות יעילות יותר ונכונות יותר ב-C#. תוכניות ניתנות לכתיבה בדרכים שונות, ושפת C# נותנת לנו כלים רבים. נציג במסמך מספר דוגמאות מתי נעדיף שימוש בכלי/תכונה זו או אחרת במהלך העבודה.
קוד הכתוב בשפת C# רץ על מכונה וירטואלית – סביבת העבודה של Net. חברת Microsoft ידועה בעולם כמייצרת מכונות וירטואליות מהירות ביותר ביחס למתחרים, אך עדיין קיימת ירידה בביצועים ביחס לתוכנות המתאימות למעבד מסויים, ולפיכך חשוב להשקיע בכתיבת התוכנית בצורה יעילה

מסמך זה מרכז למעשה שני נושאים שאין ביניהם קשר ישיר:

1. ביצועים בסביבת Net.
2. עקרונות תכנות נכון בסביבת Net.

בחרתי במסמך זה לכלול את שני נושאים אלו, ואף לשלבם ללא הפרדה מיוחדת ביניהם עקב חשיבותם של נושאים אלו. כל מתכנת ב-C# חשוב שיכיר את שני הנושאים.

שפת C# מתפתחת, וייתכן שאפילו בשנה הקרובה, עקב שיפור המימוש של סביבת Net. הבאה, חלק מהעצות הניתנות במסמך זה יהפכו למיותרות. עם זאת, מטרת המסמך היא לעורר מודעות לנושאים אלה, ולחשיבות שלהם. מסמך זה נכתב אודות סביבת Net 2003.

כשחושבים על ביצועים

כשמדובר בביצועי התוכנית, עלינו לזכור שני חוקים מרכזיים:

1. **מדידות**: הגדר לפי מה תימדד היעילות של התוכנית (מספר פלטים לשניה, זמן תגובה וכדו'). הגדר את הקריטריונים כדי שהתוצאה תחשב תוצאה טובה.
   כללי אצבע:
   - אם לא נבדוק את ביצועי התוכנית, הם יהיו גרועים.
   - ללא מדידות של ביצועי התוכנה, העבודה עליה לא הסתיימה.
2. **הכרת התחום**: עשה שיעורי בית. הכר את הטכניקות בהם אתה הולך להשתמש לפי התחלת הפרויקט עצמו. ברר מהן אופציות המפתח של סביבת העבודה, סביבת היעד, המעבד בהם אנחנו משתמשים. שים לעובדה שלא כל טכנולוגיה מתאימה בכל מצב שימוש בקבצי XML, למשל, הוא מאוד נוח בסביבת Net. אולם בתוכנות הדורשות מהירות תגובה מקסימלית, לא בטוח שנרצה להשתמש בטכנולוגיה זו.

אחרי שנגדיר את דרישות הפרויקט, יש לתכנן כיצד נעמוד בהן.
במקרים רבים פרויקטים משתמשים באלגוריתמים בסיבוכיות גבוהה – שמראש לא עומדים ביעדים שהוצבו. למשל, תוכנית המשתמשת במיון בועות, כאשר היתה יכולה להשתמש בחיפוש מהיר. במקרים כאלה – הגישה של "נשפר אחר כך" היא לא תמיד טובה, מכיוון שאנחנו מבצעים עבודה מיותרת שתזרק בהמשך וגם שיפור שלה על ידי קבוע (מעבד מהיר יותר, פחות חישובי ביניים) לא יעזור לעומת ביצוע אותה העבודה בצורה נכונה.
בעיות אלו נפוצות בפרויקטים קיימים רבים.

סביבת Net. מפשטת דברים רבים ומאפשרת לאלגוריתמים יותר גרועים לעבוד. לדוגמא: האפשרות לשרשר מחרוזות על ידי האופרטור + יוצרת בעיה במקרה שמשרשרים מחרוזות רבות. בכל פעולת שרשור מוקצה מחרוזת חדשה, וביצועי התוכניות מושפעים. למרות הקלות בכתיבת קוד שהשפה נותנת לנו, עדיין צריך לחשוב על ביצועים, ולהעדיף אלגוריתמים יעילים ככל האפשר.

Net. מכיל מחלקות כלליות רבות – רשימות, טבלאות ערבול וכו'. הכלליות עולה ביעילות הקוד. במקרים בהם הביצועים קריטיים – נמנע מכלליות ונכתוב דברים המתאימים לאותו מקרה.

דברים אותם רצית לדעת על שפת C# ולא ידעת את מי לשאול

בפרק זה אציג אוסף של עצות שימושיות לגבי תכנות בשפת C#.
כשהתחלתי לתכנת בשפת C# היא הזכירה לי מאוד את Java ו-C++, והשאלה שהתעוררה אצלי היא "במה השפה הזו שונה? איך אני צריך לכתוב כדי לפעול לפי הרוח המנחה את השפה? מהם ההבדלים הקטנים אבל המשמעותיים בינה לבין השפות האחרות?".
בפרק זה אנסה להציג את התשובות שמצאתי. התשובות מחולקות לפי הנושאים השונים בשפה.

מחרוזות

בדיקה האם המחרוזת ריקה

נביט בקוד הבא:

```
string s = "whatever";
if (s == "") Console.WriteLine("Empty string");
else Console.WriteLine(s);
```

קוד זה מבצע בדיקה האם המחרוזת s ריקה על ידי השוואתה למחרוזת "". צורת עבודה זו הינה האינטואיטיבית ביותר, והיא אכן עובדת, אולם היא אינה הדרך היעילה ביותר לעשות השוואה זו.

נחליף לדוגמא את הקוד בקוד הבא:

```
string s = "whatever";
if (s.Length == 0) Console.WriteLine("Empty string");
else Console.WriteLine(s);
```

על ידי שימוש ב-Length במקום השוואה למחרוזת שנייה יווצר הרבה פחות קוד MSIL וביצועי התוכנית ישתפרו.

במקרה הראשון נקרא ה-constructor של string עם "" כדי ליצור אובייקטים ברי השוואה. לאחר מכן נקרא האופרטור == של string כדי להשוות בין האובייקטים. במקרה השני פעולות אלו אין קורות. בדומה לשפת C++, השפה מבצעת לעתים פעולות השקופות למתכנת, שעלותן גבוהה. עלינו להזהר ולצמצם מקרים אלו.

שרשור מחרוזות

במידה ואנחנו משרשרים מספר מחרוזות קבוע נשתמש בטיפוס string עם האופרטור +.
StringBuilder זוהי מחלקה המאפשרת שרשור מחרוזות ועדכון מחרוזת בצורה יעילה.
במידה ואנחנו רוצים לשרשר מספר לא ידוע של מחרוזות, נשתמש במחלקה StringBuilder כדי לשמור את התוצאה, וכל פעם נוסיף את המחרוזת הבאה בעזרת קריאה לפונקציה ()Append.
התוצאה הינה שיפור בסדר גודל! שימוש ב-()append נעשה בזמן של $O(1)$ ואילו שימוש באופרטור + נעשה בזמן $O(n)$ (כאשר n הוא אורכה של המחרוזת החדשה).

מערכים

בניגוד לשפות כגון C/C++ בה יצירת מערך חדש מתבצעת ב-$O(1)$, יצירת מערכים בסביבת Net.
מתבצעת בזמן של $O(n)$, כאשר n הוא אורך המערך. סביבת Net. מאתחלת כל מערך מיד עם יצירתו.
לפיכך: יש לזכור כי פעולת יצירת מערך הינה פעולה יקרה. כמו כן, יש להשתמש בעובדה שהמערך כבר נוצר מאותחל ולהימנע מאתחול כפול. במידה ונרצה לשים במערך ערכים התחלתיים נשים אותם במידת האפשר מיד עם יצירת המערך (באתחול), ולא בעזרת השמה, כדי לחסוך אתחול מיותר.

Exceptions

שימוש ב-ApplicationException במקום במחלקה Exception

בדומה ל-Java, שפת C# מכילה מחלקה בשם Exception ממנה נגזרות המחלקות השונות.
עם זאת בניגוד ל-Java, גזירת מחלקה חדשה ממחלקה זו איננה צורת העבודה התואמת את רוח השפה.
כל ה-exceptions של קוד המשתמש שאינן גורמות להפסקת התוכנית מיידית צריכות להגזר מהמחלקה ApplicationException ולא מהמחלקה Exception.

זריקת מספר מועט של Exceptions ככל שניתן

העלות של זריקת exception היא עצומה, לכן יש לדאוג בקוד שמספר ה-Exceptions שייזרקו בפועל יהיה קטן ככל שניתן.

אין הכוונה להמנע משימוש ב-exceptions. מנגנון זה הוא חשוב בשפה ומוסיף לה כוח רב. הכוונה היא להמנע מלנהל את זרימת התוכנית בעזרת exceptions.

נשים לב שבעוד זריקת exception הינה פעולה יקרה, עטיפת הקוד ב-try, catch כמעט ואינה משפיעה על הביצועים, ולכן לא מומלץ להמנע מלתפוס שגיאות אפשריות

int.Parse

בהקשר לטיפ הקודם, נתייחס במיוחד להמרת מחרוזת ל-int. הפקודה המשמשת לרוב לביצוע משימה זו היא int.Parse המקבלת מחרוזת ומחזירה את המספר המתאים לו. במידה ואין מספר כזה, היא זורקת שגיאה.

אנשים רבים משתמשים במתודה זו, ותופסים את ה-exceptions. האם זו הדרך היעילה?

מהן הדרכים החלופיות לשימוש בפונקציה זו?

1. **שימוש בביטוי רגולארי** לבדיקת הביטוי לפני ההמרה, למשל על ידי:

```
public static bool IsNumeric(string sNumber)
{
      Regex reg = new Regex(@"^(\d)+$");
      return reg.Match(sNumber).Success;
}
```

מבדיקה, שימוש בקוד זה יעיל <u>פחות</u> מהשיטה הישירה של תפיסת ה-catch!.

2. **שימוש ב-double.TryParse** – ל-double יש מתודה המבצעת המרת מחרוזת למספר, שאינה זורקת exception. גם השימוש במתודה זו אינו מומלץ, מכיוון שבבדיקה שנעשתה נראה כי היא איטית פי 30 (!!) מהמתודה int.Parse().

3. **כתיבת פונקציה חדשה שתבדוק האם המחרוזת מייצגת מספר** – שיטה זו מסתברת כשיטה היעילה ביותר הקיימת כעת. נכתוב קוד שיבצע משימה זו בצורה יעילה.
בנספח 1 של מסמך זה נציג את הפונקציה PreliminaryCheck המבצעת בדיקה כזו. שימוש בה לפני הקריאה ל-Parse, במקום שימוש ישיר ב-Parse ותפיסת Exception, הביא לקוד שרץ פי 100 יותר מהר!

מסקנה: בקטעי קוד רגילים ניתן להשתמש ב-int.Parse מכיוון שהיא אינה מעכבת באופן יוצא דופן את התוכנית. בקטעי קוד קריטיים נכתוב פונקציה ייעודית לביצוע המשימה כדי לחסוך את זריקת ה-exceptions.

פונקציות ומאפיינים

המנעות מקריאות מרובות לפונקציות קצרות

בסביבת Net. העלות של כניסה ויציאה מפונקציה היא גבוהה– לכן נעדיף להמנע מפונקציות קצרות הנקראות לעיתים תכופות – נעדיף לאחד פונקציונליות לפונקציות מורכבות יותר.

הנושא נוגע בעיקר לפונקציות הקובעות,למשל, שדה אחד בתוך אובייקט. במידת האפשר וההגיון, נעדיף ליצור פונקציות הקובעות מספר שדות בו זמנית.

בשפת C++ היה ניתן להשתמש במילה השמורה inline כדי לטפל במקרים אלה. לסביבת Net 1.1. אין כרגע פתרון שיטתי לייעול הקריאה לפונקציות קצרות.

שימוש ב-Properties לעומת שימוש בפונקציות

שפת C# מציעה מנגנון מעולה של properties כדי להחליף את הכמות האדירה של פונקציות get/set הקיימות בשפות אחרות.

בשפת C# השימוש ב-properties כל הזמן נחשב לדרך התכנות הנכונה ביותר.

עם זאת, יש לשים לב ששימוש ב-property יקר יותר מבחינת זמן מאשר שימוש בפונקציה, ולכן בקטעי קוד הדורשים מהירות קריטית כדאי כן להשתמש בפונקציות.

הערה נוספת לגבי Properties והעלות שלהם, היא שבמידה ואנחנו ניגשים לאותו Property שוב ושוב (למשל, בלולאה בה אנחנו ניגשים ל-Property בשם Length של אובייקט מסויים) – מומלץ למשל לשמור את ערך ה-Property במשתנה לפני תחילת הלולאה, וכך לחסוך את הקריאות המיותרות.

לעתים המהדר יידע לבצע פעולה זו בעצמו, אך לא תמיד, ולפיכך נבצע אותה אנו במידת האפשר.

מחזור החיים של אובייקט

המנעות משימוש ב-destructors

במידת האפשר, יש להמנע משימוש ב-destructors בעת התכנות בשפת C#.

אובייקטים רגילים ב-C#, מיד עם סיום העבודה איתם, מפונים בשקט על ידי ה-GC.

אובייקטים להם מוגדר destructor ממתינים בצד עד אשר ה-GC אוסף אותם, ואז מתעוררים שוב לבצע את קוד ה-destructor שלהם. מכיוון שבהתאם לאופי השפה – הרגע המדויק בה פעולה זו תקרא איננו

ידוע, יתכנו תוצאות בלתי צפויות שונות. כמו כן, ביצועי התוכנית יכולים בפתאומיות לרדת, מכיוון שה-GC החליט כי הגיע הזמן לפעול, וקוד ה-destructors יורץ על חשבון זמן הריצה של התוכנית.

בשפת C# קיים בכל מקרה צורך פחות ב-destructor מאשר בשפת C++, למשל. הסיבה לכך היא שבשפת C++ עיקר העבודה של ה-destructor היתה שחרור הזכרון בסוף העבודה עם האובייקט. בשפת C# הזיכרון מנוהל על ידי GC ולכן ברוב המקרים כלל אין צורך בפונקציה הורסת.

השמת null בתוך אובייקטים שהעבודה עימם נגמרה

ה-GC עושה חיים קלים יותר למתכנת מכיוון שהמתכנת אינו צריך לדאוג יותר לרגע המדויק בו ישתחרר האובייקט. עם זאת, כצורת תכנות נכונה, יש לשים באובייקט בו סיימנו להשתמש את הערך null.
בצורה זו אנו אומרים ל-GC שהעבודה על האובייקט הסתיימה, וה-GC יכול לעבוד בצורה יעילה יותר.

לולאות

העדפת לולאת for על לולאת foreach בקטעים קריטיים

נביט בקוד הבא:

```
static private void TestForeachLoop(ArrayList List)
{
	int id = 0;
	foreach (int i in List)
	{
		// do something with the object
		id = i;
	}
}

static private void TestForLoop(ArrayList List)
{
	int id = 0;
	int count = List.Count;

	for (int i = 0; i < count; ++i)
	{
		id = (int)List[i];
	}
}
```

במקרה של לולאת ה-foreach, נוצר int חדש בכל איטרציה של הלולאה, המשמש כמשתנה באותה איטרציה. בלולאת ה-for, נשמר המשתנה היחיד לאורך כל הלולאה.
התוצאה: לולאת ה-for מהירה ב-33%! עבור 100,000 איברים מדובר בזמן של 10 שניות לעומת זמן 15 שניות.
נשים לב לחשיבות של השמת count, על מנת להימנע שוב ושוב מקריאה ל-property.

לולאות המייצרות קבועים

לעתים מתכנתים שמים בקוד שלהם לולאות שהפלט שלהן הוא קבוע– למשל – לולאה המייצגת mask של ביטים לצורך חישוב. מומלץ ב-C#, כמו בכל שפה אחרת, להימנע מלולאות כאלו ולהגדיר את הקבועים המתאימים. על ידי כך נמנע לולאה מיותרת, וכן נאפשר למהדר לבצע אופטימיזציות על התוכנית ואולי אף לבצע חלק מהחישובים בזמן ההידור.

שונות

Structs לעומת Classes

בשפת C#, אוביקטים מסוג class מוקצים על הערימה ואילו אובייקטים מסוג struct מוקצים על המחסנית. נוח יותר לעבוד עם class, אולם עבור מבנים פשוטים, במקרים בהם אין פעולות boxing/unboxing רבות, נעדיף להשתמש ב-struct על פני class עקב המהירות העדיפה של הגישה אל המחסנית. בדוגמא הבאה, הלולאה על ה-class לוקחת פי 5 בערך יותר זמן מאשר הלולאה על ה-struct. (0.5 שניות לעומת 2.4 שניות על Pentium4-2400).

```
using System;
namespace StructObjectsTests
{
	public struct foo
	{
		public foo(double arg){ this.y = arg; }
		public double y;
	}
	public class bar
	{
		public bar(double arg){ this.y = arg; }
		public double y;
	}
	class Class1
	{
		static void Main(string[] args)
		{
```

```
                    System.Console.WriteLine("starting struct loop...");

                    for(int i = 0; i < 50000000; i++)
                    {foo test = new foo(3.14);}
                    System.Console.WriteLine("struct loop complete.
                    starting object loop...");

                    for(int i = 0; i < 50000000; i++)
                    {bar test2 = new bar(3.14); }

                    System.Console.WriteLine("All done");
            }
        }
}
```

זמן זה כסף

בפרק זה נדון בשיטות הבאות לשפר את ביצועי התוכנית נציג מספר נקודות חולשה בשפה, וכן דרכי תכנות הבאות לשפר את מהירות ריצת התוכנית.
השיפורים שנציג אינם שיפורים בסיבוכיות – תחום זה הינו רחב וכולל אלגוריתמים חכמים ומבני נתונים. הדגש הוא שגם אם בחרנו את מבני הנתונים הנכונים והמתאימים ביותר למטרה, עדיין ניתן לשפר ביצועי התוכנית באלפי אחוזים על ידי תכנות נכון. בפרק זה נציג חלק מהשיפורים האפשריים.

ייתכן שעבור הפרויקט שלך, הנקודות הקריטיות הן אחרות. על מנת להשיג באמת ביצועים טובים, יש למדוד. תוכנות שונות (profilers) מספקות אינדיקציה כמה זמן התוכנית מבלה בכל קטע וקטע Profiler מומלץ הינו ANTS Profiler. תוכנה זו כוללת מספר תכונות, ביניהן אפשרות להציג כמה זמן התוכנית שלך בילתה בכל אחת מהפונקציות המרכיבות אותה, ויותר מזה, בתוך כל פונקציה, כמה זמן הוקדש לכל שורת קוד. על ידי בדיקה, ניתן לאתר את הנקודות הבעייתיות בפרויקט ולטפל בהן.

Collections

עלויות נסתרות בשימוש במבני הנתונים הגנריים

כאשר אנחנו שומרים משתנים שהם ערכים (values) כגון int, double וכו' בתוך מבני הנתונים הכלליים, מתבצעת עליהם פעולת boxing – עטיפתם על ידי אובייקט. כאשר אנחנו מוציאים נתונים אלו מהמבנה, מתבצע unboxing על מנת שנוכל לקבל שוב את הערך המבוקש.
פעולה זו לוקחת זמן. בנספח 2 מודגמת תוכנית המכניסה לתוך רשימה משתנים מסוג int, ולאחר מכן מכניסה לתוך רשימה משתנים מסוג string (שהוא משתנה התייחסות ולא משתנה ערך).
עבור משתני התייחסות, לא מבוצעת פעולה נוספת מלבד שמירתו במבנה הנתונים.
לפיכך, בבדיקה שבוצעה, קטע הקוד שהכניס את המחרוזות פעל פי 4 מהר יותר מקטע הקוד שהכניס את השלמים לרשימה.

שיפור העבודה עם hash

כאשר ניגשים לאיבר ב-hash table שאינו קיים, מוחזר הערך null. נוכל לנצל עובדה זו על מנת לייעל את הגישות שלנו אל hash tables. לרוב, כאשר לא מודעים לנושא זה, נכתוב קטע קוד כגון:

```
if (MyHash.ContainsKey(key))
{
      myData = (MyCasting)MyHash[key];
      // ...
}
```

ראשית נבדוק האם האיבר שאנו רוצים לגשת אליו קיים על ידי ContainsKey(), ואם כן, נקרא את הערך שלו על ידי שליפה מה-hash. אנחנו עושים זאת מכיוון שפעולת casting על null תגרום לזריקת exception.

שיפור לקוד שהוצג יכול להיות הקוד הבא:

```
object TheData;
if ( (TheData = MyHash[key]) != null)
{
      myData = (MyCasting)TheData;
}
```

אנו קוראים את הערך מה-hash לתוך משתנה התייחסות מסוג object, ולאחר מכן מבצעים בדיקה אם ערך זה הוא null. במידה ולא, אנחנו מבצעים unboxing ומקבלים את הערך שלנו.
בדוגמא זו, במקום שתי קריאות למתודות של ה-hash, ביצענו קריאה אחת והשתמשנו ב-unboxing.

התוצאה: קוד זה מהיר פי 2 מהקוד הראשון. צוואר הבקבוק במקרה זה הינו הגישה אל ה-hash. שיפור קטן זה הכפיל את ביצועי הקוד.

שימוש ב-AddRange

כאשר נאחד קבוצות של איברים נשתמש ב-AddRange אם הפקד תומך בכך, ולא נשתמש בפונקציה Add כדי להוסיפם אחד אחרי השני.
AddRange מותאם להוספה מהירה של מספר איברים ל-Collection בעוד ש-Add מותאמת להוספה מהירה של איבר בודד.

שימוש בביטויים רגולריים

ביטויים רגולריים הם כלי נוח מאוד לביצוע פעולות על מחרוזות, אולם העלות שלהם במקרה של ביטויים מורכבים היא אדירה. לדוגמא:

הביטוי הבא נכתב במהלך כתיבת אסמבלר, כדי למחוק את ההערות של המשתמש, המתחילות ב-# מהקוד, לפני הפיכת הקוד לשפת מכונה.
הביטוי הבא מוצא את כל המקומות בהם יש #, אבל לא בתוך מחרוזת (המוגבלת על ידי ""), ומוחק את השורה מאותה נקודה ואילך.

```
sAssemblyCode = Regex.Replace(sAssemblyCode,
@"([^""#\n]*?""[^""\n]*(""""")*[^""\n]*""[^""#\n]*)+(#([^\n])*)",
@"$1", RegexOptions.Multiline);
```

הביטוי הבא מטפל בכל ההערות הנמצאות בשורת בהן אין מחרוזות

```
sAssemblyCode = Regex.Replace(sAssemblyCode,@"(^[^""#\n]*?)(#.*)",
"$1", RegexOptions.Multiline);
```

לאחר בעיה של ביצועים, שני ביטויים אלו הוחלפו בלולאה הבאה:

```
string[] tempCommentsString = sAssemblyCode.Split(new char[] {'\n'});
for (int i = 0; i < tempCommentsString.Length; ++i)
{
      int InvCommaCounter = 0;
      for (int j = 0; j < tempCommentsString[i].Length; ++j)
      {
            if (tempCommentsString[i][j] == '"') InvCommaCounter++;
            else if (tempCommentsString[i][j] == '#' &&
                  InvCommaCounter % 2 == 0)
                  tempCommentsString[i] =
                        tempCommentsString[i].Substring(0, j);
      }
}
sAssemblyCode = String.Join("\n", tempCommentsString);
```

התוצאה: בהרצה על קובץ טקסט בגודל 16KB על Pentium4-2400: הלולאה הסתיימה תוך 0.001 שניות, והביטויים הרגולריים תוך 3.7 שניות!.

נספחים

נספח 1 – פונקציה יעילה לבדיקה האם מחרוזת כלשהי מכילה מספר

הקוד הבא מציג מתודה בשם PreliminaryCheck הבודקת אם מחרוזת נתונה מייצגת int חוקי. שימוש במתודה זו לפני int.Parse וויתור על שימוש ב-exception יכול במקרים קיצוניים להביא לשיפור של פי 100 בביצועי התוכנית.

```
using System;


public class Test
{

	//
	// The methods
	//
	private static readonly char[] TrimStartArray = {'0'};
	private static bool PreliminaryCheck(string x)
	{
		// Check for being too short
		if (x == null || x.Length == 0) return false;

		// Check for being too long, remembering that 000000000000123,
		// +00000000000000000123 and -000000000000123 are all valid.
		if (x.Length > 10)
		{
			bool hasSign=false;
			if (x[0]=='+' || x[0]=='-') hasSign = true;

			if (!hasSign || x.Length>11)
			{
				string tmp = hasSign ? x.Substring(1) : x;
				if (tmp.TrimStart(TrimStartArray).Length > 10)
return false;
			}
		}

		// Check each character

		for (int i=0; i < x.Length; i++)
		{
			char c = x[i];
			if ((c < '0' || c > '9') &&
				((c!='-' && c!='+')|| i>0))
				return false;
		}
		return true;
	}
```

```
        //
        // Test Program
        //
        private static readonly string[] TestStrings =
    {
                "-12345", "667890", "12345678901234567890",
           "foo", "123456789-", "-0000000012345678910",
           "000000000012345678910123 45"};

        const int Iterations = 100000;

        public static void Main()
        {

        {
               DateTime start = DateTime.Now;
               long total=0;

               for (int i=0; i < Iterations; i++)
               {
                      for (int j=0; j < TestStrings.Length; j++)
                      {
                             try
                             {
                                    total += Int32.Parse(TestStrings[j]);
                             }
                             catch
                             {
                             }
                      }
               }
               DateTime end = DateTime.Now;

               Console.WriteLine ("No extra checking: {0}", end-start);
               Console.WriteLine ("(Total={0})", total);
        }

        {
               DateTime start = DateTime.Now;
               long total=0;

               for (int i=0; i < Iterations; i++)
               {

                      for (int j=0; j < TestStrings.Length; j++)
                      {
                             try
                             {
                                    string test = TestStrings[j];
                                    if (PreliminaryCheck(test))
                                           total += Int32.Parse(test);
                             }
                             catch
                             {
                             }
                      }
               }
               DateTime end = DateTime.Now;
               Console.WriteLine ("With preliminary checking: {0}",
                      end-start);

               Console.WriteLine ("(Total={0})", total);
        }

        }

}
```

נספח 2 – עלויות נסתרות בשימוש ב-collections

קטע הקוד הבא מדגים הכנסת שלמים לרשימה מול הכנסת מחרוזות לרשימת הכנסת ה-int התבצעה פי 4 לאט יותר מהכנסת המחרוזות.

```
using System;
using System.Collections;

namespace test1
{
	class Class1
	{
		static void Main(string[] args)
		{
			str_test();
			int_test();

		}

		static void str_test()
		{
			int count;
			ArrayList myArrayList = new ArrayList();

			// Construct 1000000 strings
			string [] strList = new string[1000000];
			for(count = 0; count < 1000000; count++)
				strList[count] = count.ToString();

			// Repeat test 5 times.
			DateTime startTime = DateTime.Now;
			for(int retry = 5; retry > 0; retry--)
			{
				myArrayList.Clear();

				// Add 'Value Types' to array the ArrayList.
				for(count = 0; count < 1000000; count++)
					myArrayList.Add(strList[count]);

				// Retrieve the values.
				string s;
				for(count = 0; count < 1000000; count++)
					s = (string) myArrayList[count];
			}

			// Print results.
			DateTime endTime = DateTime.Now;
			Console.WriteLine("Start: {0}\nEnd: {1}\nElapsed: {2}",
				startTime, endTime, endTime-startTime);
			Console.WriteLine("Ready. Push ENTER to finalize...");
			Console.ReadLine();
		}


		static void int_test()
		{
			int count;
```

```
                    DateTime startTime = DateTime.Now;
                    ArrayList myArrayList = new ArrayList();

                    // Repeat test 5 times.
                    for(int retry = 5; retry > 0; retry--)
                    {
                           myArrayList.Clear();

                           // Add 'Value Types' to array the ArrayList.
                           for(count = 0; count < 1000000; count++)
                                  myArrayList.Add(count);

                           // Retrieve the values.
                           int i;
                           for(count = 0; count < 1000000; count++)
                                  i = (int) myArrayList[count];
                    }

                    // Print results.
                    DateTime endTime = DateTime.Now;
                    Console.WriteLine("Start: {0}\nEnd: {1}\nElapsed: {2}",
                           startTime, endTime, endTime-startTime);
                    Console.WriteLine("Ready. Push ENTER to finalize...");
                    Console.ReadLine();
             }
      }
}
```

EOF